Verilog-to-Routing Documentation

Release 8.1.0-dev

VTR Developers

Feb 25, 2022
QUICK START

1 VTR Quick Start 3
1.1 Setting Up VTR 3
1.2 Running VPR 4
1.3 Running the VTR Flow 6
1.4 Next Steps 16

2 VTR 17
2.1 VTR CAD Flow 17
2.2 Get VTR 19
2.3 Building VTR 21
2.4 Optional Build Information 22
2.5 Running the VTR Flow 27
2.6 Benchmarks 28
2.7 Power Estimation 32
2.8 Tasks 44
2.9 run_vtr_flow 48
2.10 run_vtr_task 51
2.11 parse_vtr_flow 53
2.12 parse_vtr_task 54
2.13 Parse Configuration 55
2.14 Pass Requirements 56
2.15 VTR Flow Python library 57

3 FPGA Architecture Description 59
3.1 Architecture Reference 59
3.2 Example Architecture Specification 122

4 VPR 135
4.1 Basic flow 135
4.2 Command-line Options 137
4.3 Graphics 165
4.4 Timing Constraints 170
4.5 SDC Commands 171
4.6 File Formats 181
4.7 Debugging Aids 203
4.8 Placer and Router Debugger 204

5 Odin II 207
5.1 Quickstart 207
5.2 User guide 208
12.4 v8.0.0-rc1 - 2019-06-13 ............................................. 384

13 Contact .................................................. 387
  13.1 Mailing Lists .............................................. 387
  13.2 Issue Tracker .............................................. 387

14 Glossary ................................................. 389

15 Publications & References ................................ 391

16 VPR API ................................................ 393
  16.1 Contexts ................................................ 393
  16.2 Netlists ................................................ 397
  16.3 Routing Resource Graph ............................ 416

17 VTRUTIL API ........................................... 427
  17.1 IDs - Ranges ............................................ 427
  17.2 Containers ............................................. 433
  17.3 Container Utils ........................................ 465
  17.4 Logging - Errors - Assertions ..................... 468
  17.5 Geometry .............................................. 472
  17.6 Other .................................................. 475

18 Indices and tables .................................... 495

Bibliography ............................................. 497

Index ..................................................... 501
For more information on the Verilog-to-Routing (VTR) project see VTR and VTR CAD Flow.
For documentation and tutorials on the FPGA architecture description language see: FPGA Architecture Description.
For more specific documentation about VPR see VPR.
CHAPTER ONE

VTR QUICK START

This is a quick introduction to VTR which covers how to run VTR and some of its associated tools (VPR, Odin II, ABC).

1.1 Setting Up VTR

1.1.1 Download VTR

The first step is to download VTR and extract VTR on your local machine.

**Note:** Developers planning to modify VTR should clone the VTR git repository.

1.1.2 Environment Setup

VTR requires several system packages and Python packages to build and run the flow. You can install the required system packages using the following command (this works on Ubuntu 18.04 and 20.04, but you may require different packages on other Linux distributions):

```
> ./install_apt_packages.sh
```

Then, to install the required Python packages (optionally within a new Python virtual environment):

```
> make env # optional: install python virtual environment
> source .venv/bin/activate # optional: activate python virtual environment
> pip install -r requirements.txt # install python packages (in virtual environment if prior commands run, system wide otherwise)
```

1.1.3 Build VTR

On most unix-like systems you can run:

```
> make
```

from the VTR root directory (hereafter referred to as $VTR_ROOT) to build VTR.
1.2 Running VPR

Lets now try taking a simple pre-synthesized circuit (consisting of LUTs and Flip-Flops) and use the VPR tool to implement it on a specific FPGA architecture.

1.2.1 Running VPR on a Pre-Synthesized Circuit

First, let’s make a directory in our home directory where we can work:

```
# Move to our home directory
> cd ~

# Make a working directory
> mkdir -p vtr_work/quickstart/vpr_tseng

# Move into the working directory
> cd ~/vtr_work/quickstart/vpr_tseng
```

Now, let’s invoke the VPR tool to implement:

- the tseng circuit ($VTR_ROOT/vtr_flow/benchmarks/blif/tseng.blif), on
- the EArch FPGA architecture ($VTR_ROOT/vtr_flow/arch/timing/EArc.xml).

We do this by passing these files to the VPR tool, and also specifying that we want to route the circuit on a version of EArch with a routing architecture channel width of 100 (--route_chan_width 100):

```
> $VTR_ROOT/vpr/vpr \
   $VTR_ROOT/vtr_flow/arch/timing/EArc.xml \ 
   $VTR_ROOT/vtr_flow/benchmarks/blif/tseng.blif \
   --route_chan_width 100
```

This will produce a large amount of output as VPR implements the circuit, but you should see something similar to:

```
VPR FPGA Placement and Routing.
Version: 8.1.0-dev-2b5807ecf
Revision: v8.0.0-1821-g2b5807ecf
```

(continues on next page)
which shows that VPR as successful (VPR succeeded), along with how long VPR took to run (~3 seconds in this case).

You will also see various result files generated by VPR which define the circuit implementation:

```bash
> ls *.net *.place *.route

tseng.net  tseng.place  tseng.route
```

along with a VPR log file which contains what VPR printed when last invoked:

```bash
> ls *.log

vpr_stdout.log
```

and various report files describing the characteristics of the implementation:

```bash
> ls *.rpt

packing_pin_util.rpt  report_timing.hold.rpt  report_unconstrained_timing.
                   hold.rpt
                    setup.rpt
```

### 1.2.2 Visualizing Circuit Implementation

**Note:** This section requires that VPR was compiled with graphic support. See *VPR Graphics* for details.

The `.net`, `.place` and `.route` files (along with the input `.blif` and architecture `.xml` files) fully defined the circuit implementation. We can visualize the circuit implementation by:

- Re-running VPR’s analysis stage (`--analysis`), and
- enabling VPR’s graphical user interface (`--disp on`).

### 1.2. Running VPR
This is done by running the following:

```
> $VTR_ROOT/vpr/vpr
  $VTR_ROOT/vtr_flow/arch/timing/EArch.xml
  $VTR_ROOT/vtr_flow/benchmarks/blif/tseng.blif
  --route_chan_width 100
  --analysis --disp on
```

which should open the VPR graphics and allow you to explore the circuit implementation.

As an exercise try the following:

- View the connectivity of a block (connections which drive it, and those which it drives)
- View the internals of a logic block (e.g. try to find the LUTs/.names and Flip-Flops/.latch)
- Visualize all the routed circuit connections

**See also:**

For more details on the various graphics options, see [VPR Graphics](#).

**Note:** If you do not provide `--analysis`, VPR will re-implement the circuit from scratch. If you also specify `--disp on`, you can see how VPR modifies the implementation as it runs. By default `--disp on` stops at key stages to allow you to view and explore the implementation. You will need to press the **Proceed** button in the GUI to allow VPR to continue to the next stage.

### 1.3 Running the VTR Flow

In the previous section we have implemented a pre-synthesized circuit onto a pre-existing FPGA architecture using VPR, and visualized the result. We now turn to how we can implement our own circuit on a pre-existing FPGA architecture.

To do this we begin by describing a circuit behaviourly using the Verilog Hardware Description Language (HDL). This allows us to quickly and consisely define the circuit’s behaviour. We will then use the VTR Flow to synthesize the behavioural Verilog description it into a circuit netlist, and implement it onto an FPGA.

#### 1.3.1 Example Circuit

We will use the following simple example circuit, which causes it’s output to toggle on and off:

```
Listing 1.1: blink.v ($VTR_ROOT/doc/src/quickstart/blink.v)

1 //A simple circuit which blinks an LED on and off periodically
2 module blink(
3   input clk,       //Input clock
4   input i_reset,   //Input active-high reset
5   output o_led);  //Output to LED
6
7   //Sequential logic
8   //
9   //A reset-able counter which increments each clock cycle
10  reg[4:0] r_counter;
```

(continues on next page)
Fig. 1.1: Routed net connections of tseng on EArch.
Fig. 1.2: Input (blue)/output (red) nets of block n_n3199 (highlighted green).
always @(posedge clk) begin
  if (i_reset) begin /*When reset is high, clear counter*/
    r_counter <= 5'd0;
  end else begin /*Otherwise increment counter each clock (note that it will overflow back to zero)*/
    r_counter <= r_counter + 1'b1;
  end
end

endmodule

This Verilog creates a sequential 5-bit register (r_counter) which increments every clock cycle. If the count is below 16 it drives the output (o_led) high, otherwise it drives it low.

1.3.2 Manually Running the VTR Flow

Let's start by making a fresh directory for us to work in:

```bash
> mkdir -p ~/vtr_work/quickstart/blink_manual
> cd ~/vtr_work/quickstart/blink_manual
```

Next we need to run the three main sets of tools:

- **Odin II** performs ‘synthesis’ which converts our behavioural Verilog (.v file) into a circuit netlist (.blif file) consisting of logic equations and FPGA architecture primitives (Flip-Flops, adders etc.),
- **ABC** performs ‘logic optimization’ which simplifies the circuit logic, and ‘technology mapping’ which converts logic equations into the Look-Up-Tables (LUTs) available on an FPGA, and
- **VPR** which performs packing, placement and routing of the circuit to implement it on the targetted FPGA architecture.

**Synthesizing with Odin II**

First we’ll run Odin II on our Verilog file to synthesize it into a circuit netlist, providing the options:

- `-a $VTR_ROOT/vtr_flow/arch/timing/Arch.xml` which specifies what FPGA architecture we are targeting,
- `-V $VTR_ROOT/doc/src/quickstart/blink.v` which specifies the verilog file we want to synthesize, and
- `-o blink.odin.blif` which specifies the name of the generated .blif circuit netlist.

The resulting command is:
which when run should end with something like:

<table>
<thead>
<tr>
<th>Total time: 14.7ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Odin ran with exit status: 0</td>
</tr>
<tr>
<td>Odin II took 0.01 seconds (max_rss 5.1 MiB)</td>
</tr>
</tbody>
</table>

where Odin ran with exit status: 0 indicates Odin successfully synthesized our verilog.

We can now take a look at the circuit which ODIN produced (blink.odin.blif). The file is long and likely harder to follow than our code in blink.v; however it implements the same functionality. Some interesting highlights are shown below:

Listing 1.2: Instantiations of rising-edge triggered Latches (i.e. Flip-Flops) in blink.odin.blif (implements part of r_counter in blink.v)

```
latch blink^MUX~0^MUX_2~23 blink^r_counter~0_FF re blink^clk 3
latch blink^MUX~0^MUX_2~27 blink^r_counter~4_FF re blink^clk 3
```

Listing 1.3: Adder primitive instantiations in blink.odin.blif, used to perform addition (implements part of the + operator in blink.v)

```
.subckt adder a[0]=blink^r_counter~0_FF b[0]=vcc cin[0]=blink^ADD~2-0[0] cout[0]=blink^ADD~2-1[0] sumout[0]=blink^ADD~2-1[1]
.subckt adder a[0]=blink^r_counter~1_FF b[0]=gnd cin[0]=blink^ADD~2-1[0] cout[0]=blink^ADD~2-2[0] sumout[0]=blink^ADD~2-2[1]
```
Listing 1.4: Logic equation (.names truth-table) in blink.odin.blif, implementing logical OR (implements part of the < operator in blink.v)

```
.names blink^LT~4^GT~10 blink^LT~4^GT~12 blink^LT~4^GT~14 blink^LT~4^GT~16 blink^LT~4^GT~18 blink^LT~4^GT~10R~9
1----- 1
-1--- 1
--1-- 1
---1- 1
----1 1
```

See also:

For more information on the BLIF file format see BLIF Netlist (.blif).

### Optimizing and Technology Mapping with ABC

Next, we’ll optimize and technology map our circuit using ABC, providing the option:

- `-c <script>`, where `<script>` is a set of commands telling ABC how to synthesize our circuit.

We’ll use the following, simple ABC commands:

```
read blink.odin.blif;
if -K 6;
write_hie blink.odin.blif blink.abc_no_clock.blif
```

**Note:** Usually you should use a more complicated script (such as that used by `run_vtr_flow`) to ensure ABC optimizes your circuit well.

The corresponding command to run is:

```
> $VTR_ROOT/abc/abc \
    -c 'read blink.odin.blif; if -K 6; write_hie blink.odin.blif blink.abc_no_clock.blif'
```

When run, ABC’s output should look similar to:

```
ABC command line: "read blink.odin.blif; if -K 6; write_hie blink.odin.blif blink.abc_no_clock.blif".
Hierarchy reader converted 6 instances of blackboxes.
The network was strashed and balanced before FPGA mapping.
Hierarchy writer reintroduced 6 instances of blackboxes.
```

If we now inspect the produced BLIF file (blink.abc_no_clock.blif) we see that ABC was able to significantly simplify and optimize the circuit’s logic (compared to blink.odin.blif):

```
# Benchmark "blink" written by ABC on Tue May 19 15:42:50 2020
.model blink
```

(continues on next page)
.inputs blink^clk blink^i_reset
.outputs blink^o_led

.latch n19 blink^r_counter~0_FF 2
.latch n24 blink^r_counter~4_FF 2
.latch n29 blink^r_counter~3_FF 2
.latch n34 blink^r_counter~2_FF 2
.latch n39 blink^r_counter~1_FF 2

.subckt adder a[0]=blink^r_counter~0_FF b[0]=vcc cin[0]=blink^ADD~2-0[0] cout[0]=blink^ADD~2-0[0] sumout[0]=blink^ADD~2-1[1]
.subckt adder a[0]=blink^r_counter~1_FF b[0]=gnd cin[0]=blink^ADD~2-1[0] cout[0]=blink^ADD~2-2[1] sumout[0]=blink^ADD~2-2[1]
.subckt adder a[0]=blink^r_counter~2_FF b[0]=gnd cin[0]=blink^ADD~2-2[0] cout[0]=blink^ADD~2-3[1] sumout[0]=blink^ADD~2-3[1]
.subckt adder a[0]=blink^r_counter~3_FF b[0]=gnd cin[0]=blink^ADD~2-3[0] cout[0]=blink^ADD~2-4[1] sumout[0]=blink^ADD~2-4[1]
.subckt adder a[0]=blink^r_counter~4_FF b[0]=gnd cin[0]=blink^ADD~2-4[0] cout[0]=blink^ADD~2-5[1] sumout[0]=blink^ADD~2-5[1]
.subckt adder a[0]=gnd b[0]=gnd cin[0]=unconn cout[0]=blink^ADD~2-0[0] sumout[0]=blink^ADD~2-0~dummy_output~0~1

.names blink^i_reset blink^ADD~2-1[1] n19
.names blink^i_reset blink^ADD~2-5[1] n24
.names blink^i_reset blink^ADD~2-4[1] n29
.names blink^i_reset blink^ADD~2-3[1] n34
.names blink^i_reset blink^ADD~2-2[1] n39
.names vcc
.names gnd
.names unconn
.names blink^r_counter~4_FF blink^o_led

.end

.model adder
.inputs a[0] b[0] cin[0]
.outputs cout[0] sumout[0]
.blackbox
.end

ABC has kept the .latch and .subckt adder primitives, but has significantly simplified the other logic (.names).
However, there is an issue with the above BLIF produced by ABC: the latches (rising edge Flip-Flops) do not have any clocks or edge sensitivity specified, which is information required by VPR.

Re-inserting clocks

We will restore the clock information by running a script which will transfer that information from the original ODIN BLIF file (writing it to the new file `blink.pre-vpr.blif`):

```
$VTR_ROOT/vtr_flow/scripts/restore_multiclock_latch.pl
  blink.odin.blif
  blink.abc_no_clock.blif
  blink.pre-vpr.blif
```

If we inspect `blink.pre-vpr.blif` we now see that the clock (`blink^clk`) has been restored to the Flip-Flops:

```
> grep 'latch' blink.pre-vpr.blif

.latch n19 blink^r_counter~0_FF re blink^clk 3
.latch n24 blink^r_counter~4_FF re blink^clk 3
.latch n29 blink^r_counter~3_FF re blink^clk 3
.latch n34 blink^r_counter~2_FF re blink^clk 3
.latch n39 blink^r_counter~1_FF re blink^clk 3
```

Implementing the circuit with VPR

Now that we have the optimized and technology mapped netlist (`blink.pre-vpr.blif`), we can invoke VPR to implement it onto the EArch FPGA architecture (in the same way we did with the `tseng` design earlier). However, since our BLIF file doesn’t match the design name we explicitly specify:

- `blink` as the circuit name, and
- the input circuit file with `--circuit_file`.

to ensure the resulting `.net`, `.place` and `.route` files will have the correct names.

The resulting command is:

```
$VTR_ROOT/vpr/vpr
  $VTR_ROOT/vtr_flow/arch/timing/EArch.xml
  blink --circuit_file blink.pre-vpr.blif
  --route_chan_width 100
```

and after VPR finishes we should see the resulting implementation files:

```
> ls *.net *.place *.route

blink.net  blink.place  blink.route
```

We can then view the implementation as usual by appending `--analysis` `--disp on` to the command:

```
$VTR_ROOT/vpr/vpr
  $VTR_ROOT/vtr_flow/arch/timing/EArch.xml
  blink --circuit_file blink.pre-vpr.blif
  --route_chan_width 100
  --analysis --disp on
```

1.3. Running the VTR Flow 13
Fig. 1.3: blink.v circuit implementation on the EArch FPGA architecture as viewed in the VPR GUI
1.3.3 Automatically Running the VTR Flow

Running each stage of the flow manually is time consuming (and potentially error prone). For convenience, VTR provides a script (`run_vtr_flow`) which automates this process.

First, make sure you sure you have activated the Python virtual environment created at the beginning of this tutorial:

```
> source $VTR_ROOT/.venv/bin/activate
```

Next, make a new directory to work in named `blink_run_flow`:

```
> mkdir -p ~/vtr_work/quickstart/blink_run_flow
> cd ~/vtr_work/quickstart/blink_run_flow
```

Now lets run the script (`$VTR_ROOT/vtr_flow/scripts/run_vtr_flow.py`) passing in:

- The circuit verilog file (`$VTR_ROOT/doc/src/quickstart/blink.v`)
- The FPGA architecture file (`$VTR_ROOT/vtr_flow/arch/timing/EAarch.xml`)

and also specifying the options:

- `-temp_dir .` to run in the current directory (`.`, on unix-like systems)
- `--route_chan_width 100` a fixed FPGA routing architecture channel width.

The resulting command is:

```
> $VTR_ROOT/vtr_flow/scripts/run_vtr_flow.py $VTR_ROOT/doc/src/quickstart/blink.v $VTR_ROOT/vtr_flow/arch/timing/EAarch.xml -temp_dir . --route_chan_width 100
```

**Note:** Options unrecognized by `run_vtr_flow` (like `--route_chan_width`) are passed on to VPR.

which should produce output similar to:

```
EArch/blink OK (took 0.26 seconds)
```

There are also multiple log files (including for ABC, ODIN and VPR), which by convention the script names with the `.out` suffix:

```
> ls *.out
0_blackboxing_latch.out odin.out report_clocks.abc.out vanilla_restore_clocks.
˓→out
abc0.out report_clk.out restore_latch0.out vpr.out
```

With the main log files of interest including the ODIN log file (`odin.out`), log files produced by ABC (e.g. `abc0.out`), and the VPR log file (`vpr.out`).

**Note:** ABC may be invoked multiple times if a circuit has multiple clock domains, producing multiple log files (`abc0.out, abc1.out,...`).

You will also see there are several BLIF files produced:
With the main files of interest being `blink.odin.blif` (netlist produced by ODIN), `blink.abc.blif` (final netlist produced by ABC after clock restoration), `blink.pre-vpr.blif` netlist used by VPR (usually identical to `blink.abc.blif`).

Like before, we can also see the implementation files generated by VPR:

```bash
$ ls *.net *.place *.route
blink.net  blink.place  blink.route
```

which we can visualize with:

```bash
$ VTR_ROOT/vpr/vpr \
    $VTR_ROOT/vtr_flow/arch/timing/EArch.xml \
    blink --circuit_file blink.pre-vpr.blif \ 
    --route_chan_width 100 \ 
    --analysis --disp on
```

## 1.4 Next Steps

Now that you’ve finished the VTR quickstart, you’re ready to start experimenting and using VTR.

Here are some possible next steps for users wishing to use VTR:

- Try modifying the Verilog file (e.g. `blink.v`) or make your own circuit and try running it through the flow.
- Learn about FPGA architecture modelling ([Tutorials, Reference](#)), and try modifying a copy of `EArch.xml` to see how it changes the implementation of `blink.v`.
- Read more about the [VTR CAD Flow](#), and [Task](#) automation framework.
- Find out more about using other benchmark sets, like how to run the [Titan Benchmark Suite](#).
- Discover how to [generate FASM](#) for bitstream creation.
- [Suggest or make enhancements to VTR’s documentation](#).

Here are some possible next steps for developers wishing to modify and improve VTR:

- Try the next steps listed for users above to learn how VTR is used.
- Work through the [new developer tutorial](#).
- Read through the [developer guide](#).
- Look for [open issues to which you can contribute](#).
- Begin exploring the source code for the main tools in VTR (e.g. VPR in `$VTR_ROOT/vpr/src`).
The Verilog-to-Routing (VTR) project \cite{LAK+14, RLY+12} is a world-wide collaborative effort to provide an open-source framework for conducting FPGA architecture and CAD research and development. The VTR design flow takes as input a Verilog description of a digital circuit, and a description of the target FPGA architecture.

It then performs:

- Elaboration & Synthesis (\textit{Odin II})
- Logic Optimization & Technology Mapping (\textit{ABC})
- Packing, Placement, Routing & Timing Analysis (\textit{VPR})

Generating FPGA speed and area results.

VTR also includes a set of benchmark designs known to work with the design flow.

### 2.1 VTR CAD Flow

In the standard VTR Flow (Fig. 2.1), \textit{Odin II} converts a Verilog Hardware Description Language (HDL) design into a flattened netlist consisting of logic gates, flip-flops, and blackboxes representing heterogeneous blocks (e.g. adders, multipliers, RAM slices) \cite{JKGS10}.

Next, the \textit{ABC} synthesis package is used to perform technology-independent logic optimization, and technology-maps the circuit into LUTs \cite{CCMB07, PHMB07, SG}. The output of ABC is a \textit{.blif} format netlist of LUTs, flip flops, and blackboxes.

\textit{VPR} then packs this netlist into more coarse-grained logic blocks, places and then routes the circuit \cite{BR97a, Bet98, BR96a, BR96b, BR97b, BR00, BRM99, MBR99, MBR00}. Generating \textit{output files} for each stage. VPR will analyze the resulting implementation, producing various statistics such as the minimum number of tracks per channel required to successfully route, the total wirelength, circuit speed, area and power. VPR can also produce a post-implementation netlist for simulation and formal verification.
Fig. 2.1: VTR CAD flow (and variants)
2.1.1 CAD Flow Variations

Titan CAD Flow

The Titan CAD Flow [MWL+13, MWL+15] interfaces Intel’s Quartus tool with VPR. This allows designs requiring industrial strength language coverage and IP to be brought into VPR.

Other CAD Flow Variants

Many other CAD flow variations are possible.

For instance, it is possible to use other logic synthesis tools like Yosys [Wol] to generate the design netlist. One could also use logic optimizers and technology mappers other than ABC; just put the output netlist from your technology-mapper into .blif format and pass it into VPR.

It is also possible to use tools other than VPR to perform the different stages of the implementation.

For example, if the logic block you are interested in is not supported by VPR, your CAD flow can bypass VPR’s packer by outputting a netlist of logic blocks in .net format. VPR can place and route netlists of any type of logic block – you simply have to create the netlist and describe the logic block in the FPGA architecture description file.

Similarly, if you want only to route a placement produced by another CAD tool you can create a .place file, and have VPR route this pre-existing placement.

If you only need to analyze an implementation produced by another tool, you can create a .route file, and have VPR analyze the implementation, to produce area/delay/power results.

Finally, if your routing architecture is not supported by VPR’s architecture generator, you can describe your routing architecture in an rr_graph.xml file, which can be loaded directly into VPR.

2.1.2 Bitstream Generation

The technology mapped netlist and packing/placement/routing results produced by VPR contain the information needed to generate a device programming bitstreams.

VTR focuses on the core physical design optimization tools and evaluation capabilities for new architectures and does not directly support generating device programming bitstreams. Bitstream generators can either ingest the implementation files directly or make use of VTR utilities to emit FASM.

2.2 Get VTR

2.2.1 How to Cite

Citations are important in academia, as they ensure contributors receive credit for their efforts. Therefore please use the following paper as a general citation whenever you use VTR:


Bibtex:
We are always interested in how VTR is being used, so feel free email the vtr-users list with how you are using VTR.

### 2.2.2 Download

The official VTR release is available from:

https://verilogtorouting.org/download

### 2.2.3 Release

The VTR 8.1 release provides the following:

- benchmark circuits,
- sample FPGA architecture description files,
- the full CAD flow, and
- scripts to run that flow.

The FPGA CAD flow takes as input, a user circuit (coded in Verilog) and a description of the FPGA architecture. The CAD flow then maps the circuit to the FPGA architecture to produce, as output, a placed-and-routed FPGA. Here are some highlights of the 8.1 full release:

- Timing-driven logic synthesis, packing, placement, and routing with multi-clock support.
- Power Analysis
- Benchmark digital circuits consisting of real applications that contain both memories and multipliers.
  - Seven of the 19 circuits contain more than 10,000 6-LUTs. The largest of which is just under 100,000 6-LUTs.
- Sample architecture files of a wide range of different FPGA architectures including:
  1. Timing annotated architectures
  2. Various fracturable LUTs (dual-output LUTs that can function as one large LUT or two smaller LUTs with some shared inputs)
  3. Various configurable embedded memories and multiplier hard blocks
  4. One architecture containing embedded floating-point cores, and
  5. One architecture with carry chains.
- A front-end Verilog elaborator that has support for hard blocks.

This tool can automatically recognize when a memory or multiplier instantiated in a user circuit is too large for a target FPGA architecture. When this happens, the tool can automatically split that memory/multiplier into multiple smaller components (with some glue logic to tie the components together). This makes it easier
to investigate different hard block architectures because one does not need to modify the Verilog if the circuit instantiates a memory/multiplier that is too large.

- Packing/Clustering support for FPGA logic blocks with widely varying functionality.
  This includes memories with configurable aspect ratios, multipliers blocks that can fracture into smaller multipliers, soft logic clusters that contain fracturable LUTs, custom interconnect within a logic block, and more.
- Ready-to-run scripts that guide a user through the complexities of building the tools as well as using the tools to map realistic circuits (written in Verilog) to FPGA architectures.
- Regression tests of experiments that we have conducted to help users error check and/or compare their work.

Along with experiments for more conventional FPGAs, we also include an experiment that explores FPGAs with embedded floating-point cores investigated in [HYL+09] to illustrate the usage of the VTR framework to explore unconventional FPGA architectures.

### 2.2.4 Development Repository

The development repository for the Verilog-to-Routing project is hosted at:

https://github.com/verilog-to-routing/vtr-verilog-to-routing

Unlike the nicely packaged official releases the code is in a constant state of flux. You should expect that the tools are not always stable and that more work is needed to get the flow to run.

### 2.3 Building VTR

#### 2.3.1 Setting up Your Environment

VTR requires several system packages. From the top-level directory, run the following script to install the required packages on a modern Debian or Ubuntu system:

```
./install_apt_packages.sh
```

You will also need several Python packages. You can optionally install and activate a Python virtual environment so that you do not need to modify your system Python installation:

```
make env
source .venv/bin/activate
```

Then to install the Python packages:

```
pip install -r requirements.txt
```

**Note:** If you chose to install the Python virtual environment, you will need to remember to activate it on any new terminal window you use, before you can run the VTR flow or regressions tests (source .venv/bin/activate).
2.3.2 Building

From the top-level, run:

```
make
```

which will build all the required tools.

The complete VTR flow has been tested on 64-bit Linux systems. The flow should work in other platforms (32-bit Linux, Windows with cygwin) but this is untested.

Full information about building VTR, including setting up required system packages and Python packages, can be found in Optional Build Information page.

Please let us know your experience with building VTR so that we can improve the experience for others.

The tools included official VTR releases have been tested for compatibility. If you download a different version of those tools, then those versions may not be mutually compatible with the VTR release.

2.3.3 Verifying Installation

To verify that VTR has been installed correctly run::

```
./vtr_flow/scripts/run_vtr_task.py regression_tests/vtr_reg_basic/basic_timing
```

The expected output is::

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>k6_N10_mem32K_40nm/single_ff</td>
<td>OK</td>
</tr>
<tr>
<td>k6_N10_mem32K_40nm/single_ff</td>
<td>OK</td>
</tr>
<tr>
<td>k6_N10_mem32K_40nm/single_wire</td>
<td>OK</td>
</tr>
<tr>
<td>k6_N10_mem32K_40nm/single_wire</td>
<td>OK</td>
</tr>
<tr>
<td>k6_N10_mem32K_40nm/diffeq1</td>
<td>OK</td>
</tr>
<tr>
<td>k6_N10_mem32K_40nm/diffeq1</td>
<td>OK</td>
</tr>
<tr>
<td>k6_N10_mem32K_40nm/ch_intrinsics</td>
<td>OK</td>
</tr>
<tr>
<td>k6_N10_mem32K_40nm/ch_intrinsics</td>
<td>OK</td>
</tr>
</tbody>
</table>

2.4 Optional Build Information

This page contains additional information about the VTR build system, and how to build VTR on other OS platforms or with non-standard build options. If you only need to the default features of VTR on a Debian/Ubuntu system, the previous Building VTR page should be sufficient and you can skip this page.

2.4.1 Dependencies

Most package and Python dependencies can be installed using the instructions on the previous Building VTR page. However, more detailed information is provided here.
CMake

VTR uses CMake as its build system. CMake provides a portable cross-platform build systems with many useful features. For unix-like systems we provide a wrapper Makefile which supports the traditional `make` and `make clean` commands, but calls CMake behind the scenes.

Tested Compilers

VTR requires a C++14 compliant compiler. It is tested against the default compilers of all Debian and Ubuntu releases within their standard support lifetime. Currently, those are the following:

- GCC/G++: 7, 8, 9, 10, 11
- Clang/Clang++: 6, 7, 10

Other compilers may work but are untested (your mileage may vary).

Package Dependencies

- At minimum you will require:
  - A modern C++ compiler supporting C++14 (such as GCC >= 4.9 or clang >= 3.6)
  - cmake, make
  - bison, flex, pkg-config
- Additional packages are required for the VPR GUI (Cairo, FreeType, libXft, libX11, fontconfig, libgtk-3-dev)
- The scripts to run the entire VTR flow, as well as the regressions scripts, require Python3 and Python packages listed in the `requirements.txt` file.
- Developers may also wish to install other packages (git, ctags, gdb, valgrind, clang-format-7)
- To generate the documentation you will need Sphinx, Doxygen, and several Python packages. The Python packages can be installed with the following command:

  ```bash
  pip install -r doc/requirements.txt
  ```

2.4.2 Build Options

Build Type

You can specify the build type by passing the `BUILD_TYPE` parameter. For instance to create a debug build (no optimization and debug symbols):

```bash
# In the VTR root
$ make BUILD_TYPE=debug
...
[100%] Built target vpr
```
Passing parameters to CMake

You can also pass parameters to CMake. For instance to set the CMake configuration variable VTR_ENABLE_SANITIZE on:

```
#In the VTR root
$ make CMAKE_PARAMS="-DVTR_ENABLE_SANITIZE=ON"
...
[100%] Built target vpr
```

Both the BUILD_TYPE and CMAKE_PARAMS can be specified concurrently:

```
#In the VTR root
$ make BUILD_TYPE=debug CMAKE_PARAMS="-DVTR_ENABLE_SANITIZE=ON"
...
[100%] Built target vpr
```

Using CMake directly

You can also use cmake directly. First create a build directory under the VTR root:

```
#In the VTR root
$ mkdir build
$ cd build

#Call cmake pointing to the directory containing the root CMakeLists.txt
$ cmake ..

#Build
$ make
```

Changing configuration on the command line

You can change the CMake configuration by passing command line parameters. For instance to set the configuration to debug:

```
#In the build directory
$ cmake . -DCMAKE_BUILD_TYPE=debug

#Re-build
$ make
```
Changing configuration interactively with ccmake

You can also use `ccmake` to modify the build configuration.

```
# From the build directory
$ ccmake . # Make some configuration change

# Build
$ make
```

2.4.3 Other platforms

CMake supports a variety of operating systems and can generate project files for a variety of build systems and IDEs. While VTR is developed primarily on Linux, it should be possible to build on different platforms (your mileage may vary). See the CMake documentation for more details about using cmake and generating project files on other platforms and build systems (e.g. Eclipse, Microsoft Visual Studio).

**Nix**

Nix can be used to build VTR on other platforms, such as MacOS.

If you don’t have Nix, you can get it with:

```
$ curl -L https://nixos.org/nix/install | sh
```

These commands will set up dependencies for Linux and MacOS and build VTR:

```
# In the VTR root
$ nix-shell dev/nix/shell.nix
$ make
```

**Microsoft Windows**

*NOTE: VTR support on Microsoft Windows is considered experimental*

**WSL**

The Windows Subsystem for Linux (WSL), “lets developers run a GNU/Linux environment – including most command-line tools, utilities, and applications – directly on Windows, unmodified, without the overhead of a traditional virtual machine or dual-boot setup.”

This is the recommended way to run VTR on Windows systems.
Cygwin

Cygwin provides a POSIX (i.e. unix-like) environment for Microsoft Windows.
From within the cygwin terminal follow the Unix-like build instructions listed above.
Note that the generated executables will rely upon Cygwin (e.g. cygwin1.dll) for POSIX compatibility.

Cross-compiling from Linux to Microsoft Windows with MinGW-W64

It is possible to cross-compile from a Linux host system to generate Microsoft Windows executables using the MinGW-W64 compilers. These can usually be installed with your Linux distribution’s package manager (e.g. sudo apt-get install mingw-w64 on Debian/Ubuntu).

Unlike Cygwin, MinGW executables will depend upon the standard Microsoft Visual C++ run-time.
To build VTR using MinGW:

```
#In the VTR root
$ mkdir build_win64
$ cd build_win64

#Run cmake specifying the toolchain file to setup the cross-compilation environment
$ cmake .. -DCMAKE_TOOLCHAIN_FILE ../cmake/toolchains/mingw-linux-cross-compile-to-windows.cmake

#Building will produce Windows executables
$ make
```

Note that by default the MS Windows target system will need to dynamically link to the libc++ and libstdc++ DLLs. These are usually found under /usr/lib/gcc on the Linux host machine.
See the toolchain file for more details.

Microsoft Visual Studio

CMake can generate a Microsoft Visual Studio project, enabling VTR to be built with the Microsoft Visual C++ (MSVC) compiler.

Installing additional tools

VTR depends on some external unix-style tools during it's build process; in particular the flex and bison parser generators.

One approach is to install these tools using MSYS2, which provides up-to-date versions of many unix tools for MS Windows.

To ensure CMake can find the flex and bison executables you must ensure that they are available on your system path. For instance, if MSYS2 was installed to C:\msys64 you would need to ensure that C:\msys64\usr\bin was included in the system PATH environment variable.
Generating the Visual Studio Project

CMake (e.g. the cmake-gui) can then be configured to generate the MSVC project.

2.5 Running the VTR Flow

VTR is a collection of tools that perform the full FPGA CAD flow from Verilog to routing.

The design flow consists of:

- **Odin II** (Logic Synthesis)
- **ABC** (Logic Optimization & Technology Mapping)
- **VPR** (Pack, Place & Route)

There is no single executable for the entire flow.

Instead, scripts are provided to allow the user to easily run the entire tool flow. The following provides instructions on using these scripts to run VTR.

2.5.1 Running a Single Benchmark

The `run_vtr_flow` script is provided to execute the VTR flow for a single benchmark and architecture.

**Note:** In the following `$VTR_ROOT` means the root directory of the VTR source code tree.

```
$VTR_ROOT/vtr_flow/scripts/run_vtr_flow.py <circuit_file> <architecture_file>
```

It requires two arguments:

- `<circuit_file>` A benchmark circuit, and
- `<architecture_file>` an FPGA architecture file

Circuits can be found under:

```
$VTR_ROOT/vtr_flow/benchmarks/
```

Architecture files can be found under:

```
$VTR_ROOT/vtr_flow/arch/
```

The script can also be used to run parts of the VTR flow.

**See also:**

`run_vtr_flow` for the detailed command line options of `run_vtr_flow.py`. 
2.5.2 Running Multiple Benchmarks & Architectures with Tasks

VTR also supports tasks, which manage the execution of the VTR flow for multiple benchmarks and architectures. By default, tasks execute the run_vtr_flow for every circuit/architecture combination.

VTR provides a variety of standard tasks which can be found under:

```bash
$VTR_ROOT/vtr_flow/tasks
```

Tasks can be executed using run_vtr_task:

```bash
$VTR_ROOT/vtr_flow/scripts/run_vtr_task.py <task_name>
```

See also:

*run_vtr_task* for the detailed command line options of run_vtr_task.py.

See also:

*Tasks* for more information on creating, modifying and running tasks.

2.5.3 Extracting Information & Statistics

VTR can also extract useful information and statistics from executions of the flow such as area, speed tool execution time etc.

For single benchmarks parse_vtr_flow extrastics statistics from a single execution of the flow.

For a Task, parse_vtr_task can be used to parse and assemble statistics for the entire task (i.e. multiple circuits and architectures).

For regression testing purposes these results can also be verified against a set of golden reference results. See parse_vtr_task for details.

2.6 Benchmarks

There are several sets of benchmark designs which can be used with VTR.

2.6.1 VTR Benchmarks

The VTR benchmarks [LAK+14, RLY+12] are a set of medium-sized benchmarks included with VTR. They are fully compatible with the full VTR flow. They are suitable for FPGA architecture research and medium-scale CAD research.
Table 2.1: The VTR 7.0 Benchmarks.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>bgm</td>
<td>Finance</td>
</tr>
<tr>
<td>blob_merge</td>
<td>Image Processing</td>
</tr>
<tr>
<td>boundtop</td>
<td>Ray Tracing</td>
</tr>
<tr>
<td>ch_intrinsics</td>
<td>Memory Init</td>
</tr>
<tr>
<td>diffeq1</td>
<td>Math</td>
</tr>
<tr>
<td>diffeq2</td>
<td>Math</td>
</tr>
<tr>
<td>LU8PEEng</td>
<td>Math</td>
</tr>
<tr>
<td>LU32PEEng</td>
<td>Math</td>
</tr>
<tr>
<td>mcml</td>
<td>Medical Physics</td>
</tr>
<tr>
<td>mkDelayWorker32B</td>
<td>Packet Processing</td>
</tr>
<tr>
<td>mkPktMerge</td>
<td>Packet Processing</td>
</tr>
<tr>
<td>mkSMAdapter4B</td>
<td>Packet Processing</td>
</tr>
<tr>
<td>or1200</td>
<td>Soft Processor</td>
</tr>
<tr>
<td>raygentop</td>
<td>Ray Tracing</td>
</tr>
<tr>
<td>sha</td>
<td>Cryptography</td>
</tr>
<tr>
<td>stereovision0</td>
<td>Computer Vision</td>
</tr>
<tr>
<td>stereovision1</td>
<td>Computer Vision</td>
</tr>
<tr>
<td>stereovision2</td>
<td>Computer Vision</td>
</tr>
<tr>
<td>stereovision3</td>
<td>Computer Vision</td>
</tr>
</tbody>
</table>

The VTR benchmarks are provided as Verilog under:

`$VTR_ROOT/vtr_flow/benchmarks/verilog`

This provides full flexibility to modify and change how the designs are implemented (including the creation of new netlist primitives).

The VTR benchmarks are also included as pre-synthesized BLIF files under:

`$VTR_ROOT/vtr_flow/benchmarks/vtr_benchmarks_blif`

### 2.6.2 Titan Benchmarks

The Titan benchmarks [MWL+13, MWL+15] are a set of large modern FPGA benchmarks. The pre-synthesized versions of these benchmarks are compatible with recent versions of VPR.

The Titan benchmarks are suitable for large-scale FPGA CAD research, and FPGA architecture research which does not require synthesizing new netlist primitives.

**Note:** The Titan benchmarks are not included with the VTR release (due to their size). However they can be downloaded and extracted by running `make get_titan_benchmarks` from the root of the VTR tree. They can also be downloaded manually.

See also:

*Running the Titan Benchmarks*
### 2.6.3 Koios Benchmarks

The Koios benchmarks [ABR+21] are a set of Deep Learning (DL) benchmarks. They are suitable for DL related architecture and CAD research. There are 19 designs that include several medium-sized benchmarks and some large benchmarks. The designs target different network types (CNNs, RNNs, MLPs, RL) and layer types (fully-connected, convolution, activation, softmax, reduction, eltwise). Some of the designs are generated from HLS tools as well. These designs use many precisions including binary, different fixed point types int8/16/32, brain floating point (bfloat16), and IEEE half-precision floating point (fp16).

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clstm_like</td>
<td>CLSTM-like accelerator</td>
</tr>
<tr>
<td>dla_like</td>
<td>Intel-DLA-like accelerator</td>
</tr>
<tr>
<td>lstm</td>
<td>LSTM engine</td>
</tr>
<tr>
<td>tpu_like</td>
<td>Google-TPU-v1-like accelerator</td>
</tr>
<tr>
<td>bnn</td>
<td>4-layer binary neural network</td>
</tr>
<tr>
<td>tiny_darknet_like</td>
<td>Accelerator for Tiny Darknet</td>
</tr>
<tr>
<td>gemm_layer</td>
<td>20x20 matrix multiplication engine</td>
</tr>
<tr>
<td>attention_layer</td>
<td>Transformer self-attention layer</td>
</tr>
<tr>
<td>conv_layer</td>
<td>GEMM based convolution</td>
</tr>
<tr>
<td>spmv</td>
<td>Sparse matrix vector multiplication</td>
</tr>
<tr>
<td>robot_rl</td>
<td>Robot+maze application</td>
</tr>
<tr>
<td>reduction_layer</td>
<td>Add/max/min reduction tree</td>
</tr>
<tr>
<td>softmax</td>
<td>Softmax classification layer</td>
</tr>
<tr>
<td>conv_layer_hls</td>
<td>Sliding window convolution</td>
</tr>
<tr>
<td>eltwise_layer</td>
<td>Matrix elementwise add/sub/mult</td>
</tr>
</tbody>
</table>

The VTR benchmarks are provided as Verilog (enabling full flexibility to modify and change how the designs are implemented) under:

```
$VTR_ROOT/vtr_flow/benchmarks/verilog/koios
```

To use these benchmarks, please see the documentation in the README file at: https://github.com/verilog-to-routing/vtr-verilog-to-routing/tree/master/vtr_flow/benchmarks/verilog/koios

### 2.6.4 MCNC20 Benchmarks

The MCNC benchmarks [Yan91] are a set of small and old (circa 1991) benchmarks. They consist primarily of logic (i.e. LUTs) with few registers and no hard blocks.

**Warning:** The MCNC20 benchmarks are not recommended for modern FPGA CAD and architecture research. Their small size and design style (e.g. few registers, no hard blocks) make them unrepresentative of modern FPGA usage. This can lead to misleading CAD and/or architecture conclusions.

The MCNC20 benchmarks included with VTR are available as .blif files under:

```
$VTR_ROOT/vtr_flow/benchmarks/blif/
```

The versions used in the VPR 4.3 release, which were mapped to \( K \)-input look-up tables using FlowMap [CD94], are available under:
where $K = <#>$.

Table 2.3: The MCNC20 benchmarks.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Approximate Number of Netlist Primitives</th>
</tr>
</thead>
<tbody>
<tr>
<td>alu4</td>
<td>934</td>
</tr>
<tr>
<td>apex2</td>
<td>1116</td>
</tr>
<tr>
<td>apex4</td>
<td>916</td>
</tr>
<tr>
<td>bigkey</td>
<td>1561</td>
</tr>
<tr>
<td>clma</td>
<td>3754</td>
</tr>
<tr>
<td>des</td>
<td>1199</td>
</tr>
<tr>
<td>difeq</td>
<td>1410</td>
</tr>
<tr>
<td>dsp</td>
<td>1559</td>
</tr>
<tr>
<td>elliptic</td>
<td>3535</td>
</tr>
<tr>
<td>ex1010</td>
<td>2669</td>
</tr>
<tr>
<td>ex5p</td>
<td>824</td>
</tr>
<tr>
<td>frisc</td>
<td>3291</td>
</tr>
<tr>
<td>misex3</td>
<td>842</td>
</tr>
<tr>
<td>pdc</td>
<td>2879</td>
</tr>
<tr>
<td>s298</td>
<td>732</td>
</tr>
<tr>
<td>s38417</td>
<td>4888</td>
</tr>
<tr>
<td>s38584.1</td>
<td>4726</td>
</tr>
<tr>
<td>seq</td>
<td>1041</td>
</tr>
<tr>
<td>spla</td>
<td>2278</td>
</tr>
<tr>
<td>tseng</td>
<td>1583</td>
</tr>
</tbody>
</table>

2.6.5 SymbiFlow Benchmarks

SymbiFlow benchmarks are a set of small and medium sized tests to verify and test the SymbiFlow-generated architectures, including primarily the Xilinx Artix-7 device families.

The tests are generated by nightly builds from the symbiflow-arch-defs repository, and uploaded to a Google Cloud Platform from where they are fetched and executed in the VTR benchmarking suite.

The circuits are the following:

Table 2.4: The SymbiFlow benchmarks.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>picosoc @100 MHz</td>
<td>simple SoC with a picorv32 CPU running @100MHz</td>
</tr>
<tr>
<td>picosoc @50MHz</td>
<td>simple SoC with a picorv32 CPU running @50MHz</td>
</tr>
<tr>
<td>base-litex</td>
<td>LiteX-based SoC with a VexRiscv CPU booting into a BIOS only</td>
</tr>
<tr>
<td>ddr-litex</td>
<td>LiteX-based SoC with a VexRiscv CPU and a DDR controller</td>
</tr>
<tr>
<td>ddr-eth-litex</td>
<td>LiteX-based SoC with a VexRiscv CPU, a DDR controller and an Ethernet core</td>
</tr>
<tr>
<td>linux-litex</td>
<td>LiteX-based SoC with a VexRiscv CPU capable of booting linux</td>
</tr>
</tbody>
</table>

The SymbiFlow benchmarks can be downloaded and extracted by running the following:

cd $VTR_ROOT
make get_symbiflow_benchmarks

2.6. Benchmarks
Once downloaded and extracted, benchmarks are provided as post-synthesized eblif files under:

```
$VTR_ROOT/vtr_flow/benchmarks/symbiflow
```

### 2.7 Power Estimation

VTR provides transistor-level dynamic and static power estimates for a given architecture and circuit.

Fig. 2.2 illustrates how power estimation is performed in the VTR flow. The actual power estimation is performed within the VPR executable; however, additional files must be provided. In addition to the circuit and architecture files, power estimation requires files detailing the signal activities and technology properties.

*Running VTR with Power Estimation* details how to run power estimation for VTR. *Supporting Tools* provides details on the supporting tools that are used to generate the signal activities and technology properties files. *Architecture Modelling* provides details about how the tool models architectures, including different modelling methods and options. *Other Architecture Options & Techniques* provides more advanced configuration options.

#### 2.7.1 Running VTR with Power Estimation

**VTR Flow**

The easiest way to run the VTR flow is to use the `run_vtr_flow` script.

In order to perform power estimation, you must add the following options:

- `run_vtr_flow.py -power`
- `run_vtr_flow.py -cmos_tech <cmos_tech_properties_file>`

The CMOS technology properties file is an XML file that contains relevant process-dependent information needed for power estimation. XML files for 22nm, 45nm, and 130nm PTM models can be found here:

```
$VTR_ROOT/vtrflow/tech/*
```

See *Technology Properties* for information on how to generate an XML file for your own SPICE technology model.

In this mode, the VTR will run ODIN->ABC->ACE->VPR. The ACE stage is additional and specific to this power estimation flow. Using `run_vtr_flow.py` will automatically run ACE 2.0 to generate activity information and a new BLIF file (see *ACE 2.0 Activity Estimation* for details).

The final power estimates will be available in file named `<circuit_name>.power` in the result directory.

Here is an example command:

```
VPR
```

Power estimation can also be run directly from VPR with the following (all required) options:

- `vpr --power`: Enables power estimation.
- `vpr --activity_file <activities.act>`: The activity file, produce by ACE 2.0, or another tool.
- `vpr --tech_properties <tech_properties.xml>`: The technology properties file.

Power estimation requires an activity file, which can be generated as described in *ACE 2.0 Activity Estimation*. 

---

32 Chapter 2. VTR
Fig. 2.2: Power Estimation in the VTR Flow
2.7.2 Supporting Tools

Technology Properties

Power estimation requires information detailing the properties of the CMOS technology. This information, which includes transistor capacitances, leakage currents, etc. is included in an .xml file, and provided as a parameter to VPR. This XML file is generated using a script which automatically runs HSPICE, performs multiple circuit simulations, and extract the necessary values.

Some of these technology XML files are included with the release, and are located here:

```
$VTR_ROOT/vtr_flow/tech/*
```

If the user wishes to use a different CMOS technology file, they must run the following script:

```
Note: HSPICE must be available on the users path
```

```
$VTR_ROOT/vtr_flow/scripts/generate_cmos_tech_data.pl <tech_file> <tech_size> <vdd> <temp>
```

where:

- `<tech_file>`: Is a SPICE technology file, containing a `pmos` and `nmos` models.
- `<tech_size>`: The technology size, in meters.

**Example:**

A 90nm technology would have the value `90e-9`.

- `<vdd>`: Supply voltage in Volts.
- `<temp>`: Operating temperature, in Celsius.

ACE 2.0 Activity Estimation

Power estimation requires activity information for the entire netlist. This activity information consists of two values:

1. **The Signal Probability**, $P_1$, is the long-term probability that a signal is logic-high.

   **Example:**
   
   A clock signal with a 50% duty cycle will have $P_1(\text{clk}) = 0.5$.

2. **The Transition Density** (or switching activity), $A_S$, is the average number of times the signal will switch during each clock cycle.

   **Example:**
   
   A clock has $A_S(\text{clk}) = 2$.

The default tool used to perform activity estimation in VTR is ACE 2.0 [LW06]. This tool was originally designed to work with the (now obsolete) Berkeley SIS tool ACE 2.0 was modified to use ABC, and is included in the VTR package here:

```
$VTR_ROOT/ace2
```

The tool can be run using the following command-line arguments:
$VTR_ROOT/ace2/ace -b <abc.blif> -c <clock_name> -o <activities.act> -n <new.blif>

where

- `<abc.blif>`: Is the input BLIF file produced by ABC.
- `<clock_name>`: Is the name of the clock in the input BLIF file.
- `<activities.act>`: Is the activity file to be created.

This will be functionally identical in function to the ABC blif; however, since ABC does not maintain internal node names, a new BLIF must be produced with node names that match the activity file. This blif file is fed to the subsequent parts of the flow (to VPR). If a user is using run_vtr_flow.py (which will run ACE 2.0 underneath if the options mentioned earlier like -power are used), then the flow will copy this ACE2 generated blif file (<circuit_name>.ace.blif) to <circuit_name>.pre-vpr.blif and then launch VPR with this new file.

User's may with to use their own activity estimation tool. The produced activity file must contain one line for each net in the BLIF file, in the following format:

```
<net name> <signal probability> <transistion density>
```

### 2.7.3 Architecture Modelling

The following section describes the architectural assumptions made by the power model, and the related parameters in the architecture file.

#### Complex Blocks

The VTR architecture description language supports a hierarchichal description of blocks. In the architecture file, each block is described as a `pb_type`, which may includes one or more children of type `pb_type`, and interconnect structures to connect them.

The power estimation algorithm traverses this hierarchy recursively, and performs power estimation for each `pb_type`. The power model supports multiple power estimation methods, and the user specifies the desired method in the architecture file:

```
<pb_type>
  <power method="<estimation-method>">
</pb_type>
```

The following is a list of valid estimation methods. Detailed descriptions of each type are provided in the following sections. The methods are listed in order from most accurate to least accurate.

1. **specify-size**: Detailed transistor level modelling.
   
The user supplies all buffer sizes and wire-lengths. Any not provided by the user are ignored.

2. **auto-size**: Detailed transistor level modelling.
   
The user can supply buffer sizes and wire-lengths; however, they will be automatically inserted when not provided.

   
The user specifies energy per toggle of the pins. Static power provided as an absolute.
4. **C-internal**: Higher-level modelling.
   The user supplies the internal capacitance of the block. Static power provided as an absolute.

5. **absolute**: Highest-level modelling.
   The user supplies both dynamic and static power as absolutes.

Other methods of estimation:
1. **ignore**: The power of the `pb_type` is ignored, including any children.
2. **sum-of-children**: Power of `pb_type` is solely the sum of all children `pb_types`.
   Interconnect between the `pb_type` and its children is ignored.

**Note**: If no estimation method is provided, it is inherited from the parent `pb_type`.

**Note**: If the top-level `pb_type` has no estimation method, `auto-size` is assumed.

**specify-size**

This estimation method provides a detailed transistor level modelling of CLBs, and will provide the most accurate power estimations. For each `pb_type`, power estimation accounts for the following components (see Fig. 2.3).

- Interconnect multiplexers
- Buffers and wire capacitances
- Child `pb_types`

**Multiplexers**: Interconnect multiplexers are modelled as 2-level pass-transistor multiplexers, comprised of minimum-size NMOS transistors. Their size is determined automatically from the `<interconnect/>` structures in the architecture description file.

**Buffers and Wires**: Buffers and wire capacitances are not defined in the architecture file, and must be explicitly added by the user. They are assigned on a per port basis using the following construct:

```xml
<pb_type>
    <input name="my_input" num_pins="1">
        <power ...options.../>
    </input>
</pb_type>
```

The wire and buffer attributes can be set using the following options. If no options are set, it is assumed that the wire capacitance is zero, and there are no buffers present. Keep in mind that the port construct allows for multiple pins per port. These attributes will be applied to each pin in the port. If necessary, the user can separate a port into multiple ports with different wire/buffer properties.

- **wire_capacitance=1.0e-15**: The absolute capacitance of the wire, in Farads.
- **wire_length=1.0e-7**: The absolute length of the wire, in meters.

   The local interconnect capacitance option must be specified, as described in *Local Interconnect Capacitance*.

- **wire_length=auto**: The wirelength is automatically sized. See *Local Wire Auto-Sizing*.
- **buffer_size=2.0**: The size of the buffer at this pin. See for more *Buffer Sizing* information.
Fig. 2.3: Sample Block
• buffer_size=auto: The size of the buffer is automatically sized, assuming it drives the above wire capacitance and a single multiplexer. See Buffer Sizing for more information.

**Primitives:** For all child pb_types, the algorithm performs a recursive call. Eventually pb_types will be reached that have no children. These are primitives, such as flip-flops, LUTs, or other hard-blocks. The power model includes functions to perform transistor-level power estimation for flip-flops and LUTs (Note: the power model doesn't, by default, include power estimation for single-bit adders that are commonly found in logic blocks of modern FPGAs). If the user wishes to use a design with other primitive types (memories, multipliers, etc), they must provide an equivalent function. If the user makes such a function, the power_usage_primitive function should be modified to call it. Alternatively, these blocks can be configured to use higher-level power estimation methods.

**auto-size**

This estimation method also performs detailed transistor-level modelling. It is almost identical to the specify-size method described above. The only difference is that the local wire capacitance and buffers are automatically inserted for all pins, when necessary. This is equivalent to using the specify-size method with the wire_length=auto and buffer_size=auto options for every port.

**Note:** This is the default power estimation method.

Although not as accurate as user-provided buffer and wire sizes, it is capable of automatically capturing trends in power dissipation as architectures are modified.

**pin-toggle**

This method allows users to specify the dynamic power of a block in terms of the energy per toggle (in Joules) of each input, output or clock pin for the pb_type. The static power is provided as an absolute (in Watts). This is done using the following construct:

```xml
<pb_type>
  ...
  <power method="pin-toggle">
    <port name="A" energy_per_toggle="1.0e-12"/>
    <port name="B[3:2]" energy_per_toggle="1.0e-12"/>
    <port name="C" energy_per_toggle="1.0e-12" scaled_by_static_prob="en1"/>
    <port name="D" energy_per_toggle="1.0e-12" scaled_by_static_prob_n="en2"/>
    <static_power power_per_instance="1.0e-6"/>
  </power>
</pb_type>
```

Keep in mind that the port construct allows for multiple pins per port. Unless an subset index is provided, the energy per toggle will be applied to each pin in the port. The energy per toggle can be scaled by another signal using the scaled_by_static_prob. For example, you could scale the energy of a memory block by the read enable pin. If the read enable were high 80% of the time, then the energy would be scaled by the signal_probability, 0.8. Alternatively scaled_by_static_prob_n can be used for active low signals, and the energy will be scaled by \((1 - \text{signal\_probability})\).

This method does not perform any transistor-level estimations; the entire power estimation is performed using the above values. It is assumed that the power usage specified here includes power of all child pb_types. No further recursive power estimation will be performed.
C-internal

This method allows the users to specify the dynamic power of a block in terms of the internal capacitance of the block. The activity will be averaged across all of the input pins, and will be supplied with the internal capacitance to the standard equation:

\[ P_{dyn} = \frac{1}{2} \alpha CV^2. \]

Again, the static power is provided as an absolute (in Watts). This is done using the following construct:

```xml
<pb_type>
  <power method="c-internal">
    <dynamic_power C_internal="1.0e-16"/>
    <static_power power_per_instance="1.0e-16"/>
  </power>
</pb_type>
```

It is assumed that the power usage specified here includes power of all child \texttt{pb_types}. No further recursive power estimation will be performed.

absolute

This method is the most basic power estimation method, and allows users to specify both the dynamic and static power of a block as absolute values (in Watts). This is done using the following construct:

```xml
<pb_type>
  <power method="absolute">
    <dynamic_power power_per_instance="1.0e-16"/>
    <static_power power_per_instance="1.0e-16"/>
  </power>
</pb_type>
```

It is assumed that the power usage specified here includes power of all child \texttt{pb_types}. No further recursive power estimation will be performed.

Global Routing

Global routing consists of switch boxes and input connection boxes.

Switch Boxes

Switch boxes are modelled as the following components (Fig. 2.4):

1. Multiplexer
2. Buffer
3. Wire capacitance

\textbf{Multiplexer}: The multiplexer is modelled as 2-level pass-transistor multiplexer, comprised of minimum-size NMOS transistors. The number of inputs to the multiplexer is automatically determined.

\textbf{Buffer}: The buffer is a multistage CMOS buffer. The buffer size is determined based upon output capacitance provided in the architecture file.
The user may override this method by providing the buffer size as shown below:

```
<switchlist>
  <switch type="mux" ... power_buf_size="16"/>
</switchlist>
```

The size is the drive strength of the buffer, relative to a minimum-sized inverter.

**Input Connection Boxes**

Input connection boxes are modelled as the following components (Fig. 2.5):

- One buffer per routing track, sized to drive the load of all input multiplexers to which the buffer is connected (For buffer sizing see *Buffer Sizing*).
- One multiplexer per block input pin, sized according to the number of routing tracks that connect to the pin.

**Clock Network**

The clock network modelled is a four quadrant spine and rib design, as illustrated in Fig. 2.6. At this time, the power model only supports a single clock. The model assumes that the entire spine and rib clock network will contain buffers separated in distance by the length of a grid tile. The buffer sizes and wire capacitances are specified in the architecture file using the following construct:

```
<clocks>
  <clock ... clock_options ... />
</clocks>
```

The following clock options are supported:

- C_wire=1e-16: The absolute capacitance, in fards, of the wire between each clock buffer.
• **C_wire_per_m=1e-12**: The wire capacitance, in fards per m.

  The capacitance is calculated using an automatically determined wirelength, based on the area of a tile in the FPGA.

• **buffer_size=2.0**: The size of each clock buffer.

  This can be replaced with the auto keyword. See *Buffer Sizing* for more information on buffer sizing.

### 2.7.4 Other Architecture Options & Techniques

#### Local Wire Auto-Sizing

Due to the significant user effort required to provide local buffer and wire sizes, we developed an algorithm to estimate them automatically. This algorithm recursively calculates the area of all entities within a CLB, which consists of the area of primitives and the area of local interconnect multiplexers. If an architecture uses new primitives in CLBs, it should include a function that returns the transistor count. This function should be called from within `power_count_transistors_primitive()`.

In order to determine the wire length that connects a parent entity to its children, the following assumptions are made:

• **Assumption 1**: All components (CLB entities, multiplexers, crossbars) are assumed to be contained in a square-shaped area.

• **Assumption 2**: All wires connecting a parent entity to its child pass through the *interconnect square*, which is the sum area of all interconnect multiplexers belonging to the parent entity.

Fig. 2.7 provides an illustration of a parent entity connected to its child entities, containing one of each interconnect type (direct, many-to-1, and complete). In this figure, the square on the left represents the area used by the transistors of the interconnect multiplexers. It is assumed that all connections from parent to child will pass through this area. Real wire lengths could be more or less than this estimate; some pins in the parent may be directly adjacent to child entities, or they may have to traverse a distance greater than just the interconnect area. Unfortunately, a more rigorous estimation would require some information about the transistor layout.
Fig. 2.6: The clock network. Squares represent CLBs, and the wires represent the clock network.
Table 2.5: Local interconnect wirelength and capacitance. \( C_{\text{inv}} \) is the input capacitance of a minimum-sized inverter.

<table>
<thead>
<tr>
<th>Connection from Entity Pin to:</th>
<th>Estimated Wirelength</th>
<th>Transistor Capacitance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct (Input or Output)</td>
<td>( 0.5 \cdot L_{\text{interc}} )</td>
<td>( 0 )</td>
</tr>
<tr>
<td>Many-to-1 (Input or Output)</td>
<td>( 0.5 \cdot L_{\text{interc}} )</td>
<td>( C_{\text{INV}} )</td>
</tr>
<tr>
<td>Complete m:n (Input)</td>
<td>( 0.5 \cdot L_{\text{interc}} + L_{\text{crossbar}} )</td>
<td>( n \cdot C_{\text{INV}} )</td>
</tr>
<tr>
<td>Complete m:n (Output)</td>
<td>( 0.5 \cdot L_{\text{interc}} )</td>
<td>( C_{\text{INV}} )</td>
</tr>
</tbody>
</table>

Table 2.5 details how local wire lengths are determined as a function of entity and interconnect areas. It is assumed that each wire connecting a pin of a \( \text{pb}_\text{type} \) to an interconnect structure is of length \( 0.5 \cdot L_{\text{interc}} \). In reality, this length depends on the actual transistor layout, and may be much larger or much smaller than the estimated value. If desired, the user can override the 0.5 constant in the architecture file:

```xml
<architecture>
  <power>
    <local_interconnect factor="0.5"/>
  </power>
</architecture>
```
Buffer Sizing

In the power estimator, a buffer size refers to the size of the final stage of multi-stage buffer (if small, only a single stage is used). The specified size is the $\frac{W}{L}$ of the NMOS transistor. The PMOS transistor will automatically be sized larger. Generally, buffers are sized depending on the load capacitance, using the following equation:

$$\text{Buffer Size} = \frac{1}{2 \cdot f_{LE}} \cdot \frac{C_{Load}}{C_{INV}}$$

In this equation, $C_{INV}$ is the input capacitance of a minimum-sized inverter, and $f_{LE}$ is the logical effort factor. The logical effort factor is the gain between stages of the multi-stage buffer, which by default is 4 (minimal delay). The term $(2 \cdot f_{LE})$ is used so that the ratio of the final stage to the driven capacitance is smaller. This produces a much lower-area, lower-power buffer that is still close to the optimal delay, more representative of common design practises. The logical effort factor can be modified in the architecture file:

```xml
<architecture>
  <power>
    <buffers logical_effor_factor="4"/>
  </power>
</architecture>
```

Local Interconnect Capacitance

If using the auto-size or wire-length options (Architecture Modelling), the local interconnect capacitance must be specified. This is specified in the units of Farads/meter.

```xml
<architecture>
  <power>
    <local_interconnect C_wire="2.5e-15"/>
  </power>
</architecture>
```

2.8 Tasks

Tasks provide a framework for running the VTR flow on multiple benchmarks, architectures and with multiple CAD tool parameters.

A task specifies a set of benchmark circuits, architectures and CAD tool parameters to be used. By default, tasks execute the `run_vtr_flow` script for every circuit/architecture/CAD parameter combination.

2.8.1 Example Tasks

- **basic_flow**: Runs the VTR flow mapping a simple Verilog circuit to an FPGA architecture.
- **timing**: Runs the flagship VTR benchmarks on a comprehensive, realistic architecture file.
- **timing_chain**: Same as timing but with carry chains.
- **regression_mcnc**: Runs VTR on the historical MCNC benchmarks on a legacy architecture file. (Note: This is only useful for comparing to the past, it is not realistic in the modern world)
- **regression_titan/titan_small**: Runs a small subset of the Titan benchmarks targetting a simplified Altera Stratix IV (commercial FPGA) architecture capture
- **regression_fpu_hard_block_arch**: Custom hard FPU logic block architecture
2.8.2 Directory Layout

All of VTR’s included tasks are located here:

`$VTR_ROOT/vtr_flow/tasks`

If users wishes to create their own task, they must do so in this location. All tasks must contain a configuration file located here:

`$VTR_ROOT/vtr_flow/tasks/<task_name>/config/config.txt`

Fig. 2.8 illustrates the directory layout for a VTR task. Every time the task is run a new `run<#>` directory is created to store the output files, where `<#>` is the smallest integer to make the run directory name unique. The symbolic link `latest` will point to the most recent `run<#>` directory.

![Task directory layout diagram](diagram.png)

Fig. 2.8: Task directory layout.

2.8.3 Creating a New Task

1. Create the folder `$VTR_ROOT/vtr_flow/tasks/<task_name>`
2. Create the folder `$VTR_ROOT/vtr_flow/tasks/<task_name>/config`
3. Create and configure the file `$VTR_ROOT/vtr_flow/tasks/<task_name>/config/config.txt`
2.8.4 Task Configuration File

The task configuration file contains key/value pairs separated by the = character. Comment line are indicted using the # symbol.

Example configuration file:

```
# Path to directory of circuits to use
circuits_dir=benchmarks/verilog

# Path to directory of architectures to use
archs_dir=arch/timing

# Add circuits to list to sweep

circuit_list_add=ch_intrinsics.v

circuit_list_add=diffeq1.v

# Add architectures to list to sweep

arch_list_add=k6_N10_memSize16384_memData64_stratix4_based_timing_sparse.xml

# Parse info and how to parse

parse_file=vpr_standard.txt
```

Note: run_vtr_task will invoke the script (default run_vtr_flow) for the cartesian product of circuits, architectures and script parameters specified in the config file.

2.8.5 Required Fields

- **circuit_dir**: Directory path of the benchmark circuits.
  
  Absolute path or relative to $VTR_ROOT/vtr_flow/.

- **arch_dir**: Directory path of the architecture XML files.
  
  Absolute path or relative to $VTR_ROOT/vtr_flow/.

- **circuit_list_add**: Name of a benchmark circuit file.
  
  Use multiple lines to add multiple circuits.

- **arch_list_add**: Name of an architecture XML file.
  
  Use multiple lines to add multiple architectures.

- **parse_file**: Parse Configuration file used for parsing and extracting the statistics.
  
  Absolute path or relative to $VTR_ROOT/vtr_flow/parselparse_config.
2.8.6 Optional Fields

• **script_path**: Script to run for each architecture/circuit combination.
  
  Absolute path or relative to $VTR_ROOT/vtr_flow/scripts/ or $VTR_ROOT/vtr_flow/tasks/
  <task_name>/config/)
  
  **Default**: run_vtr_flow
  
  Users can set this option to use their own script instead of the default. The circuit path will be provided
  as the first argument, and architecture path as the second argument to the user script.

• **script_params_common**: Common parameters to be passed to all script invocations.
  
  This can be used, for example, to run partial VTR flows.
  
  **Default**: none

• **script_params**: Alias for script_params_common

• **script_params_list_add**: Adds a set of command-line arguments
  
  Multiple script_params_list_add can be provided which are addded to the cartesian product of con-
  figurations to be evaluated.

• **sdc_dir**: Directory path to benchmark SDC files.
  
  Absolute path or relative to $VTR_ROOT/vtr_flow/.
  
  If provided, each benchmark will look for a similarly named SDC file.
  
  For instance with circuit_list_add=my_circuit.v or circuit_list_add=my_circuit.
  blif, the flow would look for an SDC file named my_circuit.sdc within the specified sdc_dir.

• **includes_dir**: Directory path to benchmark _include_ files
  
  Absolute path or relative to $VTR_ROOT/vtr_flow/.
  
  Note: Multiple _includes_dir_ are NOT allowed in a task config file.

• **include_list_add**: A path to an _include_ file, which is relative to _includes_dir_
  
  Multiple _include_list_add_ can be provided.
  
  _include_ files could act as the top module complimentary, like definitions, macros or sub-modules.
  
  Note: _include_ files will be shared among all benchmark circuits in the task config file.

• **pass_requirements_file**: Pass Requirements file.
  
  Absolute path or relative to $VTR_ROOT/vtr_flow/parse/pass_requirements/ or $VTR_ROOT/
  vtr_flow/tasks/<task_name>/config/
  
  **Default**: none
2.9 run_vtr_flow

This script runs the VTR flow for a single benchmark circuit and architecture file.

The script is located at:

```
$VTR_ROOT/vtr_flow/scripts/run_vtr_flow.py
```

### 2.9.1 Basic Usage

At a minimum `run_vtr_flow.py` requires two command-line arguments:

```
run_vtr_flow.py <circuit_file> <architecture_file>
```

where:

- `<circuit_file>` is the circuit to be processed
- `<architecture_file>` is the target FPGA architecture

**Note:** The script will create a `.temp` directory, unless otherwise specified with the `--temp_dir` option. The circuit file and architecture file will be copied to the temporary directory. All stages of the flow will be run within this directory. Several intermediate files will be generated and deleted upon completion. **Users should ensure that no important files are kept in this directory as they may be deleted.**

### 2.9.2 Output

The standard output of the script will produce a single line with the format:

```
<architecture>/<circuit_name>...<status>
```

If execution completed successfully the status will be ‘OK’. Otherwise, the status will indicate which stage of execution failed.

The script will also produce an output files (*.out) for each stage, containing the standout output of the executable(s).

### 2.9.3 Advanced Usage

Additional optional command arguments can also be passed to `run_vtr_flow.py`:

```
run_vtr_flow.py <circuit_file> <architecture_file> [options] [vpr_options]
```

where:

- `<options>` are additional arguments passed to `run_vtr_flow.py` (described below),
- `<vpr_options>` are any arguments not recognized by `run_vtr_flow.py`. These will be forwarded to VPR.

For example:

```
run_vtr_flow.py my_circuit.v my_arch.xml --track_memory_usage --pack --place
```
will run the VTR flow to map the circuit my_circuit.v onto the architecture my_arch.xml; the arguments --pack and --place will be passed to VPR (since they are unrecognized arguments to run_vtr_flow.py). They will cause VPR to perform only *packing and placement*.

### 2.9.4 Detailed Command-line Options

**Note:** Any options not recognized by this script is forwarded to VPR.

**-starting_stage** *<stage>*
Start the VTR flow at the specified stage.

Accepted values:
- odin
- abc
- scripts
- vpr

Default: odin

**-ending_stage** *<stage>*
End the VTR flow at the specified stage.

Accepted values:
- odin
- abc
- scripts
- vpr

Default: vpr

**-power**
Enables power estimation.

See *Power Estimation*

**-cmos_tech** *<file>*
CMOS technology XML file.

See *Technology Properties*

**-delete_intermediate_files**
Delete intermediate files (i.e. .dot, .xml, .rc, etc)

**-delete_result_files**
Delete result files (i.e. VPR’s .net, .place, .route outputs)

**-track_memory_usage**
Record peak memory usage and additional statistics for each stage.

**Note:** Requires /usr/bin/time -v command. Some operating systems do not report peak memory.

**Default:** off
-limit_memory_usage
  Kill benchmark if it is taking up too much memory to avoid slow disk swaps.

  Note: Requires ulimit -Sv command.

  Default: off

-timeout <float>
  Maximum amount of time to spend on a single stage of a task in seconds.

  Default: 14 days

-temp_dir <path>
  Temporary directory used for execution and intermediate files. The script will automatically create this directory
  if necessary.

  Default: ./temp

-valgrind
  Run the flow with valgrind while using the following valgrind options:

  • –leak-check=full
  • –errors-for-leak-kinds=none
  • –error-exitcode=1
  • –track-origins=yes

-min_hard_mult_size <int>
  Tells ODIN II the minimum multiplier size that should be implemented using hard multiplier (if available).
  Smaller multipliers will be implemented using soft logic.

  Default: 3

-min_hard_adder_size <int>
  Tells ODIN II the minimum adder size that should be implemented using hard adders (if available). Smaller
  adders will be implemented using soft logic.

  Default: 1

-adder_cin_global
  Tells ODIN II to connect the first cin in an adder/subtractor chain to a global gnd/vdd net. Instead of creating a
  dummy adder to generate the input signal of the first cin port of the chain.

-odin_xml <path_to_custom_xml>
  Tells VTR flow to use a custom ODIN II configuration value. The default behavior is to use the
  vtr_flow/misc/basic_odin_config_split.xml. Instead, an alternative config file might be supplied; compare the
  default and vtr_flow/misc/custom_odin_config_no_mults.xml for usage scenarios. This option is needed for
  running the entire VTR flow with additional parameters for ODIN II that are provided from within the .xml file.
2.10 run_vtr_task

This script is used to execute one or more tasks (i.e. collections of benchmarks and architectures).

See also:

See Tasks for creation and configuration of tasks.

This script runs the VTR flow for a single benchmark circuit and architecture file.

The script is located at:

```
$VTR_ROOT/vtr_flow/scripts/run_vtr_task.py
```

2.10.1 Basic Usage

Typical usage is:

```
run_vtr_task.py <task_name1> <task_name2> ...
```

Note: At least one task must be specified, either directly as a parameter or via the -l options.

2.10.2 Output

Each task will execute the script specified in the configuration file for every benchmark/circuit/option combination. The standard output of the underlying script will be forwarded to the output of this script.

If golden results exist (see parse_vtr_task), they will be inspected for runtime and memory usage.

2.10.3 Detailed Command-line Options

- **s** `<script_param> ...`
  Treat the remaining command line options as parameters to forward to the underlying script (e.g. run_vtr_flow).

- **j** `<N>`
  Perform parallel execution using N threads.

  Note: Only effective for -system local

  **Warning:** Large benchmarks will use very large amounts of memory (several to 10s of gigabytes). Because of this, parallel execution often saturates the physical memory, requiring the use of swap memory, which significantly slows execution. Be sure you have allocated a sufficiently large swap memory or errors may result.

- **l** `<task_list_file>`
  A file containing a list of tasks to execute.

  Each task name should be on a separate line, e.g.
-system {local | scripts}
Controls how the actions (e.g. invocations of run_vtr_flow) are called.

**Default:** local

- local: Runs the flow invocations on the local machine (potentially in parallel with the -j option).

  Example:

  ```
  # From $VTR_ROOT/vtr_flow/tasks
  $ ../scripts/run_vtr_task.py regression_tests/vtr_reg_basic/basic_timing
  regression_tests/vtr_reg_basic/basic_timing: k6_N10_mem32K_40nm.xml/ch_intrinsics.v/common OK (took 2.24 seconds)
  regression_tests/vtr_reg_basic/basic_timing: k6_N10_mem32K_40nm.xml/diffeq1.v/common OK (took 10.94 seconds)
  ```

- scripts: Prints out all the generated script files (instead of calling them to run all the flow invocations).

  Example:

  ```
  # From $VTR_ROOT/vtr_flow/tasks
  $ ../scripts/run_vtr_task.py regression_tests/vtr_reg_basic/basic_timing -system scripts
  /project/trees/vtr/vtr_flow/tasks/regression_tests/vtr_reg_basic/basic_timing/run001/k6_N10_mem32K_40nm.xml/ch_intrinsics.v/common/vtr_flow.sh
  /project/trees/vtr/vtr_flow/tasks/regression_tests/vtr_reg_basic/basic_timing/run001/k6_N10_mem32K_40nm.xml/diffeq1.v/common/vtr_flow.sh
  ```

Each generated script file (vtr_flow.sh) corresponds to a particular flow invocation generated by the task, and is located within its own directory.

This list of scripts can be used to run flow invocations on different computing infrastructures (e.g. a compute cluster).

**Using the output of -system scripts to run a task**

An example of using the output would be:

```
# From $VTR_ROOT/vtr_flow/tasks
$ ../scripts/run_vtr_task.py regression_tests/vtr_reg_basic/basic_timing -system scripts | parallel -j4 'cd $(dirname {}) && {}'
regression_tests/vtr_reg_basic/basic_timing: k6_N10_mem32K_40nm.xml/ch_intrinsics.v/common OK (took 2.11 seconds)
regression_tests/vtr_reg_basic/basic_timing: k6_N10_mem32K_40nm.xml/diffeq1.v/common OK (took 10.94 seconds)
```

where {} is a special variable interpreted by the parallel command to represent the input line (i.e. a script, see parallel's documentation for details). This will run the scripts generated by run_vtr_task.py in parallel (up to 4 at-a-time due to -j4). Each script is invoked in the script's containing directory (cd $(dirname {})), which mimics the behaviour of -system local -j4.
Note: While this example shows how the flow invocations could be run locally, similar techniques can be used to submit jobs to other compute infrastructures (e.g. a compute cluster)

Determining Resource Requirements

Often, when running in a cluster computing enviroment, it is useful to know what compute resources are required for each flow invocation.

Each generated vtr_flow.sh scripts contains the expected run-time and memory use of each flow invocation (derived from golden reference results). These can be inspected to determine compute requirements:

```
$ grep VTR_RUNTIME_ESTIMATE_SECONDS /project/trees/vtr/vtr_flow/tasks/...
xml/ch_intrinsics.v/common/vtr_flow.sh
VTR_RUNTIME_ESTIMATE_SECONDS=2.96
```

```
$ grep VTR_MEMORY_ESTIMATE_BYTES /project/trees/vtr/vtr_flow/tasks/...
xml/ch_intrinsics.v/common/vtr_flow.sh
VTR_MEMORY_ESTIMATE_BYTES=63422464
```

Note: If the resource estimates are unknown they will be set to 0

2.11 parse_vtr_flow

This script parses statistics generated by a single execution of the VTR flow.

Note: If the user is using the Tasks framework, parse_vtr_task should be used.

The script is located at:

```
$VTR_ROOT/vtr_flow/scripts/python_libs/vtr/parse_vtr_flow.py
```

2.11.1 Usage

Typical usage is:

```
parse_vtr_flow.py <parse_path> <parse_config_file>
```

where:

- `<parse_path>` is the directory path that contains the files to be parsed (e.g. vpr.out, odin.out, etc).
- `<parse_config_file>` is the path to the Parse Configuration file.
2.11.2 Output

The script will produce no standard output. A single file named `parse_results.txt` will be produced in the `<parse_path>` folder. The file is tab delimited and contains two lines. The first line is a list of field names that were searched for, and the second line contains the associated values.

2.12 parse_vtr_task

This script is used to parse the output of one or more Tasks. The values that will be parsed are specified using a Parse Configuration file, which is specified in the task configuration.

The script will always parse the results of the latest execution of the task.

The script is located at:

```
$VTR_ROOT/vtr_flow/scripts/python_libs/vtr/parse_vtr_task.py
```

2.12.1 Usage

Typical usage is:

```
parse_vtr_task.py <task_name1> <task_name2> ...
```

Note: At least one task must be specified, either directly as a parameter or through the `-l` option.

2.12.2 Output

By default this script produces no standard output. A tab delimited file containing the parse results will be produced for each task. The file will be located here:

```
$VTR_ROOT/vtr_flow/tasks/<task_name>/run<#>/parse_results.txt
```

If the `-check_golden` is used, the script will output one line for each task in the format:

```
<task_name>...<status>
```

where `<status>` will be [Pass], [Fail], or [Error].

2.12.3 Detailed Command-line Options

- `-l <task_list_file>`
  A file containing a list of tasks to parse. Each task name should be on a separate line.

- `-create_golden`
  The results will be stored as golden results. If previous golden results exist they will be overwritten.

  The golden results are located here:

  ```
  $VTR_ROOT/vtr_flow/tasks/<task_name>/config/golden_results.txt
  ```
The results will be compared to the golden results using the Pass Requirements file specified in the task configuration. A Pass or Fail will be output for each task (see below). In order to compare against the golden results, they must already exist, and have the same architectures, circuits and parse fields, otherwise the script will report Error.

If the golden results are missing, or need to be updated, use the -create_golden option.

### 2.13 Parse Configuration

A parse configuration file defines a set of values that will be searched for within the specified files.

#### 2.13.1 Format

The configuration file contains one line for each value to be searched for. Each line contains a semicolon delimited tuple in the following format:

```
<field_name>;<file_to_search_within>;<regex>;<default_value>
```

- `<field_name>`: The name of the value to be searched for.
  
  This name is used when generating the output files of `parse_vtr_task` and `parse_vtr_flow`.

- `<file_to_search_within>`: The name of the file that will be searched (vpr.out, odin.out, etc.)

- `<regex>`: A perl regular expression used to find the desired value.
  
  The regex must contain a single grouping ( ) which will contain the desired value to be recorded.

- `<default_value>`: The default value for the given `<field_name>` if the `<regex>` does not match.
  
  If no `<default_value>` is specified the value -1 is used.

Or an include directive to import parsing patterns from a separate file:

```
%include "<filepath>"
```

- `<filepath>` is a file containing additional parse specifications which will be included in the current file.

Comments can be specified with #. Anything following a # is ignored.

#### 2.13.2 Example File

The following is an example parse configuration file:

```
vpr_status;output.txt;vpr_status=(.*)
vpr_seconds;output.txt;vpr_seconds=(\d+)
width;vpr.out;Best routing used a channel width factor of (\d+)
pack_time;vpr.out;Packing took (.* seconds
place_time;vpr.out;Placement took (.* seconds
route_time;vpr.out;Routing took (.* seconds
num_pre_packed_nets;vpr.out;Total Nets: (\d+)
num_pre_packed_blocks;vpr.out;Total Blocks: (\d+)
num_post_packed_nets;vpr.out;Netlist num_nets: \s*(\d+)
num_clb;vpr.out;Netlist clb blocks: \s*(\d+)
```

(continues on next page)
2.14 Pass Requirements

The `parse_vtr_task` scripts allow you to compare an executed task to a golden reference result. The comparison, which is performed when using the `parse_vtr_task.py -check_golden` option, which reports either Pass or Fail. The requirements that must be met to qualify as a Pass are specified in the pass requirements file.

2.14.1 Task Configuration

Tasks can be configured to use a specific pass requirements file using the `pass_requirements_file` keyword in the `Tasks` configuration file.

2.14.2 File Location

All provided pass requirements files are located here:

```
$VTR_ROOT/vtr_flow/parse/pass_requirements
```

Users can also create their own pass requirement files.

2.14.3 File Format

Each line of the file indicates a single metric, data type and allowable values in the following format:

```
<metric>;<requirement>
```

- `<metric>`: The name of the metric.
- `<requirement>`: The metric’s pass requirement.

Valid requirement types are:

- `Equal()`: The metric value must exactly match the golden reference result.
– Range(<min_ratio>,<max_ratio>): The metric value (normalized to the golden result) must be between <min_ratio> and <max_ratio>.
– RangeAbs(<min_ratio>,<max_ratio>,<abs_threshold>): The metric value (normalized to the golden result) must be between <min_ratio> and <max_ratio>, or the metric’s absolute value must be below <abs_threshold>.

Or an include directive to import metrics from a separate file:

```verbatim
%include "<filepath>"
```

• `<filepath>`: a relative path to another pass requirements file, whose metric pass requirements will be added to the current file.

In order for a Pass to be reported, all requirements must be met. For this reason, all of the specified metrics must be included in the parse results (see Parse Configuration).

Comments can be specified with #. Anything following a # is ignored.

### 2.14.4 Example File

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vpr_status;Equal()</td>
<td>#Pass if precisely equal</td>
</tr>
<tr>
<td>vpr_seconds;RangeAbs(0.80,1.40,2)</td>
<td>#Pass if within -20%, or +40%, or absolute value less than 2</td>
</tr>
<tr>
<td>num_pre_packed_nets;Range(0.90,1.10)</td>
<td>#Pass if within +/-10%</td>
</tr>
<tr>
<td>%include &quot;routing_metrics.txt&quot;</td>
<td>#Import all pass requirements from the file</td>
</tr>
</tbody>
</table>

### 2.15 VTR Flow Python library

The VTR flow can be imported and implemented as a python library. Below are the descriptions of the useful functions.

#### 2.15.1 VTR flow

#### 2.15.2 ODIN II

#### 2.15.3 ABC

#### 2.15.4 ACE

#### 2.15.5 VPR
CHAPTER THREE

FPGA ARCHITECTURE DESCRIPTION

VTR uses an XML-based architecture description language to describe the targeted FPGA architecture. This flexible description language allows the user to describe a large number of hypothetical and commercial-like FPGA architectures.

See the Architecture Modeling for an introduction to the architecture description language. For a detailed reference on the supported options see the Architecture Reference.

3.1 Architecture Reference

This section provides a detailed reference for the FPGA Architecture description used by VTR. The Architecture description uses XML as its representation format.

As a convention, curly brackets {} represents an option with each option separated by |. For example, a={1 | 2 | open} means field a can take a value of 1, 2, or open.

3.1.1 Top Level Tags

The first tag in all architecture files is the <architecture> tag. This tag contains all other tags in the architecture file. The architecture tag contains the following tags:

- <models>
- <tiles>
- <layout>
- <device>
- <switchlist>
- <segmentlist>
- <directlist>
- <complexblocklist>
### 3.1.2 Recognized BLIF Models (<models>)

The `<models>` tag contains `<model name="string" never_prune="string">` tags. Each `<model>` tag describes the BLIF `.subckt` model names that are accepted by the FPGA architecture. The name of the model must match the corresponding name of the BLIF model.

The `never_prune` flag is optional and can be either:

- `false` (default)
- `true`

Normally blocks with no output nets are pruned away by the netlist sweepers in vpr (removed from the netlist); this is the default behaviour. If `never_prune = “true”` is set on a model, then blocks that are instances of that model will not be swept away during netlist cleanup. This can be helpful for some special blocks that do have only input nets and are required to be placed on the device for some features to be active, so space on the chip is still reserved for them, despite them not driving any connection. One example is the IDELAYCTRL of the Series7 devices, which takes as input a reference clock and internally controls and synchronizes all the IDELAYs in a specific clock region, with no output net necessary for it to function correctly.

---

**Note:** Standard blif structures (.names, .latch, .input, .output) are accepted by default, so these models should not be described in the `<models>` tag.

---

Each model tag must contain 2 tags: `<input_ports>` and `<output_ports>`. Each of these contains `<port>` tags:

```xml
<port name="string" is_clock="{0 | 1} clock="string" combinational_sink_ports="string1 string2 .."/>
```

**Required Attributes**

- `name` – The port name.

**Optional Attributes**

- `is_clock` – Identifies if the port as a clock port.

**See also:**

The *Primitive Timing Modelling Tutorial* for usage of `is_clock` to model clock control blocks such as clock generators, clock buffers/gates and clock muxes.

Default: 0

- `clock` – Indicates the port is sequential and controlled by the specified clock (which must be another port on the model marked with `is_clock=1`). Default: port is treated as combinational (if unspecified)

- `combinational_sink_ports` – A space-separated list of output ports which are combinatorially connected to the current input port. Default: No combinational connections (if unspecified)

---

Defines the port for a model.

An example models section containing a combinational primitive `adder` and a sequential primitive `single_port_ram` follows:

```xml
<models>
  <model name="single_port_ram">
    <input_ports>
      ...
    </input_ports>
  </model>
</models>
```

(continues on next page)
<port name="we" clock="clk"/>
<port name="addr" clock="clk" combinational_sink_ports="out"/>
<port name="data" clock="clk" combinational_sink_ports="out"/>
<port name="clk" is_clock="1"/>
</input_ports>
</output_ports>
</model>
</models>

Note that for single_port_ram above, the ports we, addr, data, and out are sequential since they have a clock specified. Additionally addr and data are shown to be combinatorially connected to out; this corresponds to an internal timing path between the addr and data input registers, and the out output registers.

For the adder the input ports a, b and cin are each combinatorially connected to the output ports cout and sumout (the adder is a purely combinational primitive).

See also:

For more examples of primitive timing modeling specifications see the Primitive Block Timing Modeling Tutorial

3.1.3 Global FPGA Information

<tiles>content</tiles>
Content inside this tag contains a group of <pb_type> tags that specify the types of functional blocks and their properties.

<layout/>
Content inside this tag specifies device grid layout.

See also:
FPGA Grid Layout

<device>content</device>
Content inside this tag specifies device information.

See also:
FPGA Device Information

<switchlist>content</switchlist>
Content inside this tag contains a group of <switch> tags that specify the types of switches and their properties.
3.1.4 FPGA Grid Layout

The valid tags within the `<layout>` tag are:

```
<auto_layout aspect_ratio="float">

Optional Attributes
  • aspect_ratio – The device grid’s target aspect ratio (width/height)
    Default: 1.0

Defines a scalable device grid layout which can be automatically scaled to a desired size.

Note: At most one `<auto_layout>` can be specified.
```

```
<fixed_layout name="string" width="int" height="int">

Required Attributes
  • name – The unique name identifying this device grid layout.
  • width – The device grid width
  • height – The device grid height

Defines a device grid layout with fixed dimensions.

Note: Multiple `<fixed_layout>` tags can be specified.
```

Each `<auto_layout>` or `<fixed_layout>` tag should contain a set of grid location tags.

**Grid Location Priorities**

Each grid location specification has an associated numeric `priority`. Larger priority location specifications override those with lower priority.

*Note:* If a grid block is partially overlapped by another block with higher priority the entire lower priority block is removed from the grid.
Empty Grid Locations

Empty grid locations can be specified using the special block type EMPTY.

Note: All grid locations default to EMPTY unless otherwise specified.

Grid Location Expressions

Some grid location tags have attributes (e.g. startx) which take an expression as their argument. An expression can be an integer constant, or simple mathematical formula evaluated when constructing the device grid.

Supported operators include: +, -, *, /, along with ( and ) to override the default evaluation order. Expressions may contain numeric constants (e.g. 7) and the following special variables:

- \( W \): The width of the device
- \( H \): The height of the device
- \( w \): The width of the current block type
- \( h \): The height of the current block type

Warning: All expressions are evaluated as integers, so operations such as division may have their result truncated.

As an example consider the expression \( \frac{W}{2} - \frac{w}{2} \). For a device width of 10 and a block type of width 3, this would be evaluated as \( \left\lfloor \frac{10}{2} \right\rfloor - \left\lfloor \frac{3}{2} \right\rfloor = 5 - 1 = 4 \).

Grid Location Tags

\(<fill type="string" priority="int"/>\)

Required Attributes

- **type** – The name of the top-level complex block type (i.e. \(<\text{pb_type}>\)) being specified.
- **priority** – The priority of this layout specification. Tags with higher priority override those with lower priority.

Fills the device grid with the specified block type.

Example:

\(<!-- Fill the device with CLB blocks -->\>
\(<fill type="CLB" priority="1"/>\>

\(<perimeter type="string" priority="int"/>\)

Required Attributes

- **type** – The name of the top-level complex block type (i.e. \(<\text{pb_type}>\)) being specified.
- **priority** – The priority of this layout specification. Tags with higher priority override those with lower priority.
Fig. 3.1: <fill> CLB example
Sets the perimeter of the device (i.e. edges) to the specified block type.

**Note:** The perimeter includes the corners

Example:

```xml
<!-- Create io blocks around the device perimeter -->
<perimeter type="io" priority="10"/>
```

Fig. 3.2: `<perimeter> io example`

```xml
<corners type="string" priority="int"/>
```

**Required Attributes**

- **type** – The name of the top-level complex block type (i.e. `<pb_type>`) being specified.
- **priority** – The priority of this layout specification. Tags with higher priority override those with lower priority.
Sets the corners of the device to the specified block type.

Example:

```xml
<!-- Create PLL blocks at all corners -->
<corners type="PLL" priority="20"/>
```

Fig. 3.3: <corners> PLL example

```xml
<single type="string" priority="int" x="expr" y="expr"/>
```

**Required Attributes**

- **type** – The name of the top-level complex block type (i.e. `<pb_type>`) being specified.
- **priority** – The priority of this layout specification. Tags with higher priority override those with lower priority.
- **x** – The horizontal position of the block type instance.
- **y** – The vertical position of the block type instance.
Specifies a single instance of the block type at a single grid location.

Example:

```xml
<!-- Create a single instance of a PCIE block (width 3, height 5) at location (1,1) -->
<single type="PCIE" x="1" y="1" priority="20"/>
```

Fig. 3.4: `<single>` PCIE example

```
<col type="string" priority="int" startx="expr" repeatx="expr" starty="expr" incry="expr"/>
```

**Required Attributes**

- **type** – The name of the top-level complex block type (i.e. `<pb_type>`) being specified.
- **priority** – The priority of this layout specification. Tags with higher priority override those with lower priority.
- **startx** – An expression specifying the horizontal starting position of the column.
Optional Attributes

- **repeatx** – An expression specifying the horizontal repeat factor of the column.
- **starty** – An expression specifying the vertical starting offset of the column.
  
  Default: 0
- **incr** – An expression specifying the vertical increment between block instantiations within the region.
  
  Default: h

Creates a column of the specified block type at startx.

If repeatx is specified the column will be repeated wherever \( x = startx + k \cdot repeatx \), is satisfied for any positive integer \( k \).

A non-zero starty is typically used if a `<perimeter>` tag is specified to adjust the starting position of blocks with height > 1.

Example:

```xml
<!-- Create a column of RAMs starting at column 2, and repeating every 3 columns -->
<col type="RAM" startx="2" repeatx="3" priority="3"/>
```

Example:

```xml
<!-- Create IOs around the device perimeter -->
<perimeter type="io" priority=10"/>

<!-- Create a column of RAMs starting at column 2, and repeating every 3 columns. Note that a vertical offset of 1 is needed to avoid overlapping the IOs -->
<col type="RAM" startx="2" repeatx="3" starty="1" priority="3"/>
```

`<row type="string" priority="int" starty="expr" repeaty="expr" startx="expr"/>`

Required Attributes

- **type** – The name of the top-level complex block type (i.e. `<pb_type>`) being specified.
- **priority** – The priority of this layout specification. Tags with higher priority override those with lower priority.
- **starty** – An expression specifying the vertical starting position of the row.

Optional Attributes

- **repeaty** – An expression specifying the vertical repeat factor of the row.
- **startx** – An expression specifying the horizontal starting offset of the row.
  
  Default: 0
- **incr** – An expression specifying the horizontal increment between block instantiations within the region.
  
  Default: w

Creates a row of the specified block type at starty.
Fig. 3.5: <col> RAM example
Fig. 3.6: <col> RAM and <perimeter> io example
If `repeaty` is specified the column will be repeated wherever \( y = starty + k \cdot repeaty \), is satisfied for any positive integer \( k \).

A non-zero `startx` is typically used if a `<perimeter>` tag is specified to adjust the starting position of blocks with width > 1.

Example:

```xml
<!-- Create a row of DSPs (width 1, height 3) at row 1 and repeating every 7th row -->
<row type="DSP" starty="1" repeaty="7" priority="3"/>
```

![Fig. 3.7: <row> DSP example](image)

Required Attributes

- **type** – The name of the top-level complex block type (i.e. `<pb_type>`) being specified.
• **priority** – The priority of this layout specification. Tags with higher priority override those with lower priority.

Optional Attributes

• **startx** – An expression specifying the horizontal starting position of the region (inclusive).
  
  Default: 0

• **endx** – An expression specifying the horizontal ending position of the region (inclusive).
  
  Default: W - 1

• **repeatx** – An expression specifying the horizontal repeat factor of the column.

• **incrx** – An expression specifying the horizontal increment between block instantiations within the region.
  
  Default: w

• **starty** – An expression specifying the vertical starting position of the region (inclusive).
  
  Default: 0

• **endy** – An expression specifying the vertical ending position of the region (inclusive).
  
  Default: H - 1

• **repeaty** – An expression specifying the vertical repeat factor of the column.

• **incry** – An expression specifying the vertical increment between block instantiations within the region.
  
  Default: h

Fills the rectangular region defined by \((\text{startx}, \text{starty})\) and \((\text{endx}, \text{endy})\) with the specified block type.

**Note:** endx and endy are included in the region

If **repeatx** is specified the region will be repeated wherever \(x = \text{startx} + k_1 \times \text{repeatx}\), is satisfied for any positive integer \(k_1\).

If **repeaty** is specified the region will be repeated wherever \(y = \text{starty} + k_2 \times \text{repeaty}\), is satisfied for any positive integer \(k_2\).

Example:

```xml
<!-- Fill RAMs within the rectangular region bounded by (1,1) and (5,4) -->
<region type="RAM" startx="1" endx="5" starty="1" endy="4" priority="4"/>
```

Example:

```xml
<!-- Create RAMs every 2nd column within the rectangular region bounded by (1,1) and (5,4) -->
<region type="RAM" startx="1" endx="5" starty="1" endy="4" incrx="2" priority="4"/>
```

Example:

```xml
<!-- Fill RAMs within a rectangular 2x4 region and repeat every 3 horizontal and 5 vertical units -->
<region type="RAM" startx="1" endx="2" starty="1" endy="4" repeatx="3" repeaty="5" priority="4"/>
```
Fig. 3.8: <region> RAM example
Fig. 3.9: <region> RAM increment example
Fig. 3.10: <region> RAM repeat example
Example:

```html
<!-- Create a 3x3 mesh of NoC routers (width 2, height 2) whose relative positions will scale with the device dimensions -->
<region type="NoC" startx="W/4 - w/2" starty="W/4 - w/2" inctx="W/4" incry="W/4" priority="3"/>
```

Fig. 3.11: <region> NoC mesh example
Grid Layout Example

```xml
<layout>
  <!-- Specifies an auto-scaling square FPGA floorplan -->
  <auto_layout aspect_ratio="1.0">
    <!-- Create I/Os around the device perimeter -->
    <perimeter type="io" priority="10"/>
    <!-- Nothing in the corners -->
    <corners type="EMPTY" priority="100"/>
    <!-- Create a column of RAMs starting at column 2, and repeating every 3 columns. Note that a vertical offset (starty) of 1 is needed to avoid overlapping the I/Os-->
    <col type="RAM" startx="2" repeatx="3" starty="1" priority="3"/>
    <!-- Create a single PCIE block along the bottom, overriding I/O and RAM slots -->
    <single type="PCIE" x="3" y="0" priority="20"/>
    <!-- Create an additional row of I/Os just above the PCIE, which will not override RAMs -->
    <row type="io" starty="5" priority="2"/>
    <!-- Fill remaining with CLBs -->
    <fill type="CLB" priority="1"/>
  </auto_layout>
</layout>
```

3.1.5 FPGA Device Information

The tags within the <device> tag are:

```xml
<sizing R_minW_nmos="float" R_minW_pmos="float"/>
```

**Required Attributes**

- **R_minW_nmos** – The resistance of minimum-width nmos transistor. This data is used only by the area model built into VPR.
- **R_minW_pmos** – The resistance of minimum-width pmos transistor. This data is used only by the area model built into VPR.

**Required** Yes

Specifies parameters used by the area model built into VPR.

```xml
<connection_block input_switch_name="string"/>
```

**Required Attributes**

- **switch_name** – Specifies the name of the <switch> in the <switchlist> used to connect routing tracks to block input pins (i.e. the input connection block switch).
Fig. 3.12: Example FPGA grid

Fig. 3.13: Input Pin Diagram.
<area grid_logic_tile_area="float"/>

Specifies the default area used by each 1x1 grid logic tile (in MWTAs), excluding routing.
Used for an area estimate of the amount of area taken by all the functional blocks.

Note: This value can be overridden for specific <pb_type>’s with the ‘area’ attribute.

<switch_block type="{wilton | subset | universal | custom}" fs="int"/>

Required Attributes
- type – The type of switch block to use.
- fs – The value of $F_s$

Required Yes

This parameter controls the pattern of switches used to connect the (inter-cluster) routing segments. Three fairly simple patterns can be specified with a single keyword each, or more complex custom patterns can be specified.

Non-Custom Switch Blocks:
When using bidirectional segments, all the switch blocks have $F_s = 3$ [BFRV92]. That is, whenever horizontal and vertical channels intersect, each wire segment can connect to three other wire segments. The exact topology of which wire segment connects to which can be one of three choices. The subset switch box is the planar or domain-based switch box used in the Xilinx 4000 FPGAs – a wire segment in track 0 can only connect to other wire segments in track 0 and so on. The wilton switch box is described in [Wil97], while the universal switch box is described in [CWW96]. To see the topology of a switch box, simply hit the “Toggle RR” button when a completed routing is on screen in VPR. In general the wilton switch box is the best of these three topologies and leads to the most routable FPGAs.

When using unidirectional segments, one can specify an $F_s$ that is any multiple of 3. We use a modified wilton switch block pattern regardless of the specified switch_block_type. For all segments that start/end at that switch block, we follow the wilton switch block pattern. For segments that pass through the switch block that can also turn there, we cannot use the wilton pattern because a unidirectional segment cannot be driven at an intermediate point, so we assign connections to starting segments following a round robin scheme (to balance mux size).

Note: The round robin scheme is not tileable.

Custom Switch Blocks:
Specifying custom allows custom switch blocks to be described under the <switchblocklist> XML node, the format for which is described in Custom Switch Blocks. If the switch block is specified as custom, the fs field does not have to be specified, and will be ignored if present.

<chan_width_distr>content</chan_width_distr>
Content inside this tag is only used when VPR is in global routing mode. The contents of this tag are described in Global Routing Information.
This defines the default Fc specification, if it is not specified within a <fc> tag inside a top-level complex block. The attributes have the same meaning as the <fc> tag attributes.

### 3.1.6 Switches

The tags within the `<switchlist>` tag specifies the switches used to connect wires and pins together.

```xml
<switch type="{mux|tristate|pass_gate|short|buffer}" name="string" R="float" Cin="float" Cout="float" Cinternal="float" Tdel="float" mux_trans_size="float", power_buf_size="int"/>
```

Describes a switch in the routing architecture.

**Example:**

```xml
<switch type="mux" name="my_awesome_mux" R="551" Cin=".77e-15" Cout="4e-15" → Cinternal="5e-15" Tdel="58e-12" mux_trans_size="2.630740" buf_size="27.645901"/>
```

**Required Attributes**

- **type** – The type of switch:
  - `mux`: An isolating, configurable multiplexer
  - `tristate`: An isolating, configurable tristate-able buffer
  - `pass_gate`: A non-isolating, configurable pass gate
  - `short`: A non-isolating, non-configurable electrical short (e.g. between two segments).
  - `buffer`: An isolating, non-configurable non-tristate-able buffer (e.g. in-line along a segment).

**Isolation**

Isolating switches include a buffer which partition their input and output into separate DC-connected sub-circuits. This helps reduce RC wire delays.

*Non-isolating* switch do **not** isolate their input and output, which can increase RC wire delays.

**Configurability**

Configurable switches can be turned on/off at configuration time.

*Non-configurable* switches can **not** be controlled at configuration time. These are typically used to model non-optional connections such as electrical shorts and in-line buffers.

- **name** – A unique name identifying the switch
- **R** – Resistance of the switch.
- **Cin** – Input capacitance of the switch.
- **Cout** – Output capacitance of the switch.

**Optional Attributes**

- **Cinternal** – Since multiplexers and tristate buffers are modeled as a parallel stream of pass transistors feeding into a buffer, we would expect an additional “internal capacitance” to arise when the pass transistor is enabled and the signal must propagate to the buffer. See diagram of one stream below:
Pass Transistor

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>-------</td>
<td>-------</td>
<td>--------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>--------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>/</td>
</tr>
<tr>
<td>=====</td>
<td>=====</td>
<td>C output</td>
</tr>
<tr>
<td>=====</td>
<td>=====</td>
<td>=====</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Input C** | **Internal C** | **Output C**

**Note:** Only specify a value for multiplexers and/or tristate switches.

- **Tdel** – Intrinsic delay through the switch. If this switch was driven by a zero resistance source, and drove a zero capacitance load, its delay would be: \( T_{\text{del}} + R \cdot C_{\text{out}} \).

The ‘switch’ includes both the mux and buffer mux type switches.

**Note:** Required if no \(<\text{Tdel}>\) tags are specified

**Note:** A \(<\text{switch}>\)’s resistance (\( R \)) and output capacitance (\( C_{\text{out}} \)) have no effect on delay when used for the input connection block, since VPR does not model the resistance/capacitance of block internal wires.

- **buf_size** – Specifies the buffer size in minimum-width transistor area (:term:`MWTA`) units.

  If set to auto, sized automatically from the \( R \) value. This allows you to use timing models without \( R \)’s and \( C \)’s and still be able to measure area.

  **Note:** Required for all isolating switch types.

**Default:** auto

- **mux_trans_size** – Specifies the size (in minimum width transistors) of each transistor in the two-level mux used by mux type switches.

  **Note:** Valid only for mux type switches.

- **power_buf_size** – Used for power estimation. The size is the drive strength of the buffer, relative to a minimum-sized inverter.

\(<\text{Tdel num_inputs="int" delay="float"/>\)

Instead of specifying a single Tdel value, a list of Tdel values may be specified for different values of switch fan-in. Delay is linearly extrapolated/interpolated for any unspecified fanins based on the two closest fanins.

**Required Attributes**

- **num_inputs** – The number of switch inputs (fan-in)
• **delay** – The intrinsic switch delay when the switch topology has the specified number of switch inputs

**Example:**

```
<switch type="mux" name="my_mux" R="522" Cin="3.1e-15" Cout="3e-15" Cinternal="5e-15" mux_trans_size="1.7" buf_size="23">
    <Tdel num_inputs="12" delay="8.00e-11"/>
    <Tdel num_inputs="15" delay="8.4e-11"/>
    <Tdel num_inputs="20" delay="9.4e-11"/>
</switch>
```

**Global Routing Information**

If global routing is to be performed, channels in different directions and in different parts of the FPGA can be set to different relative widths. This is specified in the content within the `<chan_width_distr>` tag.

**Note:** If detailed routing is to be performed, only uniform distributions may be used

```
<x distr="{gaussian|uniform|pulse|delta}" peak="float" width="float" xpeak="float" dc="float"/>
```

**Required Attributes**

- `distr` – The channel width distribution function
- `peak` – The peak value of the distribution

**Optional Attributes**

- `width` – The width of the distribution. Required for `pulse` and `gaussian`.
- `xpeak` – Peak location horizontally. Required for `pulse`, `gaussian` and `delta`.
- `dc` – The DC level of the distribution. Required for `pulse`, `gaussian` and `delta`.

Sets the distribution of tracks for the x-directed channels – the channels that run horizontally.

Most values are from 0 to 1.

If uniform is specified, you simply specify one argument, peak. This value (by convention between 0 and 1) sets the width of the x-directed core channels relative to the y-directed channels and the channels between the pads and core. **Fig. 3.14** should clarify the specification of uniform (dashed line) and pulse (solid line) channel widths. The gaussian keyword takes the same four parameters as the pulse keyword, and they are all interpreted in exactly the same manner except that in the gaussian case width is the standard deviation of the function.

![Fig. 3.14: Channel Distribution](image)

Chapter 3. FPGA Architecture Description
The delta function is used to specify a channel width distribution in which all the channels have the same width except one. The syntax is `chan_width_x delta peak xpeak dc`. Peak is the extra width of the single wide channel. Xpeak is between 0 and 1 and specifies the location within the FPGA of the extra-wide channel – it is the fractional distance across the FPGA at which this extra-wide channel lies. Finally, dc specifies the width of all the other channels. For example, the statement `chan_width_x delta 3 0.5 1` specifies that the horizontal channel in the middle of the FPGA is four times as wide as the other channels.

Examples:

```xml
<x distr="uniform" peak="1"/>
<x distr="gaussian" width="0.5" peak="0.8" xpeak="0.6" dc="0.2"/>
```

```xml
<y distr="{gaussian|uniform|pulse|delta}" peak=" float" width=" float" xpeak=" float" dc=" float"/>
```

Sets the distribution of tracks for the y-directed channels.

See also:

<x distr>

### 3.1.7 Physical Tiles

The content within the `<tiles>` describes the physical tiles available in the FPGA. Each tile type is specified with the `<tile>` tag within the `<tiles>` tag.

**Tile**

```xml
<tile name="string" capacity="int" width="int" height="int" area="float"/>
```

A tile refers to a placeable element within an FPGA architecture and describes its physical compositions on the grid. The following attributes are applicable to each tile. The only required one is the name of the tile.

**Attributes:**

**Required Attributes**

- **name** – The name of this tile.
  The name must be unique with respect to any other sibling `<tile>` tag.

**Optional Attributes**

- **width** – The width of the block type in grid tiles
  - Default: 1

- **height** – The height of the block type in grid tiles
  - Default: 1

- **area** – The logic area (in MWTA) of the block type
  - Default: from the `<area>` tag

The following tags are common to all `<tile>` tags:

```xml
<sub_tile name="string" capacity="{int}"/>
```

See also:

For a tutorial on describing the usage of sub tiles for heterogeneous tiles (tiles which support multiple instances of the same or different **Complex Blocks**) definition see **Heterogeneous tiles tutorial**.
Describes one or many sub tiles corresponding to the physical tile. Each sub tile identifies a set of one or more stack location on a specific x, y grid location.

Attributes:

**Required Attributes**

- **name** – The name of this tile.

  The name must be unique with respect to any other sibling `<tile>` tag.

**Optional Attributes**

- **capacity** – The number of instances of this block type at each grid location.

  Default: 1

For example:

```xml
<sub_tile name="IO" capacity="2"/>
...</sub_tile>
```

specifies there are two instances of the block type IO at each of its grid locations.

**Note:** It is mandatory to have at least one sub tile definition for each physical tile.

```xml
<input name="string" num_pins="int" equivalent="{none|full}" is_non_clock_global="{true|false}"/>
```

Defines an input port. Multiple input ports are described using multiple `<input>` tags.

**Required Attributes**

- **name** – Name of the input port.
- **num_pins** – Number of pins the input port has.

**Optional Attributes**

- **equivalent** – Describes if the pins of the port are logically equivalent. Input logical equivalence means that the pin order can be swapped without changing functionality. For example, an AND gate has logically equivalent inputs because you can swap the order of the inputs and it's still correct; an adder, on the other hand, is not logically equivalent because if you swap the MSB with the LSB, the results are completely wrong. LUTs are also considered logically equivalent since the logic function (LUT mask) can be rotated to account for pin swapping.
  - **none**: No input pins are logically equivalent.
    - Input pins can not be swapped by the router. (Generates a unique SINK rr-node for each block input port pin.)
  - **full**: All input pins are considered logically equivalent (e.g. due to logical equivalence or a full-crossbar within the cluster).
    - All input pins can be swapped without limitation by the router. (Generates a single SINK rr-node shared by each input port pin.)
  - **default**: none

- **is_non_clock_global** –
Note: Applies only to top-level pb_type.

Describes if this input pin is a global signal that is not a clock. Very useful for signals such as FPGA-wide asynchronous resets. These signals have their own dedicated routing channels and so should not use the general interconnect fabric on the FPGA.

<output name="string" num_pins="int" equivalent="{none|full|instance}"/>

Defines an output port. Multiple output ports are described using multiple <output> tags

Required Attributes

- **name** – Name of the output port.
- **num_pins** – Number of pins the output port has.

Optional Attributes

- **equivalent** – Describes if the pins of the output port are logically equivalent:
  - none: No output pins are logically equivalent.
    Output pins can not be swapped by the router. (Generates a unique SRC rr-node for each block output port pin.)
  - full: All output pins are considered logically equivalent.
    All output pins can be swapped without limitation by the router. For example, this option would be appropriate to model an output port which has a full crossbar between it and the logic within the block that drives it. (Generates a single SRC rr-node shared by each output port pin.)
  - instance: Models that sub-instances within a block (e.g. LUTs/BLFs) can be swapped to achieve a limited form of output pin logical equivalence.
    Like full, this generates a single SRC rr-node shared by each output port pin. However, each net originating from this source can use only one output pin from the equivalence group. This can be useful in modeling more complex forms of equivalence in which you can swap which BLE implements which function to gain access to different inputs.

Warning: When using instance equivalence you must be careful to ensure output swapping would not make the cluster internal routing (previously computed by the clusterer) illegal; the tool does not update the cluster internal routing due to output pin swapping.

Default: none

<clock name="string" num_pins="int" equivalent="{none|full}"/>

Describes a clock port. Multiple clock ports are described using multiple <clock> tags. See above descriptions on inputs

<equivalent_sites>

See also:

For a step-by-step walkthrough on describing equivalent sites see Equivalent Sites tutorial.

Describes the Complex Blocks that can be placed within a tile. Each physical tile can comprehend a number from 1 to N of possible Complex Blocks, or sites. A site corresponds to a top-level Complex Block that must be placeable in at least 1 physical tile locations.
<site pb_type="string" pin_mapping="string"/>

Required Attributes

• **pb_type** – Name of the corresponding pb_type.

Optional Attributes

• **pin_mapping** – Specifies whether the pin mapping between physical tile and logical pb_type:
  
  – **direct**:
    the pin mapping does not need to be specified as the tile pin definition is equal to the corresponding pb_type one;
  
  – **custom**:
    the pin mapping is user-defined.

Default: direct

Example: Equivalent Sites

```xml
<equivalent_sites>
  <site pb_type="MLAB_SITE" pin_mapping="direct"/>
</equivalent_sites>
```

**<direct from="string" to="string">**

Describes the mapping of a physical tile’s port on the logical block’s (pb_type) port. direct is an option sub-tag of site.

**Note:** This tag is needed only if the pin_mapping of the site is defined as custom

**Attributes:**

– from is relative to the physical tile pins
– to is relative to the logical block pins

```xml
<direct from="MLAB_TILE.CX" to="MLAB_SITE.BX"/>
```

**<fc in_type="{frac|abs}" in_val="{int|float}" out_type="{frac|abs}" out_val="{int|float}"**

Required Attributes

• **in_type** – Indicates how the $F_c$ values for input pins should be interpreted.
  
  frac: The fraction of tracks of each wire/segment type.
  
  abs: The absolute number of tracks of each wire/segment type.

• **in_val** – Fraction or absolute number of tracks to which each input pin is connected.

• **out_type** – Indicates how the $F_c$ values for output pins should be interpreted.
  
  frac: The fraction of tracks of each wire/segment type.
  
  abs: The absolute number of tracks of each wire/segment type.

• **out_val** – Fraction or absolute number of wires/segments to which each output pin connects.

Sets the number of tracks/wires to which each logic block pin connects in each channel bordering the pin.

The $F_c$ value [BFRV92] is interpreted as applying to each wire/segment type individually (see example).
When generating the FPGA routing architecture VPR will try to make ‘good’ choices about how pins and wires interconnect; for more details on the criteria and methods used see [BR00].

**Note:** If `<fc>` is not specified for a complex block, the architecture’s `<default_fc>` is used.

**Note:** For unidirectional routing architectures absolute $F_c$ values must be a multiple of 2.

**Example:**

Consider a routing architecture with 200 length 4 (L4) wires and 50 length 16 (L16) wires per channel, and the following $F_c$ specification:

```xml
<fc in_type="frac" in_val="0.1" out_type="abs" out_val="25"></fc>
```

The above specifies that each:

- input pin connects to 20 L4 tracks (10% of the 200 L4s) and 5 L16 tracks (10% of the 50 L16s), and
- output pin connects to 25 L4 tracks and 25 L16 tracks.

**Overriding Values:**

```xml
<fc_override fc_type="{frac|abs}" fc_val="{int|float}", port_name="{string}" segment_name="{string}"
```

Allows $F_c$ values to be overriden on a port or wire/segment type basis.

**Required Attributes**

- **fc_type** – Indicates how the override $F_c$ value should be interpreted.
  - frac: The fraction of tracks of each wire/segment type.
  - abs: The absolute number of tracks of each wire/segment type.
- **fc_val** – Fraction or absolute number of tracks in a channel.

**Optional Attributes**

- **port_name** – The name of the port to which this override applies. If left unspecified this override applies to all ports.
- **segment_name** – The name of the segment (defined under `<segmentlist>`) to which this override applies. If left unspecified this override applies to all segments.

**Note:** At least one of **port_name** or **segment_name** must be specified.

**Port Override Example: Carry Chains**

If you have complex block pins that do not connect to general interconnect (e.g. carry chains), you would use the `<fc_override>` tag, within the `<fc>` tag, to specify them:

```xml
<fc_override fc_type="frac" fc_val="0" port_name="cin"/>
<fc_override fc_type="frac" fc_val="0" port_name="cout"/>
```

Where the attribute **port_name** is the name of the pin (cin and cout in this example).

**Segment Override Example:**
It is also possible to specify per `<segment>` (i.e. routing wire) overrides:

```
<fc_override fc_type="frac" fc_val="0.1" segment_name="L4"/>
```

Where the above would cause all pins (both inputs and outputs) to use a fractional $F_c$ of $0.1$ when connecting to segments of type L4.

**Combined Port and Segment Override Example:**

The `port_name` and `segment_name` attributes can be used together. For example:

```
<fc_override fc_type="frac" fc_val="0.1" port_name="my_input" segment_name="L4"/>
<fc_override fc_type="frac" fc_val="0.2" port_name="my_output" segment_name="L4"/>
```

specifies that port `my_input` use a fractional $F_c$ of $0.1$ when connecting to segments of type L4, while the port `my_output` uses a fractional $F_c$ of $0.2$ when connecting to segments of type L4. All other port/segment combinations would use the default $F_c$ values.

**Required Attributes**

- **pattern**
  - `spread` denotes that the pins are to be spread evenly on all sides of the complex block.
  
  **Note:** Includes internal sides of blocks with width $> 1$ and/or height $> 1$.

  - `perimeter` denotes that the pins are to be spread evenly on perimeter sides of the complex block.
  
  **Note:** Excludes the internal sides of blocks with width $> 1$ and/or height $> 1$.

  - `spread_inputs_perimeter_outputs` denotes that inputs pins are to be spread on all sides of the complex block, but output pins are to be spread only on perimeter sides of the block.
  
  **Note:** This is useful for ensuring outputs do not connect to wires which fly-over a width $> 1$ and height $> 1$ block (e.g. if using short or buffer connections instead of a fully configurable switch block within the block).

- `custom` allows the architect to specify specifically where the pins are to be placed using `<loc>` tags.

Describes the locations where the input, output, and clock pins are distributed in a complex logic block.
Verilog-to-Routing Documentation, Release 8.1.0-dev

Note: ... represents repeat as needed. Do not put ... in the architecture file.

Required Attributes

• **side** – Specifies which of the four sides of a grid location the pins in the contents are located.

Optional Attributes

• **xoffset** – Specifies the horizontal offset (in grid units) from block origin (bottom left corner). The offset value must be less than the width of the block.

  Default: 0

• **yoffset** – Specifies the vertical offset (in grid units) from block origin (bottom left corner). The offset value must be less than the height of the block.

  Default: 0

Physical equivalence for a pin is specified by listing a pin more than once for different locations. For example, a LUT whose output can exit from the top and bottom of a block will have its output pin specified twice: once for the top and once for the bottom.

Note: If the `<pinlocations>` tag is missing, a spread pattern is assumed.

Describes where global routing switchblocks are created in relation to the complex block.

Note: If the `<switchblock_locations>` tag is left unspecified the default pattern is assumed.

Optional Attributes

• **pattern** –
  
  • *external_full_internal_straight*: creates full switchblocks outside and straight switchblocks inside the complex block
  
  • *all*: creates switchblocks wherever routing channels cross
  
  • *external*: creates switchblocks wherever routing channels cross outside the complex block
  
  • *internal*: creates switchblocks wherever routing channels cross inside the complex block
  
  • *none*: denotes that no switchblocks are created for the complex block
  
  • *custom*: allows the architect to specify custom switchblock locations and types using `<sb_loc>` tags

  Default: *external_full_internal_straight*

Optional Attributes
Fig. 3.15: Switchblock Location Patterns for a width = 2, height = 3 complex block
• **internal_switch** – The name of a switch (from `<switchlist>`) which should be used for internal switch blocks.

**Default:** The default switch for the wire `<segment>`

**Note:** This is typically used to specify that internal wire segments are electrically shorted together using a `short` type `<switch>`.

### Example: Electrically Shorted Internal Straight Connections

In some architectures there are no switch blocks located ‘within’ a block, and the wires crossing over the block are instead electrically shorted to their ‘straight-through’ connections.

To model this we first define a special `short` type switch to electrically short such segments together:

```xml
<switchlist>
  <switch type="short" name="electrical_short" R="0" Cin="0" Tdel="0"/>
</switchlist>
```

Next, we use the pre-defined `external_full_internal_straight` pattern, and that such connections should use our `electrical_short` switch.

```xml
<switchblock_locations pattern="external_full_internal_straight" internal_switch="electrical_short"/>
<sb_loc type="{full|straight|turns|none}" xoffset="int" yoffset="int",
switch_override="string">
  Specifies the type of switchblock to create at a particular location relative to a complex block for the custom switchblock location pattern.

  **Required Attributes**
  
  • **type** – Specifies the type of switchblock to be created at this location:
    
    – full: denotes that a full switchblock will be created (i.e. both straight and turns)
    
    – straight: denotes that a switchblock with only straight-through connections will be created (i.e. no turns)
    
    – turns: denotes that a switchblock with only turning connections will be created (i.e. no straight)
    
    – none: denotes that no switchblock will be created

  **Default:** full

  **Optional Attributes**
  
  • **xoffset** – Specifies the horizontal offset (in grid units) from block origin (bottom left corner). The offset value must be less than the width of the block.

    **Default:** 0

  • **yoffset** – Specifies the vertical offset (in grid units) from block origin (bottom left corner). The offset value must be less than the height of the block.

    **Default:** 0

  • **switch_override** – The name of a switch (from `<switchlist>`) which should be used to construct the switch block at this location.
Fig. 3.16: Switchblock Types

**Default:** The default switch for the wire `<segment>`

**Note:** The switchblock associated with a grid tile is located to the top-right of the grid tile.

**Example: Custom Description of Electrically Shorted Internal Straight Connections**

If we assume a width=2, height=3 block (e.g. Fig. 3.15), we can use a custom pattern to specify an architecture equivalent to the 'Electrically Shorted Internal Straight Connections' example:

```xml
<switchblock_locations pattern="custom">
  <!-- Internal: using straight electrical shorts -->
  <sb_loc type="straight" xoffset="0" yoffset="0" switch_override="electrical_short">
  <sb_loc type="straight" xoffset="0" yoffset="1" switch_override="electrical_short">
  <!-- External: using default switches -->
  <sb_loc type="full" xoffset="0" yoffset="2" /> <!-- Top edge -->
  <sb_loc type="full" xoffset="1" yoffset="0" /> <!-- Right edge -->
  <sb_loc type="full" xoffset="1" yoffset="1" /> <!-- Right edge -->
  <sb_loc type="full" xoffset="1" yoffset="2" /> <!-- Top Right -->
<switchblock_locations/>
```

### 3.1.8 Complex Blocks

**See also:**

For a step-by-step walkthrough on building a complex block see *Architecture Modeling*.

The content within the `<complexblocklist>` describes the complex blocks found within the FPGA. Each type of complex block is specified with a top-level `<pb_type>` tag within the `<complexblocklist>` tag.
PB Type

<pb_type name="string" num_pb="int" blif_model="string"/>

Specifies a top-level complex block, or a complex block’s internal components (sub-blocks). Which attributes are applicable depends on where the <pb_type> tag falls within the hierarchy:

• Top Level: A child of the <complexblocklist>
• Intermediate: A child of another <pb_type>
• Primitive/Leaf: Contains no <pb_type> children

For example:

```
<complexblocklist>
  <pb_type name="CLB"/> <!-- Top level -->
  ...
  <pb_type name="ble"/> <!-- Intermediate -->
  ...
  <pb_type name="lut"/> <!-- Primitive -->
  ...
  </pb_type>
  ...
  <pb_type name="ff"/>  <!-- Primitive -->
  ...
  </pb_type>
  ...
</complexblocklist>
```

General:

Required Attributes

• name – The name of this pb_type.
  
The name must be unique with respect to any parent, sibling, or child <pb_type>.

Top-level, Intermediate or Primitive:

Optional Attributes

• num_pb – The number of instances of this pb_type at the current hierarchy level.
  
  Default: 1

For example:

```
<pb_type name="CLB"/>
  ...
  <pb_type name="ble" num_pb="10"/>
  ...
  </pb_type>
  ...
</pb_type>
```

would specify that the pb_type CLB contains 10 instances of the ble pb_type.

Primitive Only:

Required Attributes

• blif_model – Specifies the netlist primitive which can be implemented by this pb_type.
  
  Accepted values:
- .input: A BLIF netlist input
- .output: A BLIF netlist output
- .names: A BLIF .names (LUT) primitive
- .latch: A BLIF .latch (DFF) primitive
- .subckt <custom_type>: A user defined black-box primitive.

For example:

```xml
<pb_type name="my_adder" blif_model=".subckt adder"/>
```

would specify that the pb_type my_adder can implement a black-box BLIF primitive named adder.

**Note:** The input/output/clock ports for primitive pb_types must match the ports specified in the <models> section.

### Optional Attributes

- **class** – Specifies that this primitive is of a specialized type which should be treated specially.
  
  See also: 

  *Classes* for more details.

The following tags are common to all <pb_type> tags:

```xml
<input name="string" num_pins="int" equivalent="{none|full}" is_non_clock_global="{true|false}"/>
```

Defines an input port. Multiple input ports are described using multiple <input> tags.

#### Required Attributes

- **name** – Name of the input port.
- **num_pins** – Number of pins the input port has.

#### Optional Attributes

- **equivalent** –

  **Note:** Applies only to top-level pb_type.

Describe the pins of the port are logically equivalent. Input logical equivalence means that the pin order can be swapped without changing functionality. For example, an AND gate has logically equivalent inputs because you can swap the order of the inputs and it’s still correct; an adder, on the other hand, is not logically equivalent because if you swap the MSB with the LSB, the results are completely wrong. LUTs are also considered logically equivalent since the logic function (LUT mask) can be rotated to account for pin swapping.

- **none**: No input pins are logically equivalent.

  Input pins can not be swapped by the router. (Generates a unique SINK rr-node for each block input port pin.)
- **full**: All input pins are considered logically equivalent (e.g. due to logical equivalence or a full-crossbar within the cluster).

  All input pins can be swapped without limitation by the router. (Generates a single SINK rr-node shared by each input port pin.)

**default**: none

- **is_non_clock_global**

---

**Note**: Applies only to top-level pb_type.

Describes if this input pin is a global signal that is not a clock. Very useful for signals such as FPGA-wide asynchronous resets. These signals have their own dedicated routing channels and so should not use the general interconnect fabric on the FPGA.

```xml
<output name="string" num_pins="int" equivalent="{none|full|instance}"/>
```

Defines an output port. Multiple output ports are described using multiple `<output>` tags

**Required Attributes**

- **name** – Name of the output port.
- **num_pins** – Number of pins the output port has.

**Optional Attributes**

- **equivalent** –

---

**Note**: Applies only to top-level pb_type.

Describes if the pins of the output port are logically equivalent:

- **none**: No output pins are logically equivalent.

  Output pins can not be swapped by the router. (Generates a unique SRC rr-node for each block output port pin.)

- **full**: All output pins are considered logically equivalent.

  All output pins can be swapped without limitation by the router. For example, this option would be appropriate to model an output port which has a full crossbar between it and the logic within the block that drives it. (Generates a single SRC rr-node shared by each output port pin.)

- **instance**: Models that sub-instances within a block (e.g. LUTs/BLEs) can be swapped to achieve a limited form of output pin logical equivalence.

  Like **full**, this generates a single SRC rr-node shared by each output port pin. However, each net originating from this source can use only one output pin from the equivalence group. This can be useful in modeling more complex forms of equivalence in which you can swap which BLE implements which function to gain access to different inputs.

**Warning**: When using **instance** equivalence you must be careful to ensure output swapping would not make the cluster internal routing (previously computed by the clusterer) illegal; the tool does not update the cluster internal routing due to output pin swapping.
Default: none

```xml
<clock name="string" num_pins="int" equivalent="\{none|full\}"/>
```

Describes a clock port. Multiple clock ports are described using multiple `<clock>` tags. See above descriptions on inputs.

```xml
<mode name="string" disable_packing="bool">
```

Required Attributes

- **name** – Name for this mode. Must be unique compared to other modes.

Specifies a mode of operation for the `<pb_type>`. Each child mode tag denotes a different mode of operation for the `<pb_type>`. Each mode tag may contains other `<pb_type>` and `<interconnect>` tags.

**Note:** Modes within the same parent `<pb_type>` are mutually exclusive.

**Note:** If a `<pb_type>` has only one mode of operation the mode tag can be omitted.

Optional Attributes

- **disable_packing** – Specify if a mode is disabled or not for VPR packer. When a mode is defined to be disabled for packing (disable_packing="true"), packer will not map any logic to the mode. This optional syntax aims to help debugging of multi-mode `<pb_type>` so that users can spot bugs in their XML definition quickly. By default, it is set to `false`.

**Note:** When a mode is specified to be disabled for packing, its child `<pb_type>` and the `<mode>` of child `<pb_type>` will be considered as disabled for packing automatically. There is no need to specify disable_packing for every `<mode>` in the tree of `<pb_type>`.

**Warning:** This is a power-user debugging option. See Multi-mode Logic Block Tutorial for a detailed how-to-use.

For example:

```xml
<!-- A fracturable 6-input LUT -->
<pb_type name="lut">
  ...
  <mode name="lut6">
    <!-- Can be used as a single 6-LUT -->
    <pb_type name="lut6" num_pb="1">
      ...
    </pb_type>
    ...
  </mode>
  ...
  <mode name="lut5x2">
    <!-- Or as two 5-LUTs -->
    <pb_type name="lut5" num_pb="2">
      ...
    </pb_type>
    ...
  </mode>
  ...
</pb_type>
```

(continues on next page)
specifies the lut pb_type can be used as either a single 6-input LUT, or as two 5-input LUTs (but not both).

Interconnect

As mentioned earlier, the mode tag contains <pb_type> tags and an <interconnect> tag. The following describes the tags that are accepted in the <interconnect> tag.

<complete name="string" input="string" output="string"/>

Required Attributes

• name – Identifier for the interconnect.
• input – Pins that are inputs to this interconnect.
• output – Pins that are outputs of this interconnect.

Describes a fully connected crossbar. Any pin in the inputs can connect to any pin at the output.

Example:

<complete input="Top.in" output="Child.in"/>

<direct name="string" input="string" output="string"/>

Required Attributes

• name – Identifier for the interconnect.
• input – Pins that are inputs to this interconnect.
• output – Pins that are outputs of this interconnect.

Describes a 1-to-1 mapping between input pins and output pins.

Example:

<direct input="Top.in[2:1]" output="Child[1].in"/>

<mux name="string" input="string" output="string"/>

Required Attributes

• name – Identifier for the interconnect.
• input – Pins that are inputs to this interconnect. Different data lines are separated by a space.
• output – Pins that are outputs of this interconnect.

Describes a bus-based multiplexer.

Note: Buses are not yet supported so all muxes must use one bit wide data only!

Example:
Fig. 3.17: Complete interconnect example.
A `<complete>`, `<direct>`, or `<mux>` tag may take an additional, optional, tag called `<pack_pattern>` that is used to describe *molecules*. A pack pattern is a power user feature directing that the CAD tool should group certain netlist atoms (e.g. LUTs, FFs, carry chains) together during the CAD flow. This allows the architect to help the CAD tool recognize structures that have limited flexibility so that netlist atoms that fit those structures be kept together as though they are one unit. This tag impacts the CAD tool only, there is no architectural impact from defining molecules.

```
<mux input="Top.A Top.B" output="Child.in"/>
```

**Warning:** This is a power user option. Unless you know why you need it, you probably shouldn’t specify it.

### Required Attributes

- **name** – The name of the pattern.
- **in_port** – The input pins of the edges for this pattern.
• **out_port** – Which output pins of the edges for this pattern.

This tag gives a hint to the CAD tool that certain architectural structures should stay together during packing. The tag labels interconnect edges with a pack pattern name. All primitives connected by the same pack pattern name becomes a single pack pattern. Any group of atoms in the user netlist that matches a pack pattern are grouped together by VPR to form a molecule. Molecules are kept together as one unit in VPR. This is useful because it allows the architect to help the CAD tool assign atoms to complex logic blocks that have interconnect with very limited flexibility. Examples of architectural structures where pack patterns are appropriate include: optionally registered inputs/outputs, carry chains, multiply-add blocks, etc.

There is a priority order when VPR groups molecules. Pack patterns with more primitives take priority over pack patterns with less primitives. In the event that the number of primitives is the same, the pack pattern with less inputs takes priority over pack patterns with more inputs.

**Special Case:**

To specify carry chains, we use a special case of a pack pattern. If a pack pattern has exactly one connection to a logic block input pin and exactly one connection to a logic block output pin, then that pack pattern takes on special properties. The prepacker will assume that this pack pattern represents a structure that spans multiple logic blocks using the logic block input/output pins as connection points. For example, let's assume that a logic block has two, 1-bit adders with a carry chain that links adjacent logic blocks. The architect would specify those two adders as a pack pattern with links to the logic block cin and cout pins. Let's assume the netlist has a group of 1-bit adder atoms chained together to form a 5-bit adder. VPR will break that 5-bit adder into 3 molecules: two 2-bit adders and one 1-bit adder connected in order by a the carry links.

**Example:**

Consider a classic basic logic element (BLE) that consists of a LUT with an optionally registered flip-flop. If a LUT is followed by a flip-flop in the netlist, the architect would want the flip-flop to be packed with the LUT in the same BLE in VPR. To give VPR a hint that these blocks should be connected together, the architect would label the interconnect connecting the LUT and flip-flop pair as a pack_pattern:

```xml
<pack_pattern name="ble" in_port="lut.out" out_port="ff.D"/>
```

![Fig. 3.20: Pack Pattern Example.](image)
Classes

Using these structures, we believe that one can describe any digital complex logic block. However, we believe that certain kinds of logic structures are common enough in FPGAs that special shortcuts should be available to make their specification easier. These logic structures are: flip-flops, LUTs, and memories. These structures are described using a class=string attribute in the <pb_type> primitive. The classes we offer are:

**class="lut"**  
Describes a K-input lookup table.  
The unique characteristic of a lookup table is that all inputs to the lookup table are logically equivalent. When this class is used, the input port must have a port_class="lut_in" attribute and the output port must have a port_class="lut_out" attribute.

**class="flipflop"**  
Describes a flipflop.  
Input port must have a port_class="D" attribute added. Output port must have a port_class="Q" attribute added. Clock port must have a port_class="clock" attribute added.

**class="memory"**  
Describes a memory.  
Memories are unique in that a single memory physical primitive can hold multiple, smaller, logical memories as long as:
1. The address, clock, and control inputs are identical and  
2. There exists sufficient physical data pins to satisfy the netlist memories when the different netlist memories are merged together into one physical memory.  
Different types of memories require different attributes.

**Single Port Memories Require:**
- An input port with port_class="address" attribute
- An input port with port_class="data_in" attribute
- An input port with port_class="write_en" attribute
- An output port with port_class="data_out" attribute
- A clock port with port_class="clock" attribute

**Dual Port Memories Require:**
- An input port with port_class="address1" attribute
- An input port with port_class="data_in1" attribute
- An input port with port_class="write_en1" attribute
- An input port with port_class="address2" attribute
- An input port with port_class="data_in2" attribute
- An input port with port_class="write_en2" attribute
- An output port with port_class="data_out1" attribute
- An output port with port_class="data_out2" attribute
- A clock port with port_class="clock" attribute
Timing

See also:
For examples of primitive timing modeling specifications see the Primitive Block Timing Modeling Tutorial

Timing is specified through tags contained within pb_type, complete, direct, or mux tags as follows:

\[
\text{<delay_constant max="float" min="float" in_port="string" out_port="string"/>}
\]

Optional Attributes

- **max** – The maximum delay value.
- **min** – The minimum delay value.

Required Attributes

- **in_port** – The input port name.
- **out_port** – The output port name.

Specifies a maximum and/or minimum delay from in_port to out_port.
- If in_port and out_port are non-sequential (i.e. combinational) inputs specifies the combinational path delay between them.
- If in_port and out_port are sequential (i.e. have T_setup and/or T_clock_to_Q tags) specifies the combinational delay between the primitive's input and/or output registers.

**Note:** At least one of the max or min attributes must be specified

**Note:** If only one of max or min are specified the unspecified value is implicitly set to the same value

\[
\text{<delay_matrix type="{max | min}" in_port="string" out_port="string"> matrix </delay>}
\]

Required Attributes

- **type** – Specifies the delay type.
- **in_port** – The input port name.
- **out_port** – The output port name.
- **matrix** – The delay matrix.

Describe a timing matrix for all edges going from in_port to out_port. Number of rows of matrix should equal the number of inputs, number of columns should equal the number of outputs.
- If in_port and out_port are non-sequential (i.e combinational) inputs specifies the combinational path delay between them.
- If in_port and out_port are sequential (i.e. have T_setup and/or T_clock_to_Q tags) specifies the combinational delay between the primitive's input and/or output registers.

**Example:** The following defines a delay matrix for a 4-bit input port in, and 3-bit output port out:

\[
\text{<delay_matrix type="max" in_port="in" out_port="out">}
    \begin{array}{ccc}
    1.2e-10 & 1.4e-10 & 3.2e-10 \\
    4.6e-10 & 1.9e-10 & 2.2e-10 \\
    4.5e-10 & 6.7e-10 & 3.5e-10 \\
    7.1e-10 & 2.9e-10 & 8.7e-10
    \end{array}
\text{ </delay>}
\]
Note: To specify both max and min delays two <delay_matrix> should be used.

<T_setup value="float" port="string" clock="string"/>

Required Attributes

• value – The setup time value.
• port – The port name the setup constraint applies to.
• clock – The port name of the clock the setup constraint is specified relative to.

Specifies a port’s setup constraint.
• If port is an input specifies the external setup time of the primitive’s input register (i.e. for paths terminating at the input register).
• If port is an output specifies the internal setup time of the primitive’s output register (i.e. for paths terminating at the output register).

Note: Applies only to primitive <pb_type> tags

<T_hold value="float" port="string" clock="string"/>

Required Attributes

• value – The hold time value.
• port – The port name the setup constraint applies to.
• clock – The port name of the clock the setup constraint is specified relative to.

Specifies a port’s hold constraint.
• If port is an input specifies the external hold time of the primitive’s input register (i.e. for paths terminating at the input register).
• If port is an output specifies the internal hold time of the primitive’s output register (i.e. for paths terminating at the output register).

Note: Applies only to primitive <pb_type> tags

<T_clock_to_Q max="float" min="float" port="string" clock="string"/>

Optional Attributes

• max – The maximum clock-to-Q delay value.
• min – The minimum clock-to-Q delay value.

Required Attributes

• port – The port name the delay value applies to.
• clock – The port name of the clock the clock-to-Q delay is specified relative to.

Specifies a port’s clock-to-Q delay.
• If port is an input specifies the internal clock-to-Q delay of the primitive’s input register (i.e. for paths starting at the input register).
• If port is an output specifies the external clock-to-Q delay of the primitive’s output register (i.e. for paths starting at the output register).

Note: At least one of the max or min attributes must be specified
Note: If only one of max or min are specified the unspecified value is implicitly set to the same value

Note: Applies only to primitive <pb_type> tags

Modeling Sequential Primitive Internal Timing Paths

See also:

For examples of primitive timing modeling specifications see the *Primitive Block Timing Modeling Tutorial*

By default, if only <T_setup> and <T_clock_to_Q> are specified on a primitive pb_type no internal timing paths are modeled. However, such paths can be modeled by using <delay_constant> and/or <delay_matrix> can be used in conjunction with <T_setup> and <T_clock_to_Q>. This is useful for modeling the speed-limiting path of an FPGA hard block like a RAM or DSP.

As an example, consider a sequential black-box primitive named seq.foo which has an input port in, output port out, and clock clk:

```xml
<pb_type name="seq.foo" blif_model=".subckt seq.foo" num_pb="1">
  <input name="in" num_pins="4"/>
  <output name="out" num_pins="1"/>
  <clock name="clk" num_pins="1"/>

  <!-- external -->
  <T_setup value="80e-12" port="seq.foo.in" clock="clk"/>
  <T_clock_to_Q max="20e-12" port="seq.foo.out" clock="clk"/>

  <!-- internal -->
  <T_clock_to_Q max="10e-12" port="seq.foo.in" clock="clk"/>
  <delay_constant max="0.9e-9" in_port="seq.foo.in" out_port="seq.foo.out"/>
  <T_setup value="90e-12" port="seq.foo.out" clock="clk"/>
</pb_type>
```

To model an internal critical path delay, we specify the internal clock-to-Q delay of the input register (10ps), the internal combinational delay (0.9ns) and the output register’s setup time (90ps). The sum of these delays corresponds to a 1ns critical path delay.

Note: Primitive timing paths with only one stage of registers can be modeled by specifying <T_setup> and <T_clock_to_Q> on only one of the ports.
Power

See also:

Power Estimation, for the complete list of options, their descriptions, and required sub-fields.

<power method="string">contents</power

Optional Attributes

• method – Indicates the method of power estimation used for the given pb_type.
  Must be one of:
  – specify-size
  – auto-size
  – pin-toggle
  – C-internal
  – absolute
  – ignore
  – sum-of-children
  Default: auto-size.

See also:

Power Architecture Modelling for a detailed description of the various power estimation
methods.

The contents of the tag can consist of the following tags:
• <dynamic_power>
• <static_power>
• <pin>

<dynamc_power power_per_instance="float" C_internal="float"/>

Optional Attributes

• power_per_instance – Absolute power in Watts.
• C_internal – Block capacitance in Farads.

<static_power power_per_instance="float"/>

Optional Attributes

• power_per_instance – Absolute power in Watts.

<port name="string" energy_per_toggle="float" scaled_by_static_prob="string" scaled_by_static_prob_n="string"/>

Required Attributes

• name – Name of the port.
• energy_per_toggle – Energy consumed by a toggle on the port specified in name.

Optional Attributes

• scaled_by_static_prob – Port name by which to scale energy_per_toggle based on its logic high probability.
• scaled_by_static_prob_n – Port name by which to scale energy_per_toggle based on its logic low probability.

3.1.9 Wire Segments

The content within the <segmentlist> tag consists of a group of <segment> tags. The <segment> tag and its contents are described below.

```
<segment name="unique_name" length="int" type="{bidir|unidir}" freq="float" Rmetal="float" Cmetal="float">content</segment>
```

Required Attributes

• name – A unique alphanumeric name to identify this segment type.

• length – Either the number of logic blocks spanned by each segment, or the keyword longline. Longline means segments of this type span the entire FPGA array.

Note: longline is only supported on with bidir routing

• freq – The supply of routing tracks composed of this type of segment. VPR automatically determines the percentage of tracks for each segment type by taking the frequency for the type specified and dividing with the sum of all frequencies. It is recommended that the sum of all segment frequencies be in the range 1 to 100.

• Rmetal – Resistance per unit length (in terms of logic blocks) of this wiring track, in Ohms. For example, a segment of length 5 with Rmetal = 10 Ohms / logic block would have an end-to-end resistance of 50 Ohms.

• Cmetal – Capacitance per unit length (in terms of logic blocks) of this wiring track, in Farads. For example, a segment of length 5 with Cmetal = 2e-14 F / logic block would have a total metal capacitance of 10e-13F.

• directionality – This is either unidirectional or bidirectional and indicates whether a segment has multiple drive points (bidirectional), or a single driver at one end of the wire segment (unidirectional). All segments must have the same directionality value. See [LLTY04] for a description of unidirectional single-driver wire segments.

• content – The switch names and the depopulation pattern as described below.

```
<sb type="pattern">int list</sb>
```

This tag describes the switch block depopulation (as illustrated in Fig. 3.21) for this particular wire segment. For example, the first length 6 wire in the figure below has an sb pattern of 1 0 1 0 1 0 1. The second wire has a pattern of 0 1 0 1 0 1 0. A 1 indicates the existence of a switch block and a 0 indicates that there is no switch box at that point. Note that there are 7 entries in the integer list for a length 6 wire. For a length L wire there must be L+1 entries separated by spaces.

Fig. 3.21: Switch block and connection block pattern example with four tracks per channel
Note: Can not be specified for longline segments (which assume full switch block population)

<cb type="pattern">int list</cb>
This tag describes the connection block depopulation (as illustrated by the circles in Fig. 3.21) for this particular wire segment. For example, the first length 6 wire in the figure below has an sb pattern of 1 1 1 1 1 1. The third wire has a pattern of 1 0 0 1 1 0. A 1 indicates the existence of a connection block and a 0 indicates that there is no connection box at that point. Note that there a 6 entries in the integer list for a length 6 wire. For a length L wire there must be L entries separated by spaces.

Note: Can not be specified for longline segments (which assume full connection block population)

<mux name="string"/>
Required Attributes

• name – Name of the mux switch type used to drive this type of segment.

Note: For UNIDIRECTIONAL only.

Tag must be included and name must be the same as the name you give in <switch type="mux" name="..."

<wire_switch name="string"/>
Required Attributes

• name – Name of the switch type used by other wires to drive this type of segment.

Note: For BIDIRECTIONAL only.

Tag must be included and the name must be the same as the name you give in <switch type="tristate|pass_gate" name="..." for the switch which represents the wire switch in your architecture.

<opin_switch name="string"/>
Required Attributes

• name – Name of the switch type used by block pins to drive this type of segment.

Note: For BIDIRECTIONAL only.

Tag must be included and name must be the same as the name you give in <switch type="tristate|pass_gate" name="..." for the switch which represents the output pin switch in your architecture.

Note: In unidirectional segment mode, there is only a single buffer on the segment. Its type is specified by assigning the same switch index to both wire_switch and opin_switch. VPR will error out if these two are not the same.

Note: The switch used in unidirectional segment mode must be buffered.
3.1.10 Clocks

There are two options for describing clocks. One method allows you to specify clocking purely for power estimation, see *Specifying Clocking Purely for Power Estimation*. The other method allows you to specify a clock architecture that is used as part of the routing resources, see *Specifying a Clock Architecture*. Both methods should not be used in tandem.

**Specifying Clocking Purely for Power Estimation**

The clocking configuration is specified with `<clock>` tags within the `<clocks>` section.

**Note:** Currently the information in the `<clocks>` section is only used for power estimation.

See also:

*Power Estimation* for more details.

```
<clock C_wire="float" C_wire_per_m="float" buffer_size="float"|"auto"/>
```

**Optional Attributes**

- `C_wire` – The absolute capacitance, in Farads, of the wire between each clock buffer.
- `C_wire_per_m` – The wire capacitance, in Farads per Meter.
- `buffer_size` – The size of each clock buffer.

**Specifying a Clock Architecture**

The element `<clocknetworks>` contains three sub-elements that collectively describe the clock architecture: the wiring parameters `<metal_layers>`, the clock distribution `<clock_network>`, and the clock connectivity `<clock_routing>`.

**Clock Architecture Example**

The following example shows how a rib-spine (row/column) style clock architecture can be defined.

```
<clocknetworks>
  <metal_layers >
    <metal_layer name="global_spine" Rmetal="50.42" Cmetal="20.7e-15"/>
    <metal_layer name="global_rib" Rmetal="50.42" Cmetal="20.7e-15"/>
  </metal_layers >

  <!-- Full Device: Center Spine -->
  <clock_network name="spine1" num_inst="2">
    <spine metal_layer="global_spine" x="W/2" starty="0" endy="H">
      <switch_point type="drive" name="drive_point" yoffset="H/2" buffer="drive_buff"/>
      <switch_point type="tap" name="taps" yoffset="0" yincr="1"/>
    </spine>
  </clock_network>

  <!-- Full Device: Each Grid Row -->
  <clock_network name="rib1" num_inst="2">
```

(continues on next page)
<rib metal_layer="global_rib" y="0" startx="0" endx="W" repeatx="W" repeaty="1">
  <switch_point type="drive" name="drive_point" xoffset="W/2" buffer="drive_buff"/>
  <switch_point type="tap" name="taps" xoffset="0" xincr="1"/>
</rib>
</clock_network>

<clock_routing>
  <!-- connections from inter-block routing to central spine -->
  <tap from="ROUTING" to="spine1.drive_point" locationx="W/2" locationy="H/2">
    --switch="general_routing_switch" fc_val="1.0"/>
  </tap>
  <!-- connections from spine to rib -->
  <tap from="spine1.taps" to="rib1.drive_point" switch="general_routing_switch" fc_val="0.5"/>
  <!-- connections from rib to clock pins -->
  <tap from="rib1.taps" to="CLOCK" switch="ipin_cblock" fc_val="1.0"/>
</clock_routing>
</clocknetworks>

Fig. 3.22: <spine> “spine1” vertical clock wire example. The two spines (num_inst="2") are located horizontally at W/2 (in the middle of the device), and spans the entire height of the device (0..H). The drive points are located at H/2, with tap points located at unit increments along their length. Buffers of drive_buff type (would be defined in <switches>) are used to drive the two halves of the spines.
Fig. 3.23: `<rib>` “rib1” horizontal clock wire example. Each rib spans the full width of the device (0..W), with the drive points located at the mid-point (W/2), and tap points in unit increments along each rib. There are two ribs at each vertical location (num_inst="2"), and pairs of ribs are stamped out at each row of the device (repeaty="1").
Clock Architecture Tags

The <metal_layers> element describes the per unit length electrical parameters, resistance (Rmetal) and capacitance (Cmetal), used to implement the clock distribution wires. Wires are modeled solely based on Rmetal and Cmetal parameters which are derived from the physical implementation of the metal layer width and spacing. There can be one or more wiring implementation options (metal layer, width and spacing) that are used by the later clock network specification and each is described in a separate <metal_layer> sub-element. The syntax of the wiring electrical information is:

<metal_layer name="string" Rmetal="float" Cmetal="float"/>

Required Attributes

• name – A unique string for reference.
• Rmetal – The resistance in Ohms of the wire per unit block in the FPGA architecture; a unit block usually corresponds to a logic cluster.
• Cmetal – The capacitance in Farads of the wire per unit block.

The <clock_network> element contains sub-elements that describe the clock distribution wires for the clock architecture. There could be more than one <clock_network> element to describe separate types of distribution wires. The high-level start tag for a clock network is as follows:

<clock_network name="string" num_inst="integer">

Required Attributes

• name – A unique string for reference.
• num_inst – which describes the number of parallel instances of the clock distribution types described in the <clock_network> sub-elements.

Note: Many parameters used in the following clock architecture tags take an expression (expr) as an argument similar to Grid Location Expressions. However, only a subset of special variables are supported: W (device width) and H (device height).

The supported clock distribution types are <spine> and <rib>. Spines are used to describe vertical clock distribution wires. Whereas, Ribs is used to describe a horizontal clock distribution wire. See Clock Architecture Example and accompanying figures Fig. 3.22 and Fig. 3.23 for example use of <spine> and <rib> parameters.

<spine metal_layer="string" x="expr" starty="expr" endy="expr" repeatx="expr" repeaty="expr"/>

Required Attributes

• metal_layer – A referenced metal layer that sets the unit resistance and capacitance of the distribution wire over the length of the wire.
• starty – The start y grid location, of the wire which runs parallel to the y-axis from starty and ends at endy, inclusive. Value can be relative to the device size.
• endy – The end of y grid location of the wire. Value can be relative to the device size.
• x – The location of the spine with respect to the x-axis. Value can be relative to the device size.

Optional Attributes
• **repeatx** – The horizontal repeat factor of the spine along the device. Value can be relative to the device size.

• **repeaty** – The vertical repeat factor of the spine along the device. Value can be relative to the device size.

The provided example clock network (*Clock Architecture Example*) defines two spines, and neither repeats as each spans the entire height of the device and is locally at the horizontal midpoint of the device.

```xml
<rib metal_layer="string" y="expr" startx="expr" endx="expr" repeatx="expr" repeaty="expr"/>
```

Required Attributes

• **metal_layer** – A referenced metal layer that sets the unit resistance and capacitance of the distribution wire over the length of the wire.

• **startx** – The start x grid location, of the wire which runs parallel to the x-axis from startx and ends at endx, inclusive. Value can be relative to the device size.

• **endx** – The end of x grid location of the wire. Value can be relative to the device size.

• **y** – The location of the rib with respect to the y-axis. Value can be relative to the device size.

Optional Attributes

• **repeatx** – The horizontal repeat factor of the rib along the device. Value can be relative to the device size.

• **repeaty** – The vertical repeat factor of the rib along the device. Value can be relative to the device size.

Along each spine and rib is a group of switch points. Switch points are used to describe drive or tap locations along the clock distribution wire, and are enclosed in the relevant `<rib>` or `<spine>` tags:

```xml
<switch_point type="{drive | tap}" name="string" yoffset="expr" xoffset="expr" xinc="expr" yinc="expr" buffer="string"/>
```

Required Attributes

• **type** –
  - **drive** – Drive points are where the clock distribution wire can be driven by a routing switch or buffer.
  - **tap** – Tap points are where it can drive a routing switch or buffer to send a signal to a different clock network or logic block.

• **buffer** – (Required only for drive points) A reference to a pre-defined routing switch; specified by `<switch>` tag, see Section *Switches*. This switch will be used at the drive point. The clock architecture generator uses two of these buffers to drive the two portions of this clock network wire when it is split at the drive point, see Figures Fig. 3.23 and Fig. 3.22.

Optional Attributes

• **xoffset** – (Only for rib network) Offset from the startx of a rib network.

• **yoffset** – (Only for spine network) Offset from the starty of a spine network.

• **xinc** – (Only for rib tap points) Describes the repeat factor of a series of evenly spaced tap points.
• **yinc** – (Only for spine tap points) Describes the repeat factor of a series of evenly spaced tap points.

**Note:** A single `<switch_point>` specification may define a *set* of tap points (type="tap", with either xincr or yincr), or a single drive point (type="drive").

Lastly the `<clock_routing>` element consists of a group of tap statements which separately describe the connectivity between clock-related routing resources (pin or wire). The tap element and its attributes are as follows:

```xml
<tap from="string" to="string" locationx="expr" locationy="expr" switch="string" fc_val="float">

**Required Attributes**

- **from** – The set of routing resources to make connections *from*. This can be either:
  - `clock_name.tap_points_name`: A set of clock network tap-type switchpoints. The format is clock network name, followed by the tap points name and delineated by a period (e.g. `spine1.taps`), or
  - `ROUTING`: a special literal which references a connection from general inter-block routing (at a location specified by `locationx` and `locationy` parameters).

Examples can be see in *Clock Architecture Example*.

- **to** – The set of routing resources to make connections *to*. Can be a unique name or special literal:
  - `clock_name.drive_point_name`: A clock network drive-type switchpoint. The format is clock network name, followed by the drive point name and delineated by a period (e.g. `rib1.drive_point`).
  - `CLOCK`: a special literal which describes connections from clock network tap points that supply the clock to clock pins on blocks at the tap locations; these are clock inputs are already specified on blocks (top-level `<pb_type>`/`<tile>`) in the VTR architecture file.

Examples can be see in *Clock Architecture Example*.

- **switch** – The routing switch (defined in `<switches>`) used for this connection.

- **fc_val** – A decimal value between 0 and 1 representing the connection block flexibility between the connecting routing resources; a value of 0.5 for example means that only 50% of the switches necessary to connect all the matching tap and drive points would be implemented.

**Optional Attributes**

- **locationx** – (Required when using the special literal "ROUTING") The x grid location of inter-block routing.

- **locationy** – (Required when using the special literal "ROUTING") The y grid location of inter-block routing.

**Note:** A single `<tap>` statement may create multiple connections if either the of the `from` or `to` correspond to multiple routing resources. In such cases the `fc_val` can control how many connections are created.
Note: locationx and locationy describe an (x,y) grid location where all the wires passing this location source the source the clock network connection depending on the fc_val

For more information you may wish to consult [Abb19] which introduces the clock modeling language.

### 3.1.11 Power

Additional power options are specified within the <architecture> level <power> section.

See also:
See Power Estimation for full documentation on how to perform power estimation.

```xml
<local_interconnect C_wire="float" factor="float"/>
```

**Required Attributes**
- **C_wire** – The local interconnect capacitance in Farads/Meter.

**Optional Attributes**
- **factor** – The local interconnect scaling factor. Default: 0.5.

```xml
<buffers logical_effort_factor="float"/>
```

**Required Attributes**
- **logical_effort_factor** – Default: 4.

### 3.1.12 Direct Inter-block Connections

The content within the <directlist> tag consists of a group of <direct> tags. The <direct> tag and its contents are described below.

```xml
<direct name="string" from_pin="string" to_pin="string" x_offset="int" y_offset="int" z_offset="int" switch_name="string"/>
```

**Required Attributes**
- **name** – is a unique alphanumeric string to name the connection.
- **from_pin** – pin of complex block that drives the connection.
- **to_pin** – pin of complex block that receives the connection.
- **x_offset** – The x location of the receiving CLB relative to the driving CLB.
- **y_offset** – The y location of the receiving CLB relative to the driving CLB.
- **z_offset** – The z location of the receiving CLB relative to the driving CLB.

**Optional Attributes**
- **switch_name** – [Optional, defaults to delay-less switch if not specified] The name of the <switch> from <switchlist> to be used for this direct connection.
- **from_side** – The associated from_pin's block size (must be one of left, right, top, bottom or left unspecified)
- **to_side** – The associated to_pin's block size (must be one of left, right, top, bottom or left unspecified)
Describes a dedicated connection between two complex block pins that skips general interconnect. This is useful for describing structures such as carry chains as well as adjacent neighbour connections.

The `from_side` and `to_side` options can usually be left unspecified. However they can be used to explicitly control how directs to physically equivalent pins (which may appear on multiple sides) are handled.

**Example:** Consider a carry chain where the `cout` of each CLB drives the `cin` of the CLB immediately below it, using the delay-less switch one would enter the following:

```xml
<direct name="adder_carry" from_pin="clb.cout" to_pin="clb.cin" x_offset="0" y_offset="-1" z_offset="0"/>
```

### 3.1.13 Custom Switch Blocks

The content under the `<switchblocklist>` tag consists of one or more `<switchblock>` tags that are used to specify connections between different segment types. An example is shown below:

```xml
<switchblocklist>
  <switchblock name="my_switchblock" type="unidir">
    <switchblock_location type="EVERYWHERE"/>
    <switchfuncs>
      <func type="lr" formula="t"/>
      <func type="lt" formula="W-t"/>
      <func type="lb" formula="W+t-1"/>
      <func type="rt" formula="W+t-1"/>
      <func type="br" formula="W-t-2"/>
      <func type="bt" formula="t"/>
      <func type="rl" formula="t"/>
      <func type="tl" formula="W-t"/>
      <func type="bl" formula="W+t-1"/>
      <func type="tr" formula="W+t-1"/>
      <func type="rb" formula="W-t-2"/>
      <func type="tb" formula="t"/>
    </switchfuncs>
    <wireconn from_type="l4" to_type="l4" from_switchpoint="0,1,2,3" to_switchpoint="0"/>
    <wireconn from_type="l8_global" to_type="l8_global" from_switchpoint="0,4" to_switchpoint="0"/>
    <wireconn from_type="l8_global" to_type="l4" from_switchpoint="0,4" to_switchpoint="0"/>
  </switchblock>
  <switchblock name="another_switch_block" type="unidir">
    ... another switch block description ...
  </switchblock>
</switchblocklist>
```

This switch block format allows a user to specify mathematical permutation functions that describe how different types of segments (defined in the architecture file under the `<segmentlist>` tag) will connect to each other at different switch points. The concept of a switch point is illustrated below for a length-4 unidirectional wire heading in the “left” direction. The switch point at the start of the wire is given an index of 0 and is incremented by 1 at each subsequent switch block until the last switch point. The last switch point has an index of 0 because it is shared between the end of the current segment and the start of the next one (similarly to how switch point 3 is shared by the two wire subsegments on each side).
A collection of wire types and switch points defines a set of wires which will be connected to another set of wires with the specified permutation functions (the ‘sets’ of wires are defined using the <wireconn> tags). This format allows for an abstract but very flexible way of specifying different switch block patterns. For additional discussion on interconnect modeling see [Pet16]. The full format is documented below.

**Overall Notes:**

1. The `<sb type="pattern">` tag on a wire segment (described under `<segmentlist>`) is applied as a mask on the patterns created by this switch block format; anywhere along a wire’s length where a switch block has not been requested (set to 0 in this tag), no switches will be added.

2. You can specify multiple switchblock tags, and the switches described by the union of all those switch blocks will be created.

```xml
<switchblock name="string" type="string">
    Required Attributes
    • name – A unique alphanumeric string
    • type – unidir or bidir. A bidirectional switch block will implicitly mirror the specified permutation functions – e.g. if a permutation function of type lr (function used to connect wires from the left to the right side of a switch block) has been specified, a reverse permutation function of type rl (right-to-left) is automatically assumed. A unidirectional switch block makes no such implicit assumptions. The type of switch block must match the directionality of the segments defined under the `<segmentlist>` node.

<switchblock> is the top-level XML node used to describe connections between different segment types.

```xml
<switchblock_location type="string"/>
```

Required Attributes

• type – Can be one of the following strings:
  - EVERYWHERE – at each switch block of the FPGA
  - PERIMETER – at each perimeter switch block (x-directed and/or y-directed channel segments may terminate here)
  - CORNER – only at the corner switch blocks (both x and y-directed channels terminate here)
  - FRINGE – same as PERIMETER but excludes corners
  - CORE – everywhere but the perimeter

Sets the location on the FPGA where the connections described by this switch block will be instantiated.

```xml
<switchfuncs>
The switchfuncs XML node contains one or more entries that specify the permutation functions with which different switch block sides should be connected, as described below.
```
<func type="string" formula="string"/>

**Required Attributes**

- **type** – Specifies which switch block sides this function should connect. With the switch block sides being left, top, right and bottom, the allowed entries are one of {lt, lr, lb, tr, tb, tl, rb, rl, rt, bl, bt, br} where lt means that the specified permutation formula will be used to connect the left-top sides of the switch block.

*Note: In a bidirectional architecture the reverse connection is implicit.*

- **formula** – Specifies the mathematical permutation function that determines the pattern with which the source/destination sets of wires (defined using the <wireconn> entries) at the two switch block sides will be connected. For example, \( W-t \) specifies a connection where the \( t \)'th wire in the source set will connect to the \( W-t \) wire in the destination set where \( W \) is the number of wires in the destination set and the formula is implicitly treated as modulo \( W \).

Special characters that can be used in a formula are:

- \( t \) – the index of a wire in the source set
- \( W \) – the number of wires in the destination set (which is not necessarily the total number of wires in the channel)

The operators that can be used in the formula are:

- Addition (+)
- Subtraction (-)
- Multiplication (*)
- Division (/)
- Brackets ( and ) are allowed and spaces are ignored.

Defined under the `<switchfuncs>` XML node, one or more `<func...>` entries is used to specify permutation functions that connect different sides of a switch block.

<wireconn num_conns="expr" from_type="string, string, string, ..." to_type="string, string, string, ..." from_switchpoint="int, int, int, ..." to_switchpoint="int, int, int, ..." from_order="{fixed | shuffled}" to_order="{fixed | shuffled}"/>

**Required Attributes**

- **num_conns** – Specifies how many connections should be created between the from_type/from_switchpoint set and the to_type/to_switchpoint set. The value of this parameter is an expression which is evaluated when the switch block is constructed.

The expression can be a single number or formula using the variables:

- **from** – The number of switchblock edges equal to the ‘from’ set size.

  This ensures that each element in the ‘from’ set is connected to an element of the ‘to’ set. However it may leave some elements of the ‘to’ set either multiply-connected or disconnected.

- **to** – The number of switchblock edges equal to the ‘to’ set size size.

  This ensures that each element of the ‘to’ set is connected to precisely one element of the ‘from’ set. However it may leave some elements of the ‘from’ set either multiply-connected or disconnected.
Examples:

- `min(from, to)` – Creates number of switchblock edges equal to the minimum of the ‘from’ and ‘to’ set sizes.
  
  This ensures no element of the ‘from’ or ‘to’ sets is connected to multiple elements in the opposing set. However it may leave some elements in the larger set disconnected.

![Diagram](image1)

- `max(from, to)` – Creates number of switchblock edges equal to the maximum of the ‘from’ and ‘to’ set sizes.

  This ensures all elements of the ‘from’ or ‘to’ sets are connected to at least one element in the opposing set. However some elements in the smaller set may be multiply-connected.

![Diagram](image2)

- `3*to` – Creates number of switchblock edges equal to three times the ‘to’ set sizes.
• **from_type** – A comma-separated list segment names that defines which segment types will be a source of a connection. The segment names specified must match the names of the segments defined under the `<segmentlist>` XML node. Required if no `<from>` or `<to>` nodes are specified within the `<wireconn>`.

• **to_type** – A comma-separated list of segment names that defines which segment types will be the destination of the connections specified. Each segment name must match an entry in the `<segmentlist>` XML node. Required if no `<from>` or `<to>` nodes are specified within the `<wireconn>`.

• **from_switchpoint** – A comma-separated list of integers that defines which switchpoints will be a source of a connection. Required if no `<from>` or `<to>` nodes are specified within the `<wireconn>`.

• **to_switchpoint** – A comma-separated list of integers that defines which switchpoints will be the destination of the connections specified. Required if no `<from>` or `<to>` nodes are specified within the `<wireconn>`.

**Note:** In a unidirectional architecture wires can only be driven at their start point so `to_switchpoint="0"` is the only legal specification in this case.

**Optional Attributes**

• **from_order** – Specifies the order in which `from_switchpoint`’s are selected when creating edges.

  – **fixed** – Switchpoints are selected in the order specified
    
    This is useful to specify a preference for connecting to specific switchpoints. For example,

    ```xml
    <wireconn num_conns="1*to" from_type="L16" from_switchpoint="0,12,8,4" from_order="fixed" to_type="L4" to_switchpoint="0"/>
    ```

    specifies L4 wires should be connected first to L16 at switchpoint 0, then at switchpoints 12, 8, and 4. This is primarily useful when we want to ensure that some switchpoints are ‘used-up’ first.

  – **shuffled** – Switchpoints are selected in a (randomly) shuffled order
    
    This is useful to ensure a diverse set of switchpoints are used. For example,

    ```xml
    <wireconn num_conns="1*to" from_type="L4" from_switchpoint="0,1,2,3" from_order="shuffled" to_type="L4" to_switchpoint="0"/>
    ```

    specifies L4 wires should be connected to other L4 wires at any of switchpoints 0, 1, 2, or 3. Shuffling the switchpoints is useful if one of the sets (e.g. from L4’s) is much larger than the other (e.g. to L4’s), and we wish to ensure a variety of switchpoints from the larger set are used.

**Default:** shuffled

• **to_order** – Specifies the order in which `to_switchpoint`’s are selected when creating edges.
<from type="string" switchpoint="int, int, int, ..."/>

**Required Attributes**

- **type** – The name of a segment specified in the `<segmentlist>`.
- **switchpoint** – A comma-separated list of integers that defines switchpoints.

**Note:** In a unidirectional architecture wires can only be driven at their start point so `to_switchpoint="0"` is the only legal specification in this case.

Specifies a subset of *source* wire switchpoints. This tag can be specified multiple times. The surrounding `<wireconn>`’s source set is the union of all contained `<from>` tags.

<to type="string" switchpoint="int, int, int, ..."/>

Specifies a subset of *destination* wire switchpoints. This tag can be specified multiple times. The surrounding `<wireconn>`’s destination set is the union of all contained `<to>` tags.

**See also:**

- `<from>` for attribute descriptions.

As an example, consider the following `<wireconn>` specification:

```
<wireconn num_conns_type="to"/>
  <from type="L4" switchpoint="0,1,2,3"/>
  <from type="L16" switchpoint="0,4,8,12"/>
  <to type="L4" switchpoint="0"/>
</wireconn>
```

This specifies that the ‘from’ set is the union of L4 switchpoints 0, 1, 2 and 3; and L16 switchpoints 0, 4, 8 and 12. The ‘to’ set is all L4 switchpoint 0’s. Note that since different switchpoints are selected from different segment types it is not possible to specify this without using `<from>` sub-tags.

### 3.1.14 Architecture metadata

Architecture metadata enables tagging of architecture or routing graph information that exists outside of the normal VPR flow (e.g. pack, place, route, etc). For example this could be used to enable bitstream generation by tagging routing edges and pb_type features.

The metadata will not be used by the vpr executable, but can be leveraged by new tools using the libvpr library. These new tools can access the metadata on the various VPR internal data structures.

To enable tagging of architecture structures with metadata, the `<metadata>` tag can be inserted under the following XML tags:

- `<pb_type>`
- Any tag under `<interconnect>` (<direct>, <mux>, etc).
- `<mode>`
• Any grid location type (<perimeter>, <fill>, <corners>, <single>, <col>, <row>, <region>)

<metadata>
Specifies the root of a metadata block. Can have 0 or more <meta> Children.

<meta name="string">
Required Attributes

• name – Key name of this metadata value.

Specifies a value within a metadata block. The name is a key for looking up the value contained within the <meta> tag. Keys can be repeated, and will be stored in a vector in order of occurrence.

The value of the <meta> is the text in the block. Both the name and <meta> value will be stored as a string. XML children are not supported in the <meta> tag.

Example of a metadata block with 2 keys:

```xml
<metadata>
  <meta name="some_key">Some value</meta>
  <meta name="other key!">Other value!</meta>
</metadata>
```

### 3.2 Example Architecture Specification

The listing below is for an FPGA with I/O pads, soft logic blocks (called CLB), configurable memory hard blocks, and fracturable multiplier hard blocks.

Notice that for the CLB, all the inputs are logically equivalent (line 157), and all the outputs are logically equivalent (line 158). This is usually true for cluster-based logic blocks, as the local routing within the block usually provides full (or near full) connectivity.

However, for other logic blocks, the inputs and all the outputs are not logically equivalent. For example, consider the memory block (lines 311-316). Swapping inputs going into the data input port changes the logic of the block because the data output order no longer matches the data input.

```xml
<!-- VPR Architecture Specification File -->
<!-- Quick XML Primer:
* Data is hierarchical and composed of tags (similar to HTML)
* All tags must be of the form <foo>content</foo> OR <foo /> with the latter form indicating no content. Don't forget the slash at the end.
* Inside a start tag you may specify attributes in the form key="value". Refer to manual for the valid attributes for each element.
* Comments may be included anywhere in the document except inside a tag where it's attribute list is defined.
* Comments may contain any characters except two dashes.
-->
<!-- Architecture based off Stratix IV
  Use closest ifar architecture: K06 N10 45nm fc 0.15 area-delay optimized, scale to 40 nm using linear scaling
  n10k06104.fc15.arealdelay1_cmos45nm.bptm_cmos45nm.xml
  * because documentation sparser for soft logic (delays not in QUIP), harder to track down, not worth our time considering the level of accuracy is approximate
  * delays multiplied by 40/45 to normalize for process difference between stratix 4 and 45 nm technology (called full scaling)
```
Use delay numbers off Altera device handbook:


multipliers at 600 MHz, no detail on 9x9 vs 36x36
* datasheets unclear
  * claims 4 18x18 independant multipliers, following test indicates that this is
    not the case:
      created 4 18x18 multipliers, logiclocked them to a single DSP block, compile
      result - 2 18x18 multipliers got packed together, the other 2 got ejected out
        of the logiclock region without error
    conclusion - just take the 600 MHz as is, and Quartus II logiclock hasn't fixed
      the bug that I've seen it do to registers when I worked at Altera (ie. eject without
      warning)
    -->

<architecture>
  <!-- ODIN II specific config -->
  <models>
    <model name="multiply">
      <input_ports>
        <port name="a" combinational_sink_ports="out"/>
        <port name="b" combinational_sink_ports="out"/>
      </input_ports>
      <output_ports>
        <port name="out"/>
      </output_ports>
    </model>
    <model name="single_port_ram">
      <input_ports>
        <port name="we" clock="clk"/>
        <port name="addr" clock="clk"/>
        <port name="data" clock="clk"/>
        <port name="clk" is_clock="1"/>
      </input_ports>
      <output_ports>
        <port name="out" clock="clk"/>
      </output_ports>
    </model>
    <model name="dual_port_ram">
      <input_ports>
        <port name="we1" clock="clk"/>
        <port name="we2" clock="clk"/>
        <port name="addr1" clock="clk"/>
      </input_ports>
      <output_ports>
        <port name="out" clock="clk"/>
      </output_ports>
    </model>
  </models>
</architecture>
<!-- address lines -->
<port name="addr2" clock="clk"/>
<!-- address lines -->
<port name="data1" clock="clk"/>
<!-- data lines can be broken down into smaller bit widths minimum size 1 -->
<port name="data2" clock="clk"/>
<!-- data lines can be broken down into smaller bit widths minimum size 1 -->
<port name="clk" is_clock="1"/>
<!-- memories are often clocked -->
</input_ports>

<output_ports>
<port name="out1" clock="clk"/>
<!-- output can be broken down into smaller bit widths minimum size 1 -->
<port name="out2" clock="clk"/>
<!-- output can be broken down into smaller bit widths minimum size 1 -->
</output_ports>
</model>
</models>
</tiles>
<tile name="io" capacity="8">
<equivalent_sites>
<site pb_type="io" pin_mapping="direct"/>
</equivalent_sites>
<input name="outpad" num_pins="1"/>
<output name="inpad" num_pins="1"/>
<clock name="clock" num_pins="1"/>
<fc in_type="frac" in_val="0.15" out_type="frac" out_val="0.10"/>
<pinlocations pattern="custom">
<loc side="left">io.outpad io.inpad io.clock</loc>
<loc side="top">io.outpad io.inpad io.clock</loc>
<loc side="right">io.outpad io.inpad io.clock</loc>
<loc side="bottom">io.outpad io.inpad io.clock</loc>
</pinlocations>
</tile>
</tiles>
<tile name="clb">
<equivalent_sites>
<site pb_type="clb" pin_mapping="direct"/>
</equivalent_sites>
<input name="I" num_pins="33" equivalent="full"/>
<output name="O" num_pins="10" equivalent="instance"/>
<clock name="clk" num_pins="1"/>
<fc in_type="frac" in_val="0.15" out_type="frac" out_val="0.10"/>
<pinlocations pattern="spread"/>
</tile>
</tiles>
<tile name="mult_36" height="4">
<equivalent_sites>
<site pb_type="mult_36" pin_mapping="direct"/>
</equivalent_sites>
<input name="a" num_pins="36"/>
<input name="b" num_pins="36"/>
<output name="out" num_pins="72"/>
<pinlocations pattern="spread"/>
<tile name="memory" height="6">
  <equivalent_sites>
    <site pb_type="memory" pin_mapping="direct"/>
  </equivalent_sites>
  <input name="addr1" num_pins="17"/>
  <input name="addr2" num_pins="17"/>
  <input name="data" num_pins="72"/>
  <input name="we1" num_pins="1"/>
  <input name="we2" num_pins="1"/>
  <output name="out" num_pins="72"/>
  <clock name="clk" num_pins="1"/>
  <fc in_type="frac" in_val="0.15" out_type="frac" out_val="0.10"/>
  <pinlocations pattern="spread"/>
</tile>
</tiles>

<!-- ODIN II specific config ends -->

<!-- Physical descriptions begin (area optimized for N8-K6-L4 -->

<layout>
  <auto_layout aspect_ratio="1.0">
    <!--Perimeter of 'io' blocks with 'EMPTY' blocks at corners-->
    <perimeter type="io" priority="100"/>
    <corners type="EMPTY" priority="101"/>
    <!--Fill with 'clb'-->
    <fill type="clb" priority="10"/>
    <!--Column of 'mult_36' with 'EMPTY' blocks wherever a 'mult_36' does not fit. Vertical offset by 1 for perimeter.-->
    <col type="mult_36" startx="4" starty="1" repeatx="8" priority="20"/>
    <col type="EMPTY" startx="4" repeatx="8" starty="1" priority="19"/>
    <!--Column of 'memory' with 'EMPTY' blocks wherever a 'memory' does not fit. Vertical offset by 1 for perimeter.-->
    <col type="memory" startx="2" starty="1" repeatx="8" priority="20"/>
    <col type="EMPTY" startx="2" repeatx="8" starty="1" priority="19"/>
  </auto_layout>
</layout>

<device>
  <sizing R_minW_nmos="6065.520020" R_minW_pmos="18138.500000"/>
  <area grid_logic_tile_area="14813.392"/>
  <!--area is for soft logic only-->
  <chan_width_distr>
    <x distr="uniform" peak="1.000000"/>
    <y distr="uniform" peak="1.000000"/>
  </chan_width_distr>
  <switch_block type="wilton" fs="3"/>
  <connection_block input_switch_name="ipin_cblock"/>
</device>

<switchlist>
  <switch type="mux" name="0" R="0.000000" Cin="0.000000e+00" Cout="0.000000e+00" Tdel="6.837e-11" mux_trans_size="2.630740" buf_size="27.645901"/>
  <!--switch ipin_cblock resistance set to yeild for 4x minimum drive strength buffer-->
  <switch type="mux" name="ipin_cblock" R="1516.380005" Cout="0." Cin="0.000000e+00" Tdel="7.247000e-11" mux_trans_size="1.222280" buf_size="auto"/>
</switchlist>

(continues on next page)
162</switchlist>
163<segmentlist>
164  <segment freq="1.000000" length="4" type="unidir" Rmetal="0.000000" Cmetal="0.000000e+00">
165    <mux name="0"/>
166    <sb type="pattern">1 1 1 1</sb>
167    <cb type="pattern">1 1 1</cb>
168  </segment>
169</segmentlist>
170<complexblocklist>
171<!-- Capacity is a unique property of I/Os, it is the maximum number of I/Os that can be placed at the same (X,Y) location on the FPGA -->
172  <pb_type name="io">
173    <input name="outpad" num_pins="1"/>
174    <output name="inpad" num_pins="1"/>
175    <clock name="clock" num_pins="1"/>
176    <!-- I/Os can operate as either inputs or outputs -->
177    <mode name="inpad">
178      <pb_type name="inpad" blif_model=".input" num_pb="1">
179        <output name="inpad" num_pins="1"/>
180      </pb_type>
181      <interconnect>
182        <direct name="inpad" input="inpad.inpad" output="io.inpad">
183          <delay_constant max="4.243e-11" in_port="inpad.inpad" out_port="io.inpad"/>
184        </direct>
185      </interconnect>
186    </mode>
187    <mode name="outpad">
188      <pb_type name="outpad" blif_model=".output" num_pb="1">
189        <input name="outpad" num_pins="1"/>
190      </pb_type>
191      <interconnect>
192        <direct name="outpad" input="io.outpad" output="outpad.outpad">
193          <delay_constant max="1.394e-11" in_port="io.outpad" out_port="outpad.outpad"/>
194        </direct>
195      </interconnect>
196    </mode>
197    <!-- I/Os go on the periphery of the FPGA, for consistency, make it physically equivalent on all sides so that only one definition of I/Os is needed. If I do not make a physically equivalent definition, then I need to define 4 different I/Os, one for each side of the FPGA -->
198  </pb_type>
199  <pb_type name="clb">
200    <input name="I" num_pins="33" equivalent="full"/> <!-- NOTE: Logically Equivalent -->
201    <output name="O" num_pins="10" equivalent="instance"/> <!-- NOTE: Logically Equivalent -->
202    <clock name="clk" num_pins="1"/>
203 </pb_type>
(continues on next page)
<pb_type name="ble" num_pb="10">
  <input name="in" num_pins="6"/>
  <output name="out" num_pins="1"/>
  <clock name="clk" num_pins="1"/>
</pb_type>

<pb_type name="soft_logic" num_pb="1">
  <input name="in" num_pins="6"/>
  <output name="out" num_pins="1"/>
  <mode name="n1_lut6">
    <pb_type name="lut6" blif_model=".names" num_pb="1" class="lut">
      <input name="in" num_pins="6" port_class="lut_in"/>
      <output name="out" num_pins="1" port_class="lut_out"/>
    </pb_type>
  </mode>
  </pb_type>
  <interconnect>
    <!-- LUT timing using delay matrix -->
    <delay_matrix type="max" in_port="lut6.in" out_port="lut6.out">
      2.690e-10
      2.690e-10
      2.690e-10
      2.690e-10
      2.690e-10
      2.690e-10
    </delay_matrix>
  </interconnect>
</pb_type>

<pb_type name="ff" blif_model=".latch" num_pb="1" class="flipflop">
  <input name="D" num_pins="1" port_class="D"/>
  <output name="Q" num_pins="1" port_class="Q"/>
  <clock name="clk" num_pins="1" port_class="clock"/>
  <T_setup value="2.448e-10" port="ff.D" clock="clk"/>
  <T_clock_to_Q max="7.732e-11" port="ff.Q" clock="clk"/>
</pb_type>

<interconnect>
  <!-- Two ff, make ff available to only corresponding luts -->
  <direct name="direct1" input="ble.in" output="soft_logic.in[5:0]"/>
  <direct name="direct2" input="soft_logic.out" output="ff.D"/>
  <direct name="direct4" input="ble.clk" output="ff.clk"/>
  <mux name="mux1" input="ff.Q soft_logic.out" output="ble.out"/>
</interconnect>

<interconnect>
  <complete name="crossbar" input="clb.I ble[9:0].out" output="ble[9:0].in">
    <delay_constant max="8.044000e-11" in_port="clb.I" out_port="ble[9:0].in"/>
    <delay_constant max="7.354000e-11" in_port="ble[9:0].out" out_port="ble[9:0].in"/>
  </complete>
  <complete name="clks" input="clb.clk" output="ble[9:0].clk"/>
  <direct name="clbouts" input="ble[9:0].out" output="clb.O"/>
</interconnect>

(continued on next page)

3.2. Example Architecture Specification
257 </pb_type>
258 <!-- This is the 36*36 uniform mult -->
259 <pb_type name="mult_36">
260 <input name="a" num_pins="36"/>
261 <input name="b" num_pins="36"/>
262 <output name="out" num_pins="72"/>
263 <mode name="two_divisible_mult_18x18">
264 <pb_type name="divisible_mult_18x18" num_pb="2">
265 <input name="a" num_pins="18"/>
266 <input name="b" num_pins="18"/>
267 <output name="out" num_pins="36"/>
268 </mode>
269 <mode name="two_mult_9x9">
270 <pb_type name="mult_9x9_slice" num_pb="2">
271 <input name="A_cfg" num_pins="9"/>
272 <input name="B_cfg" num_pins="9"/>
273 <output name="OUT_cfg" num_pins="18"/>
274 <pb_type name="mult_9x9" blif_model=".subckt multiply" num_pb="1">
275 <input name="a" num_pins="9"/>
276 <input name="b" num_pins="9"/>
277 <output name="out" num_pins="18"/>
278 <delay_constant max="1.667e-9" in_port="mult_9x9.a" out_port="mult_9x9.
279 →out"/>
280 <delay_constant max="1.667e-9" in_port="mult_9x9.b" out_port="mult_9x9.
281 →out"/>
282 </pb_type>
283 </mode>
284 <interconnect>
285 <direct name="a2a" input="mult_9x9_slice.A_cfg" output="mult_9x9.
286 →Slice[1:0].A_cfg"/>
287 <direct name="b2b" input="mult_9x9_slice.B_cfg" output="mult_9x9.
288 →Slice[1:0].B_cfg"/>
289 <direct name="out2out" input="mult_9x9.out" output="mult_9x9_slice.OUT_290 →out"/>
291 </interconnect>
292 </interconnect>
293 </pb_type>
294 <interconnect>
295 <direct name="a2a" input="divisible_mult_18x18.a" output="mult_9x9._297 →slice[1:0].A_cfg"/>
298 <direct name="b2b" input="divisible_mult_18x18.b" output="mult_9x9._299 →slice[1:0].B_cfg"/>
300 <direct name="out2out" input="mult_9x9_slice[1:0].OUT_cfg" output="301 divisible_mult_18x18.out"/>
302 </interconnect>
303 </interconnect>
304 <mode name="mult_18x18">
305 <pb_type name="mult_18x18_slice" num_pb="1">
306 <input name="A_cfg" num_pins="18"/>
307 <input name="B_cfg" num_pins="18"/>
308 <output name="OUT_cfg" num_pins="36"/>
309 <pb_type name="mult_18x18" blif_model=".subckt multiply" num_pb="1">
310 <input name="a" num_pins="18"/>
311 <input name="b" num_pins="18"/>
312 <output name="out" num_pins="36"/>
313 <delay_constant max="1.667e-9" in_port="mult_18x18.a" out_port="mult_18x18.out"/>
<delay_constant max="1.667e-9" in_port="mult_18x18.a" out_port="mult_18x18.out"/>
</pb_type>
<interconnect>
<direct name="a2a" input="mult_18x18_slice.A_cfg" output="mult_18x18.a"/>
<direct name="b2b" input="mult_18x18_slice.B_cfg" output="mult_18x18.b"/>
<direct name="out2out" input="mult_18x18.out" output="mult_18x18_slice.OUT_cfg"/>
</interconnect>
</pb_type>
</mode>
</pb_type>
</mode>
<mode name="mult_36x36">
<pb_type name="mult_36x36_slice" num_pb="1">
<input name="A_cfg" num_pins="36"/>
<input name="B_cfg" num_pins="36"/>
<output name="OUT_cfg" num_pins="72"/>
<pb_type name="mult_36x36" blif_model=".subckt multiply" num_pb="1">
<input name="a" num_pins="36"/>
<input name="b" num_pins="36"/>
<output name="out" num_pins="72"/>
<delay_constant max="1.667e-9" in_port="mult_36.a" out_port="mult_36x36.a"/>
<delay_constant max="1.667e-9" in_port="mult_36.b" out_port="mult_36x36.b"/>
<delay_constant max="1.667e-9" in_port="mult_36x36.out" out_port="mult_36x36.OUT_cfg"/>
</pb_type>
</interconnect>
</mode>
</pb_type>
</mode>
<interconnect>
<direct name="a2a" input="mult_36.a" output="mult_36x36_slice.A_cfg"/>
<direct name="b2b" input="mult_36.b" output="mult_36x36_slice.B_cfg"/>
<direct name="out2out" input="mult_36x36_slice.OUT_cfg" output="mult_36.out"/>
</interconnect>
</pb_type>
<interconnect>
<direct name="a2a" input="divisible_mult_18x18.a" output="mult_18x18_slice.A_cfg"/>
<direct name="b2b" input="divisible_mult_18x18.b" output="mult_18x18_slice.B_cfg"/>
<direct name="out2out" input="mult_18x18_slice.OUT_cfg" output="divisible_mult_18x18.out"/>
</interconnect>
</pb_type>
</mode>
</mode>
<interconnect>
<direct name="a2a" input="divisible_mult_18x18[1:0].a" output="mult_18x18_slice.A_cfg"/>
<direct name="b2b" input="divisible_mult_18x18[1:0].b" output="mult_18x18_slice.B_cfg"/>
<direct name="out2out" input="divisible_mult_18x18[1:0].out" output="mult_36.out"/>
</interconnect>
</pb_type>
<interconnect>
<direct name="a2a" input="mult_36.a" output="mult_36x36_slice.A_cfg"/>
<direct name="b2b" input="mult_36.b" output="mult_36x36_slice.B_cfg"/>
<direct name="out2out" input="mult_36x36_slice.OUT_cfg" output="mult_36.out"/>
</interconnect>
</pb_type>
<interconnect>
<direct name="a2a" input="mult_36x36_slice.A_cfg" output="mult_36x36.a"/>
<direct name="b2b" input="mult_36x36_slice.B_cfg" output="mult_36x36.b"/>
<direct name="out2out" input="mult_36x36.out" output="mult_36x36_slice.OUT_cfg"/>
</interconnect>
</pb_type>
 Verilog-to-Routing Documentation, Release 8.1.0-dev

345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
(continued from previous page)
Verilog-to-Routing Documentation, Release 8.1.0-dev

3.2. Example Architecture Specification
<output name="out1" num_pins="18" port_class="data_out1"/>
<output name="out2" num_pins="18" port_class="data_out2"/>
<clock name="clk" num_pins="1" port_class="clock"/>
<T_setup value="2.448e-10" port="mem_9182x18_dp.addr1" clock="clk"/>
<T_setup value="2.448e-10" port="mem_9182x18_dp.data1" clock="clk"/>
<T_setup value="2.448e-10" port="mem_9182x18_dp.we1" clock="clk"/>
<T_setup value="2.448e-10" port="mem_9182x18_dp.addr2" clock="clk"/>
<T_setup value="2.448e-10" port="mem_9182x18_dp.data2" clock="clk"/>
<T_setup value="2.448e-10" port="mem_9182x18_dp.we2" clock="clk"/>
<T_clock_to_Q max="1.852e-9" port="mem_9182x18_dp.out1" clock="clk"/>
<T_clock_to_Q max="1.852e-9" port="mem_9182x18_dp.out2" clock="clk"/>
</pb_type>
</interconnect>
</mode>
<mode name="mem_9182x18_sp" num_pb="1">
<pb_type name="mem_9182x18_sp" blif_model=".subckt single_port_ram" class="memory" num_pb="1">
<input name="addr" num_pins="13" port_class="address"/>
<input name="data" num_pins="18" port_class="data_in"/>
<input name="we" num_pins="1" port_class="write_en"/>
<output name="out" num_pins="18" port_class="data_out"/>
<clock name="clk" num_pins="1" port_class="clock"/>
<T_setup value="2.448e-10" port="mem_9182x18_sp.addr" clock="clk"/>
<T_setup value="2.448e-10" port="mem_9182x18_sp.data" clock="clk"/>
<T_setup value="2.448e-10" port="mem_9182x18_sp.we" clock="clk"/>
<T_clock_to_Q max="1.852e-9" port="mem_9182x18_sp.out" clock="clk"/>
</pb_type>
<interconnect>
<direct name="address1" input="memory.addr1[12:0]" output="mem_9182x18_dp.addr1"/>
<direct name="data1" input="memory.data[17:0]" output="mem_9182x18_dp.data1"/>
<direct name="writeen1" input="memory.we1" output="mem_9182x18_dp.we1"/>
<direct name="dataout1" input="mem_9182x18_dp.out1" output="memory.out[17:0]"/>
<direct name="clk" input="memory.clk" output="mem_9182x18_dp.clk"/>
</interconnect>
</mode>
<input name="addr1" num_pins="14" port_class="address1"/>
<input name="addr2" num_pins="14" port_class="address2"/>
<input name="data1" num_pins="9" port_class="data_in1"/>
<input name="data2" num_pins="9" port_class="data_in2"/>
<input name="we1" num_pins="1" port_class="write_en1"/>
<input name="we2" num_pins="1" port_class="write_en2"/>
<output name="out1" num_pins="9" port_class="data_out1"/>
<output name="out2" num_pins="9" port_class="data_out2"/>
<clock name="clk" num_pins="1" port_class="clock"/>
</pb_type>
</interconnect>
</mode>
<mode name="mem_18194x9_sp">
<pb_type name="mem_18194x9_sp" blif_model=".subckt single_port_ram" class="memory" num_pb="1">
<input name="addr" num_pins="14" port_class="address"/>
<input name="data" num_pins="9" port_class="data_in"/>
<input name="we" num_pins="1" port_class="write_en"/>
<output name="out" num_pins="9" port_class="data_out"/>
<clock name="clk" num_pins="1" port_class="clock"/>
<T_setup value="2.448e-10" port="mem_18194x9_sp.addr" clock="clk"/>
<T_setup value="2.448e-10" port="mem_18194x9_sp.data" clock="clk"/>
<T_setup value="2.448e-10" port="mem_18194x9_sp.we" clock="clk"/>
<T_clock_to_Q max="1.852e-9" port="mem_18194x9_sp.out" clock="clk"/>
</pb_type>
</interconnect>
</mode>
<mode name="mem_18194x9_sp">
<pb_type name="mem_18194x9_sp" blif_model=".subckt single_port_ram" class="memory" num_pb="1">
<input name="addr" num_pins="14" port_class="address"/>
<input name="data" num_pins="9" port_class="data_in"/>
<input name="we" num_pins="1" port_class="write_en"/>
<output name="out" num_pins="9" port_class="data_out"/>
<clock name="clk" num_pins="1" port_class="clock"/>
<T_setup value="2.448e-10" port="mem_18194x9_sp.addr" clock="clk"/>
<T_setup value="2.448e-10" port="mem_18194x9_sp.data" clock="clk"/>
<T_setup value="2.448e-10" port="mem_18194x9_sp.we" clock="clk"/>
<T_clock_to_Q max="1.852e-9" port="mem_18194x9_sp.out" clock="clk"/>
</pb_type>
</interconnect>
</mode>

</interconnect>
</mode>
</pb_type>
</complexblocklist>
</architecture>
VPR (Versatile Place and Route) is an open source academic CAD tool designed for the exploration of new FPGA architectures and CAD algorithms, at the packing, placement and routing phases of the CAD flow [BR97b, LKJ+09]. Since its public introduction, VPR has been used extensively in many academic projects partly because it is robust, well documented, easy-to-use, and can flexibly target a range of architectures.

VPR takes, as input, a description of an FPGA architecture along with a technology-mapped user circuit. It then performs packing, placement, and routing to map the circuit onto the FPGA. The output of VPR includes the FPGA configuration needed to implement the circuit and statistics about the final mapped design (e.g. critical path delay, area, etc).

### Placement (carry chains highlighted)  Critical Path

<table>
<thead>
<tr>
<th>Logical Connections</th>
<th>Routing Utilization</th>
</tr>
</thead>
</table>

**Motivation**

The study of FPGA CAD and architecture can be a challenging process partly because of the difficulty in conducting high-quality experiments. A quality CAD/architecture experiment requires realistic benchmarks, accurate architectural models, and robust CAD tools that can appropriately map the benchmark to the particular architecture in question. This is a lot of work. Fortunately, this work can be made easier if open source tools are available as a starting point.

The purpose of VPR is to make the packing, placement, and routing stages of the FPGA CAD flow robust and flexible so that it is easier for researchers to investigate future FPGAs.

### 4.1 Basic flow

The Place and Route process in VPR consists of several steps:

- Packing (combines primitives into complex blocks)
- Placement (places complex blocks within the FPGA grid)
- Routing (determines interconnections between blocks)
- Analysis (analyzes the implementation)

Each of these steps provides additional configuration options that can be used to customize the whole process.
4.1.1 Packing

The packing algorithm tries to combine primitive netlist blocks (e.g., LUTs, FFs) into groups, called Complex Blocks (as specified in the FPGA architecture file). The results from the packing process are written into a .net file. It contains a description of complex blocks with their inputs, outputs, used clocks and relations to other signals. It can be useful in analyzing how VPR packs primitives together.

A detailed description of the .net file format can be found in the Packed Netlist Format (.net) section.

4.1.2 Placement

This step assigns a location to the Complex Blocks (produced by packing) with the FPGA grid, while optimizing for wirelength and timing. The output from this step is written to the .place file, which contains the physical location of the blocks from the .net file.

The file has the following format:

<table>
<thead>
<tr>
<th>block_name</th>
<th>x</th>
<th>y</th>
<th>subblock_number</th>
</tr>
</thead>
</table>

where x and y are positions in the VPR grid and block_name comes from the .net file.

Example of a placing file:

```
Netlist_File: top.net Netlist_ID:
...SHA256:ce5217d251e04301759ee5a8f55f67c642de435b6c573148b67c195e054c1f9
Array size: 149 x 158 logic blocks

#block name x y subblk block number
#---------- -- -- ------ ------------
$auto$alumacc.cc:474:replace_alu$24.slice[1].carry4_full 53 32 0 #0
$auto$alumacc.cc:474:replace_alu$24.slice[2].carry4_full 53 31 0 #1
$auto$alumacc.cc:474:replace_alu$24.slice[3].carry4_full 53 30 0 #2
$auto$alumacc.cc:474:replace_alu$24.slice[4].carry4_full 53 29 0 #3
$auto$alumacc.cc:474:replace_alu$24.slice[5].carry4_full 53 28 0 #4
$auto$alumacc.cc:474:replace_alu$24.slice[6].carry4_part 53 27 0 #5
$auto$alumacc.cc:474:replace_alu$24.slice[0].carry4_1st_full 53 33 0 #6
out:LD7 9 5 0 #7
clk 42 26 0 #8
$false 35 26 0 #9
```

A detailed description of the .place file format can be found in the Placement File Format (.place) section.

4.1.3 Routing

This step determines how to connect the placed Complex Blocks together, according to the netlist connectivity and the routing resources of the FPGA chip. The router uses a Routing Resource (RR) Graph [BRM99] to represent the FPGA's available routing resources. The RR graph can be created in two ways:

1. Automatically generated by VPR from the FPGA architecture description [BR00], or
2. Loaded from an external RR graph file.

The output of routing is written into a .route file. The file describes each connection from input to its output through different routing resources of the FPGA. Each net starts with a SOURCE node and ends in a SINK node, potentially passing through complex block input/output pins (IPIN/OPIN nodes) and horizontal/vertical routing wires (CHANX/CHANY...
nodes). The pair of numbers in round brackets provides information on the (x, y) resource location on the VPR grid. The additional field provides information about the specific node.

An example routing file could look as follows:

```plaintext
Placement_File: top.place Placement_ID:
  SHA256:88d45f0bf7999e3f9331cddf3497d0028be58ffa324a019254c2ae7b4f5bfa7a
Array size: 149 x 158 logic blocks.

Routing:

Net 0 (counter[4])
  Node: 203972  SOURCE (53,32)  Class: 40  Switch: 0
  Node: 204095  OPIN (53,32)  Pin: 40  BLK-TL-SLICEL.CQ[0]  Switch: 189
  Node: 1027363  CHANY (52,32)  Track: 165  Switch: 7
  Node: 955959  CHANY (52,32)  Track: 240  Switch: 161
  Node: 955968  CHANY (52,32)  to (52,33)  Track: 90  Switch: 130
  Node: 203982  IPIN (53,32)  Pin: 1  BLK-TL-SLICEL.A2[0]  Switch: 0
  Node: 203933  SINK (53,32)  Class: 1  Switch: -1

Net 1 ($auto$alumacc.cc:474:replace_alu$24.O[6])
...
```

A detailed description of the .route file format can be found in the *Routing File Format (.route)* section.

### 4.1.4 Analysis

This step analyzes the resulting implementation, producing information about:

- Resource usage (e.g. block types, wiring)
- Timing (e.g. critical path delays and timing paths)
- Power (e.g. total power used, power broken down by blocks)

Note that VPR’s analysis can be used independently of VPR’s optimization stages, so long as the appropriate .net/.place/.route files are available.

### 4.2 Command-line Options
4.2.1 Basic Usage

At a minimum VPR requires two command-line arguments:

```
  vpr architecture circuit
```

where:

- **architecture** is an FPGA architecture description file
- **circuit** is the technology mapped netlist in BLIF format to be implemented

VPR will then pack, place, and route the circuit onto the specified architecture.

By default VPR will perform a binary search routing to find the minimum channel width required to route the circuit.

4.2.2 Detailed Command-line Options

VPR has a lot of options. The options most people will be interested in are:

- `--route_chan_width` (route at a fixed channel width), and
- `--disp` (turn on/off graphics).

In general for the other options the defaults are fine, and only people looking at how different CAD algorithms perform will try many of them. To understand what the more esoteric placer and router options actually do, see [BRM99] or download [BR96a, BR96b, BR97b, MBR00] from the author’s web page.

In the following text, values in angle brackets e.g. `<int> <float> <string> <file>`, should be replaced by the appropriate number, string, or file path. Values in curly braces separated by vertical bars, e.g. `{on | off}`, indicate all the permissible choices for an option.

**Stage Options**

VPR runs all stages of (pack, place, route, and analysis) if none of `--pack`, `--place`, `--route` or `--analysis` are specified.

- `--pack` Run packing stage
  
  **Default:** off

- `--place` Run placement stage
  
  **Default:** off

- `--route` Run routing stage This also implies –analysis if routing was successful.
  
  **Default:** off

- `--analysis` Run final analysis stage (e.g. timing, power).
  
  **Default:** off
Graphics Options

--disp {on | off}
Controls whether VPR’s interactive graphics are enabled. Graphics are very useful for inspecting and debugging the FPGA architecture and/or circuit implementation.

Default: off

--auto <int>
Can be 0, 1, or 2. This sets how often you must click Proceed to continue execution after viewing the graphics. The higher the number, the more infrequently the program will pause.

Default: 1

--save_graphics {on | off}
If set to on, this option will save an image of the final placement and the final routing created by vpr to pdf files on disk, with no need for any user interaction. The files are named vpr_placement.pdf and vpr_routing.pdf.

Default: off

--graphics_commands <string>
A set of semi-colon separated graphics commands.

• save_graphics <file> Saves graphics to the specified file (.png/.pdf/.svg). If <file> contains {i}, it will be replaced with an integer which increments each time graphics is invoked.

• set_macros <int> Sets the placement macro drawing state

• set_nets <int> Sets the net drawing state

• set_cpd <int> Sets the critica path delay drawing state

• set_routing_util <int> Sets the routing utilization drawing state

• set_clip_routing_util <int> Sets whether routing utilization values are clipped to [0., 1.]. Useful when a consistent scale is needed across images

• set_draw_block_outlines <int> Sets whether blocks have an outline drawn around them

• set_draw_block_text <int> Sets whether blocks have label text drawn on them

• set_draw_block_internals <int> Sets the level to which block internals are drawn

• set_draw_net_max_fanout <int> Sets the maximum fanout for nets to be drawn (if fanout is beyond this value the net will not be drawn)

• set_congestion <int> Sets the routing congestion drawing state

• exit <int> Exits VPR with specified exit code

Example:

```
save_graphics place.png; 
set_nets 1; save_graphics nets1.png;
set_nets 2; save_graphics nets2.png; set_nets 0;
set_cpd 1; save_graphics cpd1.png; 
set_cpd 3; save_graphics cpd3.png; set_cpd 0; 
set_routing_util 5; save_graphics routing_util5.png; 
set_routing_util 0; 
set_congestion 1; save_graphics congestion1.png;
```

The above toggles various graphics settings (e.g. drawing nets, drawing critical path) and then saves the results to .png files.

Note that drawing state is reset to its previous state after these commands are invoked.

Like the interactive graphics option`--disp` option, the `--auto` option controls how often the commands specified with this option are invoked.

4.2. Command-line Options
General Options

-h, --help
   Display help message then exit.

--version
   Display version information then exit.

--device <string>
   Specifies which device layout/floorplan to use from the architecture file.
   auto uses the smallest device satisfying the circuit’s resource requirements. Other values are assumed to be the names of device layouts defined in the FPGA Grid Layout section of the architecture file.

   Note: If the architecture contains both <auto_layout> and <fixed_layout> specifications, specifying an auto device will use the <auto_layout>.

   Default: auto

-j, --num_workers <int>
   Controls how many parallel workers VPR may use:
   • 1 implies VPR will execute serially,
   • >1 implies VPR may execute in parallel with up to the specified concurrency
   • 0 implies VPR may execute with up to the maximum concurrency supported by the host machine
   If this option is not specified it may be set from the VPR_NUM_WORKERS environment variable; otherwise the default is used.

   Note: To compile VPR to allow the usage of parallel workers, libtbb-dev must be installed in the system.

   Default: 1

--timing_analysis {on | off}
   Turn VPR timing analysis off. If it is off, you don’t have to specify the various timing analysis parameters in the architecture file.

   Default: on

--echo_file {on | off}
   Generates echo files of key internal data structures. These files are generally used for debugging vpr, and typically end in .echo

   Default: off

--verify_file_digests {on | off}
   Checks that any intermediate files loaded (e.g. previous packing/placement/routing) are consistent with the current netlist/architecture.
   If set to on will error if any files in the upstream dependancy have been modified. If set to off will warn if any files in the upstream dependancy have been modified.

   Default: on

--target_utilization <float>
   Sets the target device utilization. This corresponds to the maximum target fraction of device grid-tiles to be used.
   A value of 1.0 means the smallest device (which fits the circuit) will be used.

   Default: 1.0
--constant_net_method {global | route}
Specifies how constant nets (i.e. those driven to a constant value) are handled:

- global: Treat constant nets as globals (not routed)
- route: Treat constant nets as normal nets (routed)

Default: global

--clock_modeling {ideal | route | dedicated_network}
Specifies how clock nets are handled:

- ideal: Treat clock pins as ideal (i.e. no routing delays on clocks)
- route: Treat clock nets as normal nets (i.e. routed using inter-block routing)
- dedicated_network: Use the architectures dedicated clock network (experimental)

Default: ideal

--two_stage_clock_routing {on | off}
Routes clock nets in two stages using a dedicated clock network.

- First stage: From the net source (e.g. an I/O pin) to a dedicated clock network root (e.g. center of chip)
- Second stage: From the clock network root to net sinks.

Note this option only works when specifying a clock architecture, see Clock Architecture Format; it does not work when reading a routing resource graph (i.e. --read_rr_graph).

Default: off

--exit_before_pack {on | off}
Causes VPR to exit before packing starts (useful for statistics collection).

Default: off

--strict_checks {on, off}
Controls whether VPR enforces some consistency checks strictly (as errors) or treats them as warnings.

Usually these checks indicate an issue with either the targetted architecture, or consistency issues with VPR’s internal data structures/algorithms (possibly harming optimization quality). In specific circumstances on specific architectures these checks may be too restrictive and can be turned off.

Warning: Exercise extreme caution when turning this option off – be sure you completely understand why the issue is being flagged, and why it is OK to treat as a warning instead of an error.

Default: on

Filename Options

VPR by default appends .blif, .net, .place, and .route to the circuit name provided by the user, and looks for an SDC file in the working directory with the same name as the circuit. Use the options below to override this default naming behaviour.

--circuit_file <file>
Path to technology mapped user circuit in BLIF format.

Note: If specified the circuit positional argument is treated as the circuit name.

See also:
--circuit_format
--circuit_format {auto | blif | eblif}
File format of the input technology mapped user circuit.
  • auto: File format inferred from file extension (e.g. .blif or .eblif)
  • blif: Strict structural BLIF
  • eblif: Structural BLIF with extensions
Default: auto

--net_file <file>
Path to packed user circuit in net format.
Default: circuit.net

--place_file <file>
Path to final placement file.
Default: circuit.place

--route_file <file>
Path to final routing file.
Default: circuit.route

--sdc_file <file>
Path to SDC timing constraints file.
If no SDC file is found default timing constraints will be used.
Default: circuit.sdc

--write_rr_graph <file>
Writes out the routing resource graph generated at the last stage of VPR into RR Graph XML format
file describes the filename for the generated routing resource graph. The output can be read into VPR using
--read_rr_graph

--read_rr_graph <file>
Reads in the routing resource graph named <file> loads it for use during the placement and routing stages.
The routing resource graph overthrows all the architecture definitions regarding switches, nodes, and edges. Other
information such as grid information, block types, and segment information are matched with the architecture
file to ensure accuracy.
This file should be in XML format and can be easily obtained through --write_rr_graph

See also:
Routing Resource XML File.

--outfile_prefix <string>
Prefix for output files

Netlist Options

By default VPR will remove buffer LUTs, and iteratively sweep the netlist to remove unused primary inputs/outputs,
nets and blocks, until nothing else can be removed.

--absorb_buffer_luts {on | off}
Controls whether LUTs programmed as wires (i.e. implementing logical identity) should be absorbed into the
downstream logic.

Usually buffer LUTS are introduced in BLIF circuits by upstream tools in order to rename signals (like assign
statements in Verilog). Absorbing these buffers reduces the number of LUTs required to implement the circuit.
Ocassionally buffer LUTs are inserted for other purposes, and this option can be used to preserve them. Disabling buffer absorption can also improve the matching between the input and post-synthesis netlist/SDF.

**Default:** on

**--const_gen_inference {none | comb | comb_seq}**

Controls how constant generators are inferred/detected in the input circuit. Constant generators and the signals they drive are not considered during timing analysis.

- **none:** No constant generator inference will occur. Any signals which are actually constants will be treated as non-constants.
- **comb:** VPR will infer constant generators from combinational blocks with no non-constant inputs (always safe).
- **comb_seq:** VPR will infer constant generators from combinational *and* sequential blocks with only constant inputs (usually safe).

**Note:** In rare circumstances `comb_seq` could incorrectly identify certain blocks as constant generators. This would only occur if a sequential netlist primitive has an internal state which evolves *completely independently* of any data input (e.g. a hardened LFSR block, embedded thermal sensor).

**Default:** comb_seq

**--sweep_dangling_primary_ios {on | off}**

Controls whether the circuits dangling primary inputs and outputs (i.e. those who do not drive, or are not driven by anything) are swept and removed from the netlist.

Disabling sweeping of primary inputs/outputs can improve the matching between the input and post-synthesis netlists. This is often useful when performing formal verification.

See also:

**--sweep_constant_primary_outputs**

**Default:** on

**--sweep_dangling_nets {on | off}**

Controls whether dangling nets (i.e. those who do not drive, or are not driven by anything) are swept and removed from the netlist.

**Default:** on

**--sweep_dangling_blocks {on | off}**

Controls whether dangling blocks (i.e. those who do not drive anything) are swept and removed from the netlist.

**Default:** on

**--sweep_constant_primary_outputs {on | off}**

Controls whether primary outputs driven by constant values are swept and removed from the netlist.

See also:

**--sweep_dangling_primary_ios**

**Default:** off

**--netlist_verbosity <int>**

Controls the verbosity of netlist processing (constant generator detection, swept netlist components). High values produce more detailed output.

**Default:** 1
Packing Options

AAPack is the packing algorithm built into VPR. AAPack takes as input a technology-mapped blif netlist consisting of LUTs, flip-flops, memories, multipliers, etc and outputs a .net formatted netlist composed of more complex logic blocks. The logic blocks available on the FPGA are specified through the FPGA architecture file. For people not working on CAD, you can probably leave all the options to their default values.

--connection_driven_clustering {on | off}
Controls whether or not AAPack prioritizes the absorption of nets with fewer connections into a complex logic block over nets with more connections.

Default: on

--allow_unrelated_clustering {on | off | auto}
Controls whether primitives with no attraction to a cluster may be packed into it.

Unrelated clustering can increase packing density (decreasing the number of blocks required to implement the circuit), but can significantly impact routability.

When set to auto VPR automatically decides whether to enable unrelated clustering based on the targeted device and achieved packing density.

Default: auto

--alpha_clustering <float>
A parameter that weights the optimization of timing vs area.

A value of 0 focuses solely on area, a value of 1 focuses entirely on timing.

Default: 0.75

--beta_clustering <float>
A tradeoff parameter that controls the optimization of smaller net absorption vs. the optimization of signal sharing.

A value of 0 focuses solely on signal sharing, while a value of 1 focuses solely on absorbing smaller nets into a cluster. This option is meaningful only when connection_driven_clustering is on.

Default: 0.9

--timing_driven_clustering {on|off}
Controls whether or not to do timing driven clustering.

Default: on

--cluster_seed_type {blend | timing | max_inputs}
Controls how the packer chooses the first primitive to place in a new cluster.

timing means that the unclustered primitive with the most timing-critical connection is used as the seed.

max_inputs means the unclustered primitive that has the most connected inputs is used as the seed.

blend uses a weighted sum of timing criticality, the number of tightly coupled blocks connected to the primitive, and the number of its external inputs.

max_pins selects primitives with the most number of pins (which may be used, or unused).

max_input_pins selects primitives with the most number of input pins (which may be used, or unused).

blend2 An alternative blend formulation taking into account both used and unused pin counts, number of tightly coupled blocks and criticality.

Default: blend2 if timing_driven_clustering is on; max_inputs otherwise.
--clustering_pin_feasibility_filter  {on | off}
Controls whether the pin counting feasibility filter is used during clustering. When enabled the clustering engine counts the number of available pins in groups/classes of mutually connected pins within a cluster. These counts are used to quickly filter out candidate primitives/atoms/molecules for which the cluster has insufficient pins to route (without performing a full routing). This reduces packing run-time.

Default: on

--balance_block_type_utilization  {on, off, auto}
Controls how the packer selects the block type to which a primitive will be mapped if it can potentially map to multiple block types.

• on: Try to balance block type utilization by picking the block type with the (currenty) lowest utilization.
• off: Do not try to balance block type utilization
• auto: Dynamically enabled/disabled (based on density)

Default: auto

--target_ext_pin_util  { auto | <float> | <float>, <float> | <string>:<float> | <string>:<float>,<float> }
Sets the external pin utilization target (fraction between 0.0 and 1.0) during clustering. This determines how many pin the clustering engine will aim to use in a given cluster before closing it and opening a new cluster.

Setting this to 1.0 guides the packer to pack as densely as possible (i.e. it will keep adding molecules to the cluster until no more can fit). Setting this to a lower value will guide the packer to pack less densely, and instead creating more clusters. In the limit setting this to 0.0 will cause the packer to create a new cluster for each molecule.

Typically packing less densely improves routability, at the cost of using more clusters.

This option can take several different types of values:
• auto VPR will automatically determine appropriate target utilizations.
• <float> specifies the target input pin utilization for all block types.

For example:
– 0.7 specifies that all blocks should aim for 70% input pin utilization.
• <float>,<float> specifies the target input and output pin utilizations respectively for all block types.

For example:
– 0.7,0.9 specifies that all blocks should aim for 70% input pin utilization, and 90% output pin utilization.
• <string>:<float> and <string>:<float>,<float> specify the target pin utilizations for a specific block type (as above).

For example:
– clb:0.7 specifies that only clb type blocks should aim for 70% input pin utilization.
– clb:0.7,0.9 specifies that only clb type blocks should aim for 70% input pin utilization, and 90% output pin utilization.

Note: If a pin utilization target is unspecified it defaults to 1.0 (i.e. 100% utilization).

For example:
• 0.7 leaves the output pin utilization unspecified, which is equivalent to 0.7, 1.0.
• clb:0.7,0.9 leaves the pin utilizations for all other block types unspecified, so they will assume a default utilization of 1.0,1.0.

This option can also take multiple space-separated values. For example:
--target_ext_pin_util clb:0.5 dsp:0.9,0.7 0.8

would specify that clb blocks use a target input pin utilization of 50%, dsp blocks use a targets of 90% and 70% for inputs and outputs respectively, and all other blocks use an input pin utilization target of 80%.

Note: This option is only a guideline. If a molecule (e.g. a carry-chain with many inputs) would not otherwise fit into a cluster type at the specified target utilization the packer will fallback to using all pins (i.e. a target utilization of 1.0).

Note: This option requires --clustering_pin_feasibility_filter to be enabled.

Default: auto

--pack_prioritize_transitive_connectivity {on | off}
Controls whether transitive connectivity is prioritized over high-fanout connectivity during packing.

Default: on

--pack_high_fanout_threshold {auto | <int> | <string>:<int>}
Defines the threshold for high fanout nets within the packer.

This option can take several different types of values:
• auto VPR will automatically determine appropriate thresholds.
• <int> specifies the fanout threshold for all block types.

For example:
– 64 specifies that a threshold of 64 should be used for all blocks.
• <string>:<float> specifies the the threshold for a specific block type.

For example:
– clb:16 specifies that clb type blocks should use a threshold of 16.

This option can also take multiple space-separated values. For example:

--pack_high_fanout_threshold 128 clb:16

would specify that clb blocks use a threshold of 16, while all other blocks (e.g. DSPs/RAMs) would use a threshold of 128.

Default: auto

--pack_transitive_fanout_threshold <int>
Packer transitive fanout threshold.

Default: 4

--pack_feasible_block_array_size <int>
This value is used to determine the max size of the priority queue for candidates that pass the early filter legality test but not the more detailed routing filter.

Default: 30

--pack_verbosity <int>
Controls the verbosity of clustering output. Larger values produce more detailed output, which may be useful for debugging architecture packing problems.

Default: 2
--write_block_usage <file>
   Writes out to the file under path <file> cluster-level block usage summary in machine readable (JSON or XML)
or human readable (TXT) format. Format is selected based on the extension of <file>.

Placer Options

The placement engine in VPR places logic blocks using simulated annealing. By default, the automatic annealing
schedule is used [BR97b, BRM99]. This schedule gathers statistics as the placement progresses, and uses them to
determine how to update the temperature, when to exit, etc. This schedule is generally superior to any user-specified
schedule. If any of init_t, exit_t or alpha_t is specified, the user schedule, with a fixed initial temperature, final temper-
ature and temperature update factor is used.

See also:

Timing-Driven Placer Options

--seed <int>
   Sets the initial random seed used by the placer.
   Default: 1

--enable_timing_computations {on | off}
   Controls whether or not the placement algorithm prints estimates of the circuit speed of the placement it generates.
   This setting affects statistics output only, not optimization behaviour.
   Default: on if timing-driven placement is specified, off otherwise.

--inner_num <float>
   The number of moves attempted at each temperature is inner_num * num_blocks^((4/3)) in the circuit. The number
   of blocks in a circuit is the number of pads plus the number of clbs. Changing inner_num is the best way to
   change the speed/quality tradeoff of the placer, as it leaves the highly-efficient automatic annealing schedule on
   and simply changes the number of moves per temperature.

   Specifying -inner_num 10 will slow the placer by a factor of 10 while typically improving placement quality
   only by 10% or less (depends on the architecture). Hence users more concerned with quality than CPU time may
   find this a more appropriate value of inner_num.
   Default: 1.0

--init_t <float>
   The starting temperature of the anneal for the manual annealing schedule.
   Default: 100.0

--exit_t <float>
   The manual anneal will terminate when the temperature drops below the exit temperature.
   Default: 0.01

--alpha_t <float>
   The temperature is updated by multiplying the old temperature by alpha_t when the manual annealing schedule
   is enabled.
   Default: 0.8

--fix_pins {free | random}
   Controls how the placer handles I/O pads during placement.
   • free: The placer can move I/O locations to optimize the placement.
   • random: Fixes I/O pads to arbitrary locations and does not allow the placer to move them during the anneal
     (models the effect of poor board-level I/O constraints).
Note: the fix_pins option also used to accept a third argument - a place file that specified where I/O pins should be placed. This argument is no longer accepted by fix_pins. Instead, the fix_clusters option can now be used to lock down I/O pins.

**Default:** free.

**--fix_clusters** `{<file.place>}`
Controls how the placer handles blocks (of any type) during placement.

- `<file.place>`: A path to a file listing the desired location of blocks in the netlist.

This place location file is in the same format as a normal placement file, but does not require the first two lines which are normally at the top of a placement file that specify the netlist file, netlist ID, and array size.

**Default:** `''`

**--place_algorithm** `{bounding_box | criticality_timing | slack_timing}`
Controls the algorithm used by the placer.

- **bounding_box** Focuses purely on minimizing the bounding box wirelength of the circuit. Turns off timing analysis if specified.
- **criticality_timing** Focuses on minimizing both the wirelength and the connection timing costs (criticality * delay).
- **slack_timing** Focuses on improving the circuit slack values to reduce critical path delay.

**Default:** `criticality_timing`

**--place_quench_algorithm** `{bounding_box | criticality_timing | slack_timing}`
Controls the algorithm used by the placer during placement quench. The algorithm options have identical functionality as the ones used by the option **--place_algorithm**. If specified, it overrides the option **--place_algorithm** during placement quench.

**Default:** `criticality_timing`

**--place_chan_width** `<int>`
Tells VPR how many tracks a channel of relative width 1 is expected to need to complete routing of this circuit. VPR will then place the circuit only once, and repeatedly try routing the circuit as usual.

**Default:** 100

**--place_rlim_escape** `<float>`
The fraction of moves which are allowed to ignore the region limit. For example, a value of 0.1 means 10% of moves are allowed to ignore the region limit.

**Default:** 0.0

Setting any of the following options selects Dusty’s annealing schedule.

**--alpha_min** `<float>`
The minimum (starting) update factor (alpha) used. Ranges between 0 and alpha_max.

**Default:** 0.2

**--alpha_max** `<float>`
The maximum (stopping) update factor (alpha) used after which simulated annealing will complete. Ranges between alpha_min and 1.

**Default:** 0.9

**--alpha_decay** `<float>`
The rate at which alpha will approach 1: alpha(n) = 1 - (1 - alpha(n-1)) * alpha_decay Ranges between 0 and 1.

**Default:** 0.7
--anneal_success_min <float>
The minimum success ratio after which the temperature will reset to maintain the target success ratio. Ranges between 0 and anneal_success_target.
Default: 0.1

--anneal_success_target <float>
The temperature after each reset is selected to keep this target success ratio. Ranges between anneal_success_target and 1.
Default: 0.25

Timing-Driven Placer Options

The following options are only valid when the placement engine is in timing-driven mode (timing-driven placement is used by default).

--timing_tradeoff <float>
Controls the trade-off between bounding box minimization and delay minimization in the placer.
A value of 0 makes the placer focus completely on bounding box (wirelength) minimization, while a value of 1 makes the placer focus completely on timing optimization.
Default: 0.5

--recompute_crit_iter <int>
Controls how many temperature updates occur before the placer performs a timing analysis to update its estimate of the criticality of each connection.
Default: 1

--inner_loop_recompute_divider <int>
Controls how many times the placer performs a timing analysis to update its criticality estimates while at a single temperature.
Default: 0

--td_place_exp_first <float>
Controls how critical a connection is considered as a function of its slack, at the start of the anneal.
If this value is 0, all connections are considered equally critical. If this value is large, connections with small slacks are considered much more critical than connections with small slacks. As the anneal progresses, the exponent used in the criticality computation gradually changes from its starting value of td_place_exp_first to its final value of --td_place_exp_last.
Default: 1.0

--td_place_exp_last <float>
Controls how critical a connection is considered as a function of its slack, at the end of the anneal.
See also:
--td_place_exp_first
Default: 8.0

--place_delay_model {delta, delta_override}
Controls how the timing-driven placer estimates delays.
- delta The router is used to profile delay from various locations in the grid for various differences in position.
- delta_override Like delta but also includes special overrides to ensure effects of direct connects between blocks are accounted for. This is potentially more accurate but is more complex and depending on the architecture (e.g. number of direct connects) may increase place run-time.
**Default:** delta

**--place_delay_model_reducer** `{min, max, median, arithmean, geomean}`

When calculating delta delays for the placement delay model how are multiple values combined?

**Default:** min

**--place_delay_offset** `<float>`

A constant offset (in seconds) applied to the placer’s delay model.

**Default:** 0.0

**--place_delay_ramp_delta_threshold** `<float>`

The delta distance beyond which **--place_delay_ramp** is applied. Negative values disable the placer delay ramp.

**Default:** -1

**--place_delay_ramp_slope** `<float>`

The slope of the ramp (in seconds per grid tile) which is applied to the placer delay model for delta distance beyond **--place_delay_ramp_delta_threshold**.

**Default:** 0.0e-9

**--place_tsu_rel_margin** `<float>`

Specifies the scaling factor for cell setup times used by the placer. This effectively controls whether the placer should try to achieve extra margin on setup paths. For example a value of 1.1 corresponds to requesting 10%% setup margin.

**Default:** 1.0

**--place_tsu_abs_margin** `<float>`

Specifies an absolute offset added to cell setup times used by the placer. This effectively controls whether the placer should try to achieve extra margin on setup paths. For example a value of 500e-12 corresponds to requesting an extra 500ps of setup margin.

**Default:** 0.0

**--post_place_timing_report** `<file>`

Name of the post-placement timing report file to generate (not generated if unspecified).

---

**Router Options**

VPR uses a negotiated congestion algorithm (based on Pathfinder) to perform routing.

---

**Note:** By default the router performs a binary search to find the minimum routable channel width. To route at a fixed channel width use **--route_chan_width**.

---

**See also:**

*Timing-Driven Router Options*

**--max_router_iterations** `<int>`

The number of iterations of a Pathfinder-based router that will be executed before a circuit is declared unroutable (if it hasn’t routed successfully yet) at a given channel width.

*Speed-quality trade-off:* reducing this number can speed up the binary search for minimum channel width, but at the cost of some increase in final track count. This is most effective if **-initial_pres_fac** is simultaneously increased. Increase this number to make the router try harder to route heavily congested designs.

**Default:** 50
**--first_iter_pres_fac** <float>

Similar to **--initial_pres_fac**. This sets the present overuse penalty factor for the very first routing iteration. **--initial_pres_fac** sets it for the second iteration.

**Note:** A value of 0.0 causes congestion to be ignored on the first routing iteration.

Default: 0.0

**--initial_pres_fac** <float>

Sets the starting value of the present overuse penalty factor.

*Speed-quality trade-off:* increasing this number speeds up the router, at the cost of some increase in final track count. Values of 1000 or so are perfectly reasonable.

Default: 0.5

**--pres_fac_mult** <float>

Sets the growth factor by which the present overuse penalty factor is multiplied after each router iteration.

Default: 1.3

**--acc_fac** <float>

Specifies the accumulated overuse factor (historical congestion cost factor).

Default: 1

**--bb_factor** <int>

Sets the distance (in channels) outside of the bounding box of its pins a route can go. Larger numbers slow the router somewhat, but allow for a more exhaustive search of possible routes.

Default: 3

**--base_cost_type** {demand_only | delay_normalized | delay_normalized_length | delay_normalized_frequency | delay_normalized_length_frequency}

Sets the basic cost of using a routing node (resource).

- **demand_only** sets the basic cost of a node according to how much demand is expected for that type of node.
- **delay_normalized** is similar to **demand_only**, but normalizes all these basic costs to be of the same magnitude as the typical delay through a routing resource.
- **delay_normalized_length** like **delay_normalized**, but scaled by routing resource length.
- **delay_normalized_frequency** like **delay_normalized**, but scaled inversely by routing resource frequency.
- **delay_normalized_length_frequency** like **delay_normalized**, but scaled by routing resource length and scaled inversely by routing resource frequency.

Default: **delay_normalized_length** for the timing-driven router and **demand_only** for the breadth-first router

**--bend_cost** <float>

The cost of a bend. Larger numbers will lead to routes with fewer bends, at the cost of some increase in track count. If only global routing is being performed, routes with fewer bends will be easier for a detailed router to subsequently route onto a segmented routing architecture.

Default: 1 if global routing is being performed, 0 if combined global/detailed routing is being performed.

**--route_type** {global | detailed}

Specifies whether global routing or combined global and detailed routing should be performed.

Default: detailed (i.e. combined global and detailed routing)

**--route_chan_width** <int>

Tells VPR to route the circuit at the specified channel width.
Note: If the channel width is >= 0, no binary search on channel capacity will be performed to find the minimum number of tracks required for routing. VPR simply reports whether or not the circuit will route at this channel width.

Default: -1 (perform binary search for minimum routable channel width)

--min_route_chan_width_hint <int>
Hint to the router what the minimum routable channel width is.

The value provided is used to initialize the binary search for minimum channel width. A good hint may speed-up the binary search by avoiding time spent at congested channel widths which are not routable.

The algorithm is robust to incorrect hints (i.e. it continues to binary search), so the hint does not need to be precise.

This option may occasionally produce a different minimum channel width due to the different initialization.

See also:
--verify_binary_search

--verify_binary_search {on | off}
Force the router to check that the channel width determined by binary search is the minimum.

The binary search occasionally may not find the minimum channel width (e.g. due to router sub-optimality, or routing pattern issues at a particular channel width).

This option attempts to verify the minimum by routing at successively lower channel widths until two consecutive routing failures are observed.

--router_algorithm {breadth_first | timing_driven}
Selects which router algorithm to use.

Warning: The breadth_first router should NOT be used to compare the run-time/quality of alternate routing algorithms.

It is inferior to the timing_driven router from a circuit speed (2x - 10x slower) and run-time perspective (takes 10-100x longer on the large benchmarks). The breadth_first router is deprecated and may be removed in a future release.

The breadth_first router [BRM99] focuses solely on routing a design successfully, while the timing_driven router [BRM99, MZB20] focuses both on achieving a successful route and achieving good circuit speed.

The breadth-first router is capable of routing a design using slightly fewer tracks than the timing-driving router (typically 5% if the timing-driven router uses its default parameters. This can be reduced to about 2% if the router parameters are set so the timing-driven router pays more attention to routability and less to area). The designs produced by the timing-driven router are much faster, however, (2x - 10x) and it uses less CPU time to route.

Default: timing_driven

--min_incremental_reroute_fanout <int>
Incrementally re-route nets with fanout above the specified threshold.

This attempts to re-use the legal (i.e. non-congested) parts of the routing tree for high fanout nets, with the aim of reducing router execution time.

To disable, set value to a value higher than the largest fanout of any net.

Default: 16
--max_logged_overused_rr_nodes <int>
Prints the information on overused RR nodes to the VPR log file after each failed routing attempt.
If the number of overused nodes is above the given threshold N, then only the first N entries are printed to the log file.
Default: 20

--generate_rr_node_overuse_report {on | off}
Generates a detailed report on the overused RR nodes' information: report_overused_nodes.rpt.
This report is generated only when the final routing attempt fails (i.e. the whole routing process has failed).
In addition to the information that can be seen via --max_logged_overused_rr_nodes, this report prints out all the net ids that are associated with each overused RR node. Also, this report does not place a threshold upon the number of RR nodes printed.
Default: off

--write_timing_summary <file>
Writes out to the file under path <file> final timing summary in machine readable (JSON or XML) or human readable (TXT) format. Format is selected based on the extension of <file>. The summary consists of parameters:
- cpd - Final critical path delay (least slack) [ns]
- fmax - Maximal frequency of the implemented circuit [MHz]
- swns - setup Worst Negative Slack (sWNS) [ns]
- stns - Setup Total Negative Slack (sTNS) [ns]

Timing-Driven Router Options
The following options are only valid when the router is in timing-driven mode (the default).

--astar_fac <float>
Sets how aggressive the directed search used by the timing-driven router is.
Values between 1 and 2 are reasonable, with higher values trading some quality for reduced CPU time.
Default: 1.2

--max_criticality <float>
Sets the maximum fraction of routing cost that can come from delay (vs. coming from routability) for any net.
A value of 0 means no attention is paid to delay; a value of 1 means nets on the critical path pay no attention to congestion.
Default: 0.99

--criticality_exp <float>
Controls the delay - routability tradeoff for nets as a function of their slack.
If this value is 0, all nets are treated the same, regardless of their slack. If it is very large, only nets on the critical path will be routed with attention paid to delay. Other values produce more moderate tradeoffs.
Default: 1.0

--router_init_wirelength_abort_threshold <float>
The first routing iteration wirelength abort threshold. If the first routing iteration uses more than this fraction of available wirelength routing is aborted.
Default: 0.85

--incremental reroute_delay_ripup {on | off | auto}
Controls whether incremental net routing will rip-up (and re-route) a critical connection for delay, even if the routing is legal. auto enables delay-based rip-up unless routability becomes a concern.
Verilog-to-Routing Documentation, Release 8.1.0-dev

Default: auto

--routing_failure_predictor {safe | aggressive | off}

Controls how aggressive the router is at predicting when it will not be able to route successfully, and giving up early. Using this option can significantly reduce the runtime of a binary search for the minimum channel width.

safe only declares failure when it is extremely unlikely a routing will succeed, given the amount of congestion existing in the design.

aggressive can further reduce the CPU time for a binary search for the minimum channel width but can increase the minimum channel width by giving up on some routings that would succeed.

off disables this feature, which can be useful if you suspect the predictor is declaring routing failure too quickly on your architecture.

See also:

--verify_binary_search

Default: safe

--routing_budgets_algorithm { disable | minimax | scale_delay }

Warning: Experimental

Controls how the routing budgets are created. Routing budgets are used to guide VPR’s routing algorithm to consider both short path and long path timing constraints [FBC08].

disable is used to disable the budget feature. This uses the default VPR and ignores hold time constraints.

minimax sets the minimum and maximum budgets by distributing the long path and short path slacks depending on the the current delay values. This uses the routing cost valleys and Minimax-PERT algorithm [FBC08, YLS92].

scale_delay has the minimum budgets set to 0 and the maximum budgets is set to the delay of a net scaled by the pin criticality (net delay/pin criticality).

Default: disable

--save_routing_per_iteration {on | off}

Controls whether VPR saves the current routing to a file after each routing iteration. May be helpful for debugging.

Default: off

--congested_routing_iteration_threshold CONGESTED_ROUTING_ITERATION_THRESHOLD

Controls when the router enters a high effort mode to resolve lingering routing congestion. Value is the fraction of max_router_iterations beyond which the routing is deemed congested.

Default: 1.0 (never)

--route_bb_update {static, dynamic}

Controls how the router’s net bounding boxes are updated:

• static: bounding boxes are never updated

• dynamic: bounding boxes are updated dynamically as routing progresses (may improve routability of congested designs)

Default: dynamic

--router_high_fanout_threshold ROUTER_HIGH_FANOUT_THRESHOLD

Specifies the net fanout beyond which a net is considered high fanout. Values less than zero disable special behaviour for high fanout nets.
Default: 64

--router_lookahead \{classic, map\}
Controls what lookahead the router uses to calculate cost of completing a connection.
  • classic: The classic VPR lookahead
  • map: A more advanced lookahead which accounts for diverse wire types and their connectivity

Default: classic

--router_max_convergence_count <float>
Controls how many times the router is allowed to converge to a legal routing before halting. If multiple legal solutions are found the best quality implementation is used.

Default: 1

--router_reconvergence_cpd_threshold <float>
Specifies the minimum potential CPD improvement for which the router will continue to attempt re-convergent routing.

For example, a value of 0.99 means the router will not give up on reconvergent routing if it thinks a > 1% CPD reduction is possible.

Default: 0.99

--router_initial_timing \{all_critical | lookahead\}
Controls how criticality is determined at the start of the first routing iteration.
  • all_critical: All connections are considered timing critical.
  • lookahead: Connection criticalities are determined from timing analysis assuming (best-case) connection delays as estimated by the router’s lookahead.

Default: all_critical for the classic --router_lookahead, otherwise lookahead

--router_update_lower_bound_delays \{on | off\}
Controls whether the router updates lower bound connection delays after the 1st routing iteration.

Default: on

--router_first_iter_timing_report <file>
Name of the timing report file to generate after the first routing iteration completes (not generated if unspecified).

--router_debug_net <int>

Note: This option is likely only of interest to developers debugging the routing algorithm

Controls which option the router produces detailed debug information for.
  • For values >= 0, the value is the net ID for which detailed router debug information should be produced.
  • For value == -1, detailed router debug information is produced for all nets.
  • For values < -1, no router debug output is produced.

Warning: VPR must have been compiled with VTR_ENABLE_DEBUG_LOGGING on to get any debug output from this option.

Default: -2

--router_debug_sink_rr ROUTER_DEBUG_SINK_RR

Note: This option is likely only of interest to developers debugging the routing algorithm

Controls when router debugging is enabled for the specified sink RR.
• For values >= 0, the value is taken as the sink RR Node ID for which to enable router debug output.
• For values < 0, sink-based router debug output is disabled.

**Warning:** VPR must have been compiled with `VTR_ENABLE_DEBUG_LOGGING` on to get any debug output from this option.

Default: -2

### Analysis Options

---

**full_stats**

Print out some extra statistics about the circuit and its routing useful for wireability analysis.

Default: off

**gen_post_synthesis_netlist** { on | off }

Generates the Verilog and SDF files for the post-synthesized circuit. The Verilog file can be used to perform functional simulation and the SDF file enables timing simulation of the post-synthesized circuit.

The Verilog file contains instantiated modules of the primitives in the circuit. Currently VPR can generate Verilog files for circuits that only contain LUTs, Flip Flops, IOs, Multipliers, and BRAMs. The Verilog description of these primitives are in the primitives.v file. To simulate the post-synthesized circuit, one must include the generated Verilog file and also the primitives.v Verilog file, in the simulation directory.

See also:

Post-Implementation Timing Simulation

If one wants to generate the post-synthesized Verilog file of a circuit that contains a primitive other than those mentioned above, he/she should contact the VTR team to have the source code updated. Furthermore to perform simulation on that circuit the Verilog description of that new primitive must be appended to the primitives.v file as a separate module.

Default: off

**post_synth_netlist_unconn_inputs** { unconnected | nets | gnd | vcc }

Controls how unconnected input cell ports are handled in the post-synthesis netlist

- unconnected: leave unconnected
- nets: connect each unconnected input pin to its own separate undriven net named: `__vpr__unconn<ID>`, where <ID> is index assigned to this occurrence of unconnected port in design
- gnd: tie all to ground (1'b0)
- vcc: tie all to VCC (1'b1)

Default: unconnected

**post_synth_netlist_unconn_outputs** { unconnected | nets }

Controls how unconnected output cell ports are handled in the post-synthesis netlist

- unconnected: leave unconnected
- nets: connect each unconnected output pin to its own separate undriven net named: `__vpr__unconn<ID>`, where <ID> is index assigned to this occurrence of unconnected port in design

Default: unconnected

**timing_report_npaths** <int>

Controls how many timing paths are reported.

Note: The number of paths reported may be less than the specified value, if the circuit has fewer paths.
Default: 100

```
--timing_report_detail { netlist | aggregated | detailed }
```

Controls the level of detail included in generated timing reports.

We obtained the following results using the k6_frac_N10_frac_chain_mem32K_40nm.xml architecture and multi-clock.blif circuit.

- **netlist**: Timing reports show only netlist primitive pins.

For example:

```
#Path 2
Startpoint: FFC.Q[0] (.latch clocked by clk)
Endpoint : out:out1.outpad[0] (.output clocked by virtual_io_clock)
Path Type : setup

Point          ˓→ Incr          Path

--------------------------------------------------------------------------
| clock clk (rise edge)          0.        | clock source latency          0.        |
| ...000  0.000                  | ...000  0.000                  |
| clk.inpad[0] (.input)          0.        | FFC.clk[0] (.latch)            0.        |
| ...000  0.000                  | ...042  0.042                  |
| FFC.Q[0] (.latch) [clock-to-output] 0.    | ...124  0.166                  |
| ...550  0.717                  | out:out1.outpad[0] (.output)   0.        |
|                             | ...550  0.717                  |

data arrival time          ˓→ 0.717

--------------------------------------------------------------------------
| clock virtual_io_clock (rise edge) 0.          | clock source latency 0.          |
| ...000  0.000                  | ...000  0.000                  |
| clock uncertainty            0.        | output external delay 0.        |
| ...000  0.000                  | ...000  0.000                  |
| data required time          ˓→ 0.000     |

--------------------------------------------------------------------------
| data required time          ˓→ 0.000     |
| data arrival time          ˓→ -0.717    |

--------------------------------------------------------------------------
| slack (VIOLATED)          ˓→ -0.717    |
```
aggregated: Timing reports show netlist pins, and an aggregated summary of intra-block and inter-block routing delays.

For example:

```text
#Path 2
Startpoint: FFC.Q[0] (.latch at (3,3) clocked by clk)
Endpoint : out:out1.outpad[0] (.output at (3,4) clocked by virtual_io_clock)
Path Type : setup

Point \rightarrow \text{Incr Path}

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>clock clk</td>
<td>0.</td>
<td>0.000</td>
</tr>
<tr>
<td>clock source latency</td>
<td>0.</td>
<td>0.000</td>
</tr>
<tr>
<td>clk.inpad[0] (.input at (4,2))</td>
<td>0.</td>
<td>0.000</td>
</tr>
<tr>
<td>\text{(intra 'io' routing)}</td>
<td>0.</td>
<td>0.042</td>
</tr>
<tr>
<td>\text{(inter-block routing)}</td>
<td>0.</td>
<td>0.042</td>
</tr>
<tr>
<td>\text{(intra 'clb' routing)}</td>
<td>0.</td>
<td>0.042</td>
</tr>
<tr>
<td>FFC.clk[0] (.latch at (3,3))</td>
<td>0.</td>
<td>0.042</td>
</tr>
<tr>
<td>\text{(primitive '.latch Tcq_max)}</td>
<td>0.</td>
<td>0.124</td>
</tr>
<tr>
<td>FFC.Q[0] (.latch at (3,3)) [clock-to-output]</td>
<td>0.</td>
<td>0.166</td>
</tr>
<tr>
<td>\text{(intra 'clb' routing)}</td>
<td>0.</td>
<td>0.045</td>
</tr>
<tr>
<td>\text{(inter-block routing)}</td>
<td>0.</td>
<td>0.211</td>
</tr>
<tr>
<td>\text{(intra 'io' routing)}</td>
<td>0.</td>
<td>0.491</td>
</tr>
<tr>
<td>\text{(inter-block routing)}</td>
<td>0.</td>
<td>0.703</td>
</tr>
<tr>
<td>\text{(intra 'io' routing)}</td>
<td>0.</td>
<td>0.714</td>
</tr>
<tr>
<td>out:out1.outpad[0] (.output at (3,4))</td>
<td>0.</td>
<td>0.717</td>
</tr>
<tr>
<td>data arrival time</td>
<td>\text{\rightarrow 0.717}</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>clock virtual_io_clock (rise edge)</td>
<td>0.</td>
<td>0.000</td>
</tr>
<tr>
<td>clock source latency</td>
<td>0.</td>
<td>0.000</td>
</tr>
<tr>
<td>clock uncertainty</td>
<td>0.</td>
<td>0.000</td>
</tr>
<tr>
<td>output external delay</td>
<td>0.</td>
<td>0.000</td>
</tr>
<tr>
<td>data required time</td>
<td>\text{\rightarrow 0.000}</td>
<td></td>
</tr>
</tbody>
</table>
```

(continues on next page)
where each line prefixed with | (pipe character) represent a sub-delay of an edge within the timing graph.

For instance:

| FF.C.Q[0] (.latch at (3,3)) [clock-to-output] 0. |
| --- | --- |
| | 0.000 0.166 |
| | (intra 'clb' routing) 0. |
| | 0.045 0.211 |
| | (inter-block routing) 0. |
| | 0.491 0.703 |
| | (intra 'io' routing) 0. |
| | 0.014 0.717 |
| out:out1.outpad[0] (.output at (3,4)) 0. |
| --- | --- |
| | 0.000 0.717 |

indicates that between the netlist pins FF.C.Q[0] and out:out1.outpad[0] there are delays of:

- 45 ps from the .latch output pin to an output pin of a clb block,
- 491 ps through the general inter-block routing fabric, and
- 14 ps from the input pin of a io block to .output.

Also note that a connection between two pins can be contained within the same clb block, and does not use the general inter-block routing network. As an example from a completely different circuit-architecture pair:

| n168.out[0] (.names) 0. |
| --- | --- |
| | 0.000 0.902 |
| | (intra 'clb' routing) 0. |
| | 0.000 0.902 |
| top^finish_FF_NODE.D[0] (.latch) 0. |
| --- | --- |
| | 0.000 0.902 |

• detailed: Like aggregated, the timing reports show netlist pins, and an aggregated summary of intra-block. In addition, it includes a detailed breakdown of the inter-block routing delays.

It is important to note that detailed timing report can only list the components of a non-global net, otherwise, it reports inter-block routing as well as an incremental delay of 0, just as in the aggregated and netlist reports.

For example:
where each line prefixed with | (pipe character) represent a sub-delay of an edge within the timing graph. In the detailed mode, the inter-block routing has now been replaced by the net components.

For OPINS and IPINS, this is the format of the name: | (ROUTING_RESOURCE_NODE_TYPE:ROUTING_RESOURCE_NODE_ID side:SIDE (START_COORDINATES)->(END_COORDINATES))

For CHANX and CHANY, this is the format of the name: | (ROUTING_RESOURCE_NODE_TYPE:ROUTING_RESOURCE_NODE_ID SEGMENT_NAME length:LENGTH (START_COORDINATES)->(END_COORDINATES))

Here is an example of the breakdown:

| FFC.Q[0] (.latch at (3,3)) [clock-to-output] 0. |
| (intra 'clb' routing) 0. |
| (OPIN:1479 side:TOP (3,3)) 0. |
| (CHANX:2073 unnamed_segment_0 length:1 (3,3)->(2,3)) 0. |
| (CHANY:2139 unnamed_segment_0 length:0 (1,3)->(1,3)) 0. |
| (CHANX:2040 unnamed_segment_0 length:1 (2,2)->(3,2)) 0. |
| (CHANY:2166 unnamed_segment_0 length:0 (2,3)->(3,3)) 0. |
| (CHANX:2076 unnamed_segment_0 length:0 (3,3)->(3,3)) 0. |
| (IPIN:1532 side:BOTTOM (3,4)) 0. |
| (intra 'io' routing) 0. |
| out:out1.outpad[0] (.output at (3,4)) 0. |

(continues on next page)
indicates that between the netlist pins FFC.Q[0] and out:out1.outpad[0] there are delays of:

- 45 ps from the .latch output pin to an output pin of a clb block,
- 0 ps from the clb output pin to the CHANX:2073 wire,
- 95 ps from the CHANX:2073 to the CHANY:2139 wire,
- 75 ps from the CHANY:2139 to the CHANX:2040 wire,
- 95 ps from the CHANX:2040 to the CHANY:2166 wire,
- 76 ps from the CHANY:2166 to the CHANX:2076 wire,
- 78 ps from the CHANX:2076 to the input pin of a io block,
- 14 ps input pin of a io block to .output.

In the initial description we referred to the existence of global nets, which also occur in this net:

<table>
<thead>
<tr>
<th>clk.inpad[0] (.input at (4,2))</th>
<th>0.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>.000</td>
</tr>
<tr>
<td></td>
<td>.042</td>
</tr>
<tr>
<td></td>
<td>.000</td>
</tr>
<tr>
<td></td>
<td>(intra 'io' routing)</td>
</tr>
<tr>
<td></td>
<td>.000</td>
</tr>
<tr>
<td></td>
<td>.000</td>
</tr>
<tr>
<td></td>
<td>(inter-block routing:global net)</td>
</tr>
<tr>
<td></td>
<td>.000</td>
</tr>
<tr>
<td>FFC.clk[0] (.latch at (3,3))</td>
<td>0.</td>
</tr>
<tr>
<td></td>
<td>.000</td>
</tr>
</tbody>
</table>

Global nets are unrouted nets, and their route trees happen to be null.

Finally, is interesting to note that the consecutive channel components may not seem to connect. There are two types of occurrences:

1. The preceding channel’s ending coordinates extend past the following channel’s starting coordinates (example from a different path):

| | (chany:2113 unnamed_segment_0 length:2 (1, 3) -> (1, 1)) | 0. |
|-------------------------------|----|
|                               | .116 | 0.405 |
| | (chanx:2027 unnamed_segment_0 length:0 (1, 2) -> (1, 2)) | 0. |
|                               | .078 | 0.482 |

It is possible that by opening a switch between (1,2) to (1,1), CHANY:2113 actually only extends from (1,3) to (1,2).

2. The preceding channel’s ending coordinates have no relation to the following channel’s starting coordinates. There is no logical contradiction, but for clarification, it is best to see an explanation of the VPR coordinate system. The path can also be visualized by VPR graphics, as an illustration of this point:

Fig. 4.1 shows the routing resources used in Path #2 and their locations on the FPGA.

1. The signal emerges from near the top-right corner of the block to FFC (OPIN:1479) and joins the topmost horizontal segment of length 1 (CHANX:2073).
Fig. 4.1: Illustration of Path #2 with insight into the coordinate system.
2. The signal proceeds to the left, then connects to the outermost, blue vertical segment of length 0 (CHANY:2139).

3. The signal continues downward and attaches to the horizontal segment of length 1 (CHANX:2040).

4. Of the aforementioned horizontal segment, after travelling one linear unit to the right, the signal jumps on a vertical segment of length 0 (CHANY:2166).

5. The signal travels upward and promptly connects to a horizontal segment of length 0 (CHANX:2076).

6. This segment connects to the green destination io (3,4).

   • **debug**: Like detailed, but includes additional VPR internal debug information such as timing graph node IDs (tnode) and routing SOURCE/SINK nodes.

   **Default**: netlist

   **--echo_dot_timing_graph_node** { string | int }

   Controls what subset of the timing graph is echoed to a GraphViz DOT file when `vpr --echo_file` is enabled.

   Value can be a string (corresponding to a VPR atom netlist pin name), or an integer representing a timing graph node ID. Negative values mean the entire timing graph is dumped to the DOT file.

   **Default**: -1

   **--timing_report_skew** { on | off }

   Controls whether clock skew timing reports are generated.

   **Default**: off

---

### Power Estimation Options

The following options are used to enable power estimation in VPR.

**See also:**

*Power Estimation* for more details.

**--power**

Enable power estimation

**Default**: off

**--tech_properties** `<file>`

XML File containing properties of the CMOS technology (transistor capacitances, leakage currents, etc). These can be found at `$VTR_ROOT/vtr_flow/tech/`, or can be created for a user-provided SPICE technology (see *Power Estimation*).

**--activity_file** `<file>`

File containing signal activities for all of the nets in the circuit. The file must be in the format:

```
<net name1> <signal probability> <transition density>
<net name2> <signal probability> <transition density>
...
```

Instructions on generating this file are provided in *Power Estimation*. 
4.2.3 Command-line Auto Completion

To simplify using VPR on the command-line you can use the `dev/vpr_bash_completion.sh` script, which will enable TAB completion for VPR commandline arguments (based on the output of `vpr -h`).

Simply add:

```
source $VTR_ROOT/dev/vpr_bash_completion.sh
```

to your `.bashrc`. $VTR_ROOT refers to the root of the VTR source tree on your system.

4.3 Graphics

VPR includes easy-to-use graphics for visualizing both the targetted FPGA architecture, and the circuit VPR has implementation on the architecture.

4.3.1 Enabling Graphics

Compiling with Graphics Support

The build system will attempt to build VPR with graphics support by default. If all the required libraries are found the build system will report:

```
-- EZGL: graphics enabled
```

If the required libraries are not found cmake will report:

```
-- EZGL: graphics disabled
```

and list the missing libraries:

```
-- EZGL: Failed to find required X11 library (on debian/ubuntu try 'sudo apt-get install libx11-dev' to install)
-- EZGL: Failed to find required Xft library (on debian/ubuntu try 'sudo apt-get install libxft-dev' to install)
-- EZGL: Failed to find required fontconfig library (on debian/ubuntu try 'sudo apt-get install fontconfig' to install)
-- EZGL: Failed to find required cairo library (on debian/ubuntu try 'sudo apt-get install libcairo2-dev' to install)
```
Enabling Graphics at Run-time

When running VPR provide `vpr --disp` on to enable graphics.

Saving Graphics at Run-time

When running VPR provide `vpr --save_graphics` on to enable graphics.

A graphical window will now pop up when you run VPR.

4.3.2 Navigation

Click on Zoom-Fit buttons to zoom the view. Click and drag with the left mouse button to pan the view, or scroll the mouse wheel to zoom in and out. Click on the Window, then on the diagonally opposite corners of a box, to zoom in on a particular area.

Click on Save to save the image on screen to PDF, PNG, or SVG file.

Proceed tells VPR to continue with the next step in placing and routing the circuit.

Note: Menu buttons will be greyed out when they are not selectable (e.g. VPR is working).

4.3.3 Visualizing Placement

By default VPR’s graphics displays the FPGA floorplan (block grid) and current placement.

Fig. 4.2: Placement with macros (carry chains) highlighted

If the Placement Macros drop down is set, any placement macros (e.g. carry chains, which require specific relative placements between some blocks) will be highlighted.

4.3.4 Visualizing Netlist Connectivity

The Toggle Nets drop-down list toggles the nets in the circuit visible/invisible.

When a placement is being displayed, routing information is not yet known so nets are simply drawn as a “star;” that is, a straight line is drawn from the net source to each of its sinks. Click on any clb in the display, and it will be highlighted in green, while its fanin and fanout are highlighted in blue and red, respectively. Once a circuit has been routed the true path of each net will be shown.

Fig. 4.3: Logical net connectivity during placement

If the nets routing are shown, click on a clb or pad to highlight its fanins and fanouts, or click on a pin or channel wire to highlight a whole net in magenta. Multiple nets can be highlighted by pressing ctrl + mouse click.
4.3.5 Visualizing the Critical Path

During placement and routing you can click on the Crit. Path drop-down menu to visualize the critical path. Each stage between primitive pins is shown in a different colour. Cliking the Crit. Path button again will toggle through the various visualizations: * During placement the critical path is shown only as flylines. * During routing the critical path can be shown as both flylines and routed net connections.

Fig. 4.4: Critical Path flylines during placement and routing

4.3.6 Visualizing Routing Architecture

When a routing is on-screen, clicking on Toggle RR lets you to choose between various views of the routing resources available in the FPGA.

Fig. 4.5: Routing Architecture Views

The routing resource view can be very useful in ensuring that you have correctly described your FPGA in the architecture description file – if you see switches where they shouldn’t be or pins on the wrong side of a clb, your architecture description needs to be revised.

Wiring segments are drawn in black, input pins are drawn in sky blue, and output pins are drawn in pink. Sinks are drawn in dark slate blue, and sources in plum. Direct connections between output and input pins are shown in medium purple. Connections from wiring segments to input pins are shown in sky blue, connections from output pins to wiring segments are shown in pink, and connections between wiring segments are shown in green. The points at which wiring segments connect to clb pins (connection box switches) are marked with an x.

Switch box connections will have buffers (triangles) or pass transistors (circles) drawn on top of them, depending on the type of switch each connection uses. Clicking on a clb or pad will overlay the routing of all nets connected to that block on top of the drawing of the FPGA routing resources, and will label each of the pins on that block with its pin number. Clicking on a routing resource will highlight it in magenta, and its fanouts will be highlighted in red and fanins in blue. Multiple routing resources can be highlighted by pressing ctrl + mouse click.

4.3.7 Visualizing Routing Congestion

When a routing is shown on-screen, clicking on the Congestion drop-down menu will show a heat map of any overused routing resources (wires or pins). Lighter colours (e.g. yellow) correspond to highly overused resources, while darker colours (e.g. blue) correspond to lower overuse. The overuse range shown at the bottom of the window.

Fig. 4.6: Routing Congestion during placement and routing
### 4.3.8 Visualizing Routing Utilization

When a routing is shown on-screen, clicking on the **Routing Util** drop-down menu will show a heat map of routing wire utilization (i.e. fraction of wires used in each channel). Lighter colours (e.g. yellow) correspond to highly utilized channels, while darker colours (e.g. blue) correspond to lower utilization.

![Fig. 4.7: Routing Utilization during placement and routing](image)

### 4.3.9 Toggle Block Internal

During placement and routing you can adjust the level of block detail you visualize by using the **Toggle Block Internal**. Each block can contain a number of flip flops (ff), look up tables (lut), and other primitives. The higher the number, the deeper into the hierarchy within the cluster level block you see.

![Fig. 4.8: Visualizing Block Internals](image)

### 4.3.10 Button Description Table

<table>
<thead>
<tr>
<th>Buttons</th>
<th>Stages</th>
<th>Functionalities</th>
<th>Detailed Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blk Internal</td>
<td>Placement/Routing</td>
<td>Controls depth of sub-blocks shown</td>
<td>Click multiple times to show more details; Click to reset when reached maximum level of detail</td>
</tr>
<tr>
<td>Toggle Block Internal</td>
<td>Placement/Routing</td>
<td>Adjusts the level of visualized block detail</td>
<td>Click multiple times to go deeper into the hierarchy within the cluster level block</td>
</tr>
<tr>
<td>Blk Pin Util</td>
<td>Placement/Routing</td>
<td>Visualizes block pin utilization</td>
<td>Click multiple times to visualize all block pin utilization, input block pin utilization, or output block pin utilization</td>
</tr>
<tr>
<td>Cong. Cost</td>
<td>Routing</td>
<td>Visualizes the congestion costs of routing resources</td>
<td></td>
</tr>
<tr>
<td>Congestion</td>
<td>Routing</td>
<td>Visualizes a heat map of overused routing resources</td>
<td></td>
</tr>
<tr>
<td>Crit. Path</td>
<td>Placement/Routing</td>
<td>Visualizes the critical path of the circuit</td>
<td></td>
</tr>
<tr>
<td>Place Macros</td>
<td>Placement/Routing</td>
<td>Visualizes placement macros</td>
<td></td>
</tr>
<tr>
<td>Route BB</td>
<td>Routing</td>
<td>Visualizes net bounding boxes one by one</td>
<td>Click multiple times to sequence through the net being shown</td>
</tr>
<tr>
<td>Router Cost</td>
<td>Routing</td>
<td>Visualizes the router costs of different routing resources</td>
<td></td>
</tr>
<tr>
<td>Routing Util</td>
<td>Routing</td>
<td>Visualizes routing channel utilization with colors indicating the fraction of wires used within a channel</td>
<td></td>
</tr>
<tr>
<td>Toggle Nets</td>
<td>Placement/Routing</td>
<td>Visualizes the nets in the circuit</td>
<td>Click multiple times to set the nets to be visible / invisible</td>
</tr>
<tr>
<td>Toggle RR</td>
<td>Placement/Routing</td>
<td>Visualizes different views of the routing resources</td>
<td>Click multiple times to switch between routing resources available in the FPGA</td>
</tr>
</tbody>
</table>
4.3.11 Manual Moves

The manual moves feature allows the user to specify the next move in placement. If the move is legal, blocks are swapped and the new move is shown on the architecture.

To enable the feature, activate the Manual Move toggle button and press Proceed. Alternatively, the user can activate the Manual Move toggle button and click on the block to be moved.

On the manual move window, the user can specify the Block ID/Block name of the block to move and the To location, with the x position, y position and subtile position. For the manual move to be valid:

- The To location requested by the user should be within the grid’s dimensions.
- The block to be moved is found, valid and not fixed.
- The blocks to be swapped are compatible.
- The location chosen by the user is different from the block’s current location.

If the manual move is legal, the cost summary window will display the delta cost, delta timing, delta bounding box cost and the placer’s annealing decision that would result from this move.

The user can Accept or Reject the manual move based on the values provided. If accepted the block’s new location is shown.
4.4 Timing Constraints

VPR supports setting timing constraints using Synopsys Design Constraints (SDC), an industry-standard format for specifying timing constraints.

VPR’s default timing constraints are explained in Default Timing Constraints. The subset of SDC supported by VPR is described in SDC Commands. Additional SDC examples are shown in SDC Examples.

See also:
The Primitive Timing Modelling Tutorial which covers how to describe the timing characteristics of architecture primitives.

4.4.1 Default Timing Constraints

If no timing constraints are specified, VPR assumes default constraints based on the type of circuit being analyzed.

Combinational Circuits

Constrain all I/Os on a virtual clock virtual_io_clock, and optimize this clock to run as fast as possible.

Equivalent SDC File:

```verilog
create_clock -period 0 -name virtual_io_clock
set_input_delay -clock virtual_io_clock -max 0 [get_ports {*}]  
set_output_delay -clock virtual_io_clock -max 0 [get_ports {*}]  
```

Single-Clock Circuits

Constrain all I/Os on the netlist clock, and optimize this clock to run as fast as possible.

Equivalent SDC File:

```verilog
create_clock -period 0 *  
set_input_delay -clock * -max 0 [get_ports {*}] 
set_output_delay -clock * -max 0 [get_ports {*}]  
```
Multi-Clock Circuits

Constrain all I/Os a virtual clock `virtual_io_clock`. Does not analyse paths between netlist clock domains, but analyses all paths from I/Os to any netlist domain. Optimizes all clocks, including I/O clocks, to run as fast as possible.

**Warning:** By default VPR does not analyze paths between netlist clock domains.

Equivalent SDC File:

```vpr
create_clock -period 0 *
create_clock -period 0 -name virtual_io_clock
set_clock_groups -exclusive -group {clk} -group {clk2}
set_input_delay -clock virtual_io_clock -max 0 [get_ports {*}]}
set_output_delay -clock virtual_io_clock -max 0 [get_ports {*}]}
```

Where `clk` and `clk2` are the netlist clocks in the design. This is similarly extended if there are more than two netlist clocks.

### 4.5 SDC Commands

The following subset of SDC syntax is supported by VPR.

#### 4.5.1 create_clock

Creates a netlist or virtual clock.

Assigns a desired period (in nanoseconds) and waveform to one or more clocks in the netlist (if the `–name` option is omitted) or to a single virtual clock (used to constrain input and outputs to a clock external to the design). Netlist clocks can be referred to using regular expressions, while the virtual clock name is taken as-is.

**Example Usage:**

```vpr
#Create a netlist clock
create_clock -period <float> <netlist clock list or regexes>

#Create a virtual clock
create_clock -period <float> -name <virtual clock name>

#Create a netlist clock with custom waveform/duty-cycle
create_clock -period <float> -waveform {rising_edge falling_edge} <netlist clock list or regexes>
```

Omitting the waveform creates a clock with a rising edge at 0 and a falling edge at the half period, and is equivalent to using `-waveform {0 <period/2>}`. Non-50% duty cycles are supported but behave no differently than 50% duty cycles, since falling edges are not used in analysis. If a virtual clock is assigned using a create_clock command, it must be referenced elsewhere in a set_input_delay or set_output_delay constraint.

```vpr
create_clock
    -period <float>
        Specifies the clock period.

    Required: Yes
```
-waveform {<float> <float>}
  Overrides the default clock waveform.
  The first value indicates the time the clock rises, the second the time the clock falls.
  
  Required: No
  Default: 50% duty cycle (i.e. -waveform {0 <period/2>}).

-name <string>
  Creates a virtual clock with the specified name.
  
  Required: No

<netlist clock list or regexes>
  Creates a netlist clock
  
  Required: No

Note: One of -name or <netlist clock list or regexes> must be specified.

Warning: If a netlist clock is not specified with a create_clock command, paths to and from that clock domain will not be analysed.

4.5.2 set_clock_groups

Specifies the relationship between groups of clocks. May be used with netlist or virtual clocks in any combination.

Since VPR supports only the -exclusive option, a set_clock_groups constraint is equivalent to a set_false_path constraint (see below) between each clock in one group and each clock in another.

For example, the following sets of commands are equivalent:

```text
#Do not analyze any timing paths between clk and clk2, or between
#clk and clk3
set_clock_groups -exclusive -group {clk} -group {clk2 clk3}
```

and

```text
set_false_path -from [get_clocks {clk}] -to [get_clocks {clk2 clk3}]
set_false_path -from [get_clocks {clk2 clk3}] -to [get_clocks {clk}]
```

set_clock_groups
   -exclusive
     Indicates that paths between clock groups should not be analyzed.
     
     Required: Yes

Note: VPR currently only supports exclusive clock groups

   -group {<clock list or regexes>}
     Specifies a group of clocks.
Note: At least 2 groups must be specified.

Required: Yes

### 4.5.3 set_false_path

Cuts timing paths unidirectionally from each clock in -from to each clock in -to. Otherwise equivalent to `set_clock_groups`.

**Example Usage:**

```plaintext
#Do not analyze paths launched from clk and captured by clk2 or clk3
set_false_path -from [get_clocks {clk}] -to [get_clocks {clk2 clk3}]

#Do not analyze paths launched from clk2 or clk3 and captured by clk
set_false_path -from [get_clocks {clk2 clk3}] -to [get_clocks {clk}]
```

Note: False paths are supported between entire clock domains, but not between individual registers.

### 4.5.4 set_max_delay/set_min_delay

Overrides the default setup (max) or hold (min) timing constraint calculated using the information from `create_clock` with a user-specified delay.

**Example Usage:**

```plaintext
#Specify a maximum delay of 17 from input_clk to output_clk
set_max_delay 17 -from [get_clocks {input_clk}] -to [get_clocks {output_clk}]

#Specify a minimum delay of 2 from input_clk to output_clk
set_min_delay 2 -from [get_clocks {input_clk}] -to [get_clocks {output_clk}]
```

Note: Max/Min delays are supported between entire clock domains, but not between individual netlist elements.
<delay>
    The delay value to apply.
    Required: Yes

-from [get_clocks <clock list or regexes>]
    Specifies the source clock domain(s).
    Required: No
    Default: All clocks

-to [get_clocks <clock list or regexes>]
    Specifies the sink clock domain(s).
    Required: No
    Default: All clocks

4.5.5 set_multicycle_path

Sets how many clock cycles elapse between the launch and capture edges for setup and hold checks.

The default the setup multicycle value is 1 (i.e. the capture setup check is performed against the edge one cycle after
the launch edge).

The default hold multicycle is one less than the setup multicycle path (e.g. the capture hold check occurs in the same
cycle as the launch edge for the default setup multicycle).

Example Usage:

```
#Create a 4 cycle setup check, and 0 cycle hold check from clkA to clkB
set_multicycle_path -from [get_clocks {clkA}] -to [get_clocks {clkB}] 4

#Create a 3 cycle setup check from clk to clk2
# Note that this moves the default hold check to be 2 cycles
set_multicycle_path -setup -from [get_clocks {clk}] -to [get_clocks {clk2}] 3

#Create a 0 cycle hold check from clk to clk2
# Note that this moves the default hold check back to it's original
# position before the previous setup setup_multicycle_path was applied
set_multicycle_path -hold -from [get_clocks {clk}] -to [get_clocks {clk2}] 2

#Create a multicycle to a specific pin
set_multicycle_path -to [get_pins {my_inst.in\[0\]}] 2
```

Note: Multicycles are supported between entire clock domains, and ending at specific registers.

set_multicycle_path
    -setup
        Indicates that the multicycle-path applies to setup analysis.
        Required: No

    -hold
        Indicates that the multicycle-path applies to hold analysis.
        Required: No
-from [get_clocks <clock list or regexes>]
  Specifies the source clock domain(s).
  
  Required: No
  Default: All clocks

-to [get_clocks <clock list or regexes>]
  Specifies the sink clock domain(s).
  
  Required: No
  Default: All clocks

-to [get_pins <pin list or regexes>]
  Specifies the sink/capture netlist pins to which the multicycle is applied.
  
  See also:
  VPR’s pin naming convention.
  
  Required: No
  
<path_multiplier>
  The number of cycles that apply to the specified path(s).
  
  Required: Yes

Note: If neither -setup nor -hold the setup multicycle is set to path_multiplier and the hold multicycle offset to 0.

Note: Only a single -to option can be specified (either clocks or pins, but not both).

4.5.6 set_input_delay/set_output_delay

Use set_input_delay if you want timing paths from input I/Os analyzed, and set_output_delay if you want timing paths to output I/Os analyzed.

Note: If these commands are not specified in your SDC, paths from and to I/Os will not be timing analyzed.

These commands constrain each I/O pad specified after get_ports to be timing-equivalent to a register clocked on the clock specified after -clock. This can be either a clock signal in your design or a virtual clock that does not exist in the design but which is used only to specify the timing of I/Os.

The specified delays are added to I/O timing paths and can be used to model board level delays.

For single-clock circuits, -clock can be wildcarded using * to refer to the single netlist clock, although this is not supported in standard SDC. This allows a single SDC command to constrain I/Os in all single-clock circuits.

Example Usage:

```
#Set a maximum input delay of 0.5 (relative to input_clk) on
#ports in1, in2 and in3
set_input_delay -clock input_clk -max 0.5 [get_ports {in1 in2 in3}]
```

(continues on next page)
set_input_delay/set_output_delay
    -clock <virtual or netlist clock>
        Specifies the virtual or netlist clock the delay is relative to.
        Required: Yes

    -max
        Specifies that the delay value should be treated as the maximum delay.
        Required: No

    -min
        Specifies that the delay value should be treated as the minimum delay.
        Required: No

    <delay>
        Specifies the delay value to be applied
        Required: Yes

    [get_ports <|I/O list or regexes|>]
        Specifies the port names or port name regex.
        Required: Yes

    Note: If neither -min nor -max are specified the delay value is applied to both.

4.5.7 set_clock_uncertainty

Sets the clock uncertainty between clock domains. This is typically used to model uncertainty in the clock arrival times due to clock jitter.

Example Usage:

#Sets the clock uncertainty between all clock domain pairs to 0.025
set_clock_uncertainty 0.025

#Sets the clock uncertainty from 'clk' to all other clock domains to 0.05
set_clock_uncertainty -from [get_clocks {clk}] 0.05

#Sets the clock uncertainty from 'clk' to 'clk2' to 0.75
set_clock_uncertainty -from [get_clocks {clk}] -to [get_clocks {clk2}] 0.75

set_clock_uncertainty
    -from [get_clocks <clock list or regexes>]
        Specifies the source clock domain(s).
Required: No
Default: All clocks
-to [get_clocks <clock list or regexes>]
Specifies the sink clock domain(s).
Required: No
Default: All clocks

-setup
Specifies the clock uncertainty for setup analysis.
Required: No

-hold
Specifies the clock uncertainty for hold analysis.
Required: No

<uncertainty>
The clock uncertainty value between the from and to clocks.
Required: Yes

Note: If neither -setup nor -hold are specified the uncertainty value is applied to both.

4.5.8 set_clock_latency

Sets the latency of a clock. VPR automatically calculates on-chip clock network delay, and so only source latency is supported.

Source clock latency corresponds to the delay from the true clock source (e.g. off-chip clock generator) to the on-chip clock definition point.

#Sets the source clock latency of 'clk' to 1.0
set_clock_latency -source 1.0 [get_clocks {clk}]

set_clock_latency
-source
Specifies that the latency is the source latency.
Required: Yes

-early
Specifies that the latency applies to early paths.
Required: No

-late
Specifies that the latency applies to late paths.
Required: No

<latency>
The clock’s latency.
Required: Yes
[get_clocks <clock list or regexes>]
   Specifies the clock domain(s).
   Required: Yes

Note: If neither -early nor -late are specified the latency value is applied to both.

4.5.9 set_disable_timing

Disables timing between a pair of connected pins in the netlist. This is typically used to manually break combinational loops.

```
#Disables the timing edge between the pins 'FFA.Q[0]' and 'to_FFD.in[0]' on
set_disable_timing -from [get_pins {FFA.Q\[0\]}] -to [get_pins {to_FFD.in\[0\]}]
```

set_disable_timing
   -from [get_pins <pin list or regexes>]
      Specifies the source netlist pins.
      See also:
      VPR's pin naming convention.
      Required: Yes
   -to [get_pins <pin list or regexes>]
      Specifies the sink netlist pins.
      See also:
      VPR's pin naming convention.
      Required: Yes

Note: Make sure to escape the characters in the regexes.

4.5.10 Special Characters

# (comment), \ (line continued), * (wildcard), {} (string escape)
   # starts a comment – everything remaining on this line will be ignored.
   \ at the end of a line indicates that a command wraps to the next line.
   * is used in a get_clocks/get_ports command or at the end of create_clock to match all netlist clocks. Partial wildcarding (e.g. clk* to match clk and clk2) is also supported. As mentioned above, * can be used in set_input_delay and set_output_delay to refer to the netlist clock for single-clock circuits only, although this is not supported in standard SDC.
   {} escapes strings, e.g. {top^clk} matches a clock called top^clk, while top^clk without braces gives an error because of the special ^ character.
4.5.11 SDC Examples

The following are sample SDC files for common non-default cases (assuming netlist clock domains `clk` and `clk2`).

A

Cut I/Os and analyse only register-to-register paths, including paths between clock domains; optimize to run as fast as possible.

```
create_clock -period 0 *
```

B

Same as A, but with paths between clock domains cut. Separate target frequencies are specified.

```
create_clock -period 2 clk
create_clock -period 3 clk2
set_clock_groups -exclusive -group {clk} -group {clk2}
```

C

Same as B, but with paths to and from I/Os now analyzed. This is the same as the multi-clock default, but with custom period constraints.

```
create_clock -period 2 clk
create_clock -period 3 clk2
create_clock -period 3.5 -name virtual_io_clock
set_clock_groups -exclusive -group {clk} -group {clk2}
set_input_delay -clock virtual_io_clock -max 0 [get_ports *]
set_output_delay -clock virtual_io_clock -max 0 [get_ports *]
```

D

Changing the phase between clocks, and accounting for delay through I/Os with set_input/output delay constraints.

```
#Custom waveform rising edge at 1.25, falling at 2.75
create_clock -period 3 -waveform {1.25 2.75} clk
create_clock -period 2 clk2
create_clock -period 2.5 -name virtual_io_clock
set_input_delay -clock virtual_io_clock -max 1 [get_ports *]
set_output_delay -clock virtual_io_clock -max 0.5 [get_ports *]
```
Sample using many supported SDC commands. Inputs and outputs are constrained on separate virtual clocks.

```verilog
create_clock -period 3 -waveform {1.25 2.75} clk
create_clock -period 2 clk2
create_clock -period 1 -name input_clk
create_clock -period 0 -name output_clk
set_clock_groups -exclusive -group input_clk -group clk2
set_false_path -from [get_clocks {clk}] -to [get_clocks {output_clk}]
set_max_delay 17 -from [get_clocks {input_clk}] -to [get_clocks {output_clk}]
set_multicycle_path -setup -from [get_clocks {clk}] -to [get_clocks {clk2}] 3
set_input_delay -clock input_clk -max 0.5 [get_ports {in1 in2 in3}]
set_output_delay -clock output_clk -max 1 [get_ports {out*}]
```

Sample using all remaining SDC commands.

```verilog
create_clock -period 3 -waveform {1.25 2.75} clk
create_clock -period 2 clk2
create_clock -period 1 -name input_clk
create_clock -period 0 -name output_clk
set_clock_latency -source 1.0 [get_clocks{clk}]
# if neither early nor late is specified then the latency applies to early paths
set_clock_groups -exclusive -group input_clk -group clk2
set_false_path -from [get_clocks{clk}] -to [get_clocks{output_clk}]
set_input_delay -clock input_clk -max 0.5 [get_ports{in1 in2 in3}]
set_output_delay -clock output_clk -min 1 [get_ports{out*}]
set_max_delay 17 -from [get_clocks{input_clk}] -to [get_clocks{output_clk}]
set_min_delay 2 -from [get_clocks{input_clk}] -to [get_clocks{output_clk}]
set_multicycle_path -setup -from [get_clocks{clk}] -to [get_clocks{clk2}] 3
# For multicycle_path, if setup is specified then hold is also implicitly specified
set_clock_uncertainty -from [get_clocks{clk}] -to [get_clocks{clk2}] 0.75
# For set_clock_uncertainty, if neither setup nor hold is unspecified then uncertainty is applied to both
set_disable_timing -from [get_pins {FFA.Q\[0\\}}} -to [get_pins {to_FFD.in\[0\\}]}
```
4.6 File Formats

VPR consumes and produces several files representing the packing, placement, and routing results.

4.6.1 FPGA Architecture (.xml)

The target FPGA architecture is specified as an architecture file. For details of this file format see FPGA Architecture Description.

4.6.2 BLIF Netlist (.blif)

The technology mapped circuit to be implement on the target FPGA is specified as a Berkely Logic Interchange Format (BLIF) netlist. The netlist must be flattened and consist of only primitives (e.g. .names, .latch, .subckt).

For a detailed description of the BLIF file format see the BLIF Format Description.

Note that VPR supports only the structural subset of BLIF, and does not support the following BLIF features:

- Subfile References (.search).
- Finite State Machine Descriptions (.start_kiss, .end_kiss etc.).
- Clock Constraints (.cycle, .clock_event).
- Delay Constraints (.delay etc.).

Clock and delay constraints can be specified with an SDC File.

Note: By default VPR assumes file with .blif are in structural BLIF format. The format can be controlled with vpr --circuit_format.

Black Box Primitives

Black-box architectural primitives (RAMs, Multipliers etc.) should be instantiated in the netlist using BLIF’s .subckt directive. The BLIF file should also contain a black-box .model definition which defines the input and outputs of each .subckt type.

VPR will check that blackbox .models are consistent with the <models> section of the architecture file.

Unconnected Primitive Pins

Unconnected primitive pins can be specified through several methods.

1. The unnconn net (input pins only).

VPR treats any input pin connected to a net named unnconn as disconnected.

For example:

```
.names unnconn out
0 1
```
specifies an inverter with no connected input.

**Note:** `unconn` should only be used for **input pins**. It may cause name conflicts and create multi-driven nets if used with output pins.

2. Implicitly disconnected `.subckt` pins.

For `.subckt` instantiations, VPR treats unlisted primitive pins as implicitly disconnected. This works for both input and output pins.

For example, the following `.subckt` instantiations are equivalent:

```
.subckt single_port_ram
  clk=top^clk
  data=top.memory_controller+memtroll^MULTI_PORT_MUX~8^MUX_2~546
  addr[0]=top.memory_controller+memtroll^MULTI_PORT_MUX~8^MUX_2~541
  addr[1]=top.memory_controller+memtroll^MULTI_PORT_MUX~8^MUX_2~542
  addr[2]=top.memory_controller+memtroll^MULTI_PORT_MUX~8^MUX_2~543
  addr[3]=top.memory_controller+memtroll^MULTI_PORT_MUX~8^MUX_2~544
  addr[4]=top.memory_controller+memtroll^MULTI_PORT_MUX~8^MUX_2~545
  addr[5]=unconn
  addr[6]=unconn
  addr[7]=unconn
  addr[8]=unconn
  addr[9]=unconn
  addr[10]=unconn
  addr[12]=unconn
  addr[13]=unconn
  addr[14]=unconn
  we=top.memory_controller+memtroll^MULTI_PORT_MUX~8^MUX_2~554
  out=top.memory_controller+memtroll.single_port_ram+str^out~0
```

```
.subckt single_port_ram
  clk=top^clk
  data=top.memory_controller+memtroll^MULTI_PORT_MUX~8^MUX_2~546
  addr[0]=top.memory_controller+memtroll^MULTI_PORT_MUX~8^MUX_2~541
  addr[1]=top.memory_controller+memtroll^MULTI_PORT_MUX~8^MUX_2~542
  addr[2]=top.memory_controller+memtroll^MULTI_PORT_MUX~8^MUX_2~543
  addr[3]=top.memory_controller+memtroll^MULTI_PORT_MUX~8^MUX_2~544
  addr[4]=top.memory_controller+memtroll^MULTI_PORT_MUX~8^MUX_2~545
  we=top.memory_controller+memtroll^MULTI_PORT_MUX~8^MUX_2~554
  out=top.memory_controller+memtroll.single_port_ram+str^out~0
```

3. Dummy nets with no sinks (output pins only)

By default, VPR sweeps away nets with no sinks (see `vpr --sweep_dangling_nets`). As a result, output pins can be left ‘disconnected’ by connecting them to dummy nets.

For example:

```
.names in dummy_net1
0 1
```

specifies an inverter with no connected output (provided `dummy_net1` is connected to no other pins).
Note: This method requires that every disconnected output pin should be connected to a uniquely named dummy net.

**BLIF File Format Example**

The following is an example BLIF file. It implements a 4-bit ripple-carry adder and some simple logic.

The main `.model` is named `top`, and its input and output pins are listed using the `.inputs` and `.outputs` directives.

The 4-bit ripple-carry adder is built of 1-bit `adder` primitives which are instantiated using the `.subckt` directive. Note that the adder primitive is defined as its own `.model` (which describes its pins), and is marked as `.blackbox` to indicate it is an architectural primitive.

The signal `all_sum_high_comb` is computed using combinational logic (`.names`) which ANDs all the sum bits together.

The `.latch` directive instantiates a rising-edge (re) latch (i.e. an edge-triggered Flip-Flop) clocked by `clk`. It takes in the combinational signal `all_sum_high_comb` and drives the primary output `all_sum_high_reg`.

Also note that the last `.subckt adder` has its `cout` output left implicitly disconnected.

```verilog
.model top
.names gnd
  0

.subckt adder a=a[0] b=b[0] cin=gnd cout=cin[1] sumout=sum[0]

 1111 1

.latch all_sum_high_comb all_sum_high_reg re clk 0

.end

.model adder
  .inputs a b cin
  .outputs cout sumout
  .blackbox
.end
```

4.6. File Formats 183
BLIF Naming Convention

VPR follows a naming convention to refer to primitives and pins in the BLIF netlist. These names appear in the VPR GUI, in log and error messages, and in can be used elsewhere (e.g. in SDC constraints).

Net Names

The BLIF format uses explicit names to refer to nets. These names are used directly as is by VPR (although some nets may be merged/removed by netlist cleaning).

For example, the following netlist:

```
.model top
.inputs a b
.outputs c
.names a b c
11 1
.end
```

contains nets named:

- a
- b
- c

Primitive Names

The standard BLIF format has no mechanism for specifying the names of primitives (e.g. .names/.latch/.subckt).

As a result, tools processing BLIF follow a naming convention which generates unique names for each netlist primitive.

The VPR primitive naming convention is as follows:

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Drives at least one net?</th>
<th>Primitive Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>.input</td>
<td>Yes</td>
<td>Name of first driven net</td>
</tr>
<tr>
<td>.names</td>
<td>No</td>
<td>Arbitrarily generated (e.g.</td>
</tr>
<tr>
<td>.latch</td>
<td></td>
<td>unnamed_instances_K)</td>
</tr>
<tr>
<td>.subckt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.output</td>
<td>N/A</td>
<td>.output name prefixed with out:</td>
</tr>
</tbody>
</table>

which ensures each netlist primitive is given a unique name.

For example, in the following:

```
.model top
.inputs a b x y z clk
.outputs c c_reg cout[0] sum[0]
.names a b c
11 1
```

(continues on next page)
The circuit primary inputs (.inputs) are named: a, b, x, y, z, clk.

• The 2-LUT (.names) is named c,

• The FF (.latch) is named c_reg,

• The adder (.subckt) is named cout[0] (the name of the first net it drives), and

• The circuit primary outputs (.outputs) are named: out:c, out:c_reg, out:cout[0], out:sum[0].

See also:
EBLIF’s .cname extension, which allows explicit primitive names to be specified.

**Pin Names**

It is useful to be able to refer to particular pins in the netlist. VPR uses the convention: <primitive_instance_name>.<pin_name>. Where <primitive_instance_name> is replaced with the netlist primitive name, and <pin_name> is the name of the relevant pin.

For example, the following adder:

```
.subckt adder a=x b=y cin=z cout=cout[0] sumout=sum[0]
```

which has pin names:

• cout[0].a[0] (driven by net x)
• cout[0].b[0] (driven by net y)
• cout[0].cin[0] (driven by net z)
• cout[0].cout[0] (drives net cout[0])
• cout[0].sumout[0] (drives net sum[0])

Since the primitive instance itself is named cout[0] by convention.


**Built-in Primitive Pin Names**

The built-in primitives in BLIF (.names, .latch) do not explicitly list the names of their input/output pins. VPR uses the following convention:

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Port</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>.names</td>
<td>input</td>
<td>in</td>
</tr>
<tr>
<td></td>
<td>output</td>
<td>out</td>
</tr>
<tr>
<td>.latch</td>
<td>input</td>
<td>D</td>
</tr>
<tr>
<td></td>
<td>output</td>
<td>Q</td>
</tr>
<tr>
<td></td>
<td>control</td>
<td>clk</td>
</tr>
</tbody>
</table>

Consider the following:

```
.names a b c d e f
11111 1
.latch g h re clk 0
```

The .names’ pin names are:
- f.in[0] (driven by net a)
- f.in[1] (driven by net b)
- f.in[2] (driven by net c)
- f.in[3] (driven by net d)
- f.in[4] (driven by net e)
- f.out[0] (drives net f)

and the .latch pin names are:
- h.D[0] (driven by net g)
- h.Q[0] (drives net h)
- h.clk[0] (driven by net clk)

since the .names and .latch primitives are named f and h *by convention*.

Note: To support pins within multi-bit ports unambiguously, the bit index of the pin within its associated port is included in the pin name (for single-bit ports this will always be [0]).

### 4.6.3 Extended BLIF (.eblif)

VPR also supports several extensions to *structural BLIF* to address some of its limitations.

Note: By default VPR assumes file with .eblif are in extended BLIF format. The format can be controlled with `vpr --circuit_format`. 
.conn

The .conn statement allows direct connections between two wires.

For example:

```
.model top
.input a
.output b

#Direct connection
.conn a b
.end
```

specifies that 'a' and 'b' are direct connected together. This is analogous to Verilog's `assign b = a;`.

This avoids the insertion of a .names buffer which is required in standard BLIF, for example:

```
.model top
.input a
.output b

#Buffer LUT required in standard BLIF
.names a b
 1 1
.end
```

.cname

The .cname statement allows names to be specified for BLIF primitives (e.g. .latch, .names, .subckt).

**Note:** .cname statements apply to the previous primitive instantiation.

For example:

```
.names a b c
11 1
.cname my_and_gate
```

Would name of the above .names instance my_and_gate.
.param

The .param statement allows parameters (e.g. primitive modes) to be tagged on BLIF primitives.

**Note:** .param statements apply to the previous primitive instantiation.

For example:

```
.subckt dsp a=a_in b=b_in cin=c_in cout=c_out s=sum_out
.param mode adder
```

Would set the parameter mode of the above dsp .subckt to adder.

.param statements propagate to <parameter> elements in the packed netlist.

.attr

The .attr statement allows attributes (e.g. source file/line) to be tagged on BLIF primitives.

**Note:** .attr statements apply to the previous primitive instantiation.

For example:

```
.latch a_and_b dff_q re clk 0
.attr src my_design.v:42
```

Would set the attribute src of the above .latch to my_design.v:42.

.attr statements propagate to <attribute> elements in the packed netlist.

**Extended BLIF File Format Example**

```
.model top
.inputs a b clk
.outputs o_dff

.names a b a_and_b
11 1
.cname lut_a_and_b
.param test_names_param "test_names_param_value"
.attr test_names_attrib "test_names_param_attrib"

.latch a_and_b dff_q re clk 0
.cname mydff
.param test_latch_param "test_latch_param_value"
.attr test_latch_attrib "test_latch_param_attrib"

.conn dff_q o_dff
.end
```
4.6.4 Timing Constraints (.sdc)

Timing constraints are specified using SDC syntax. For a description of VPR’s SDC support see SDC Commands.

Note: Use vpr --sdc_file to specify the SDC file used by VPR.

Timing Constraints File Format Example

See SDC Examples.

4.6.5 Packed Netlist Format (.net)

The circuit .net file is an xml file that describes a post-packed user circuit. It represents the user netlist in terms of the complex logic blocks of the target architecture. This file is generated from the packing stage and used as input to the placement stage in VPR.

The .net file is constructed hierarchically using block tags. The top level block tag contains the I/Os and complex logic blocks used in the user circuit. Each child block tag of this top level tag represents a single complex logic block inside the FPGA. The block tags within a complex logic block tag describes, hierarchically, the clusters/modes/primitives used internally within that logic block.

A block tag has the following attributes:

- **name** A name to identify this component of the FPGA. This name can be completely arbitrary except in two situations. First, if this is a primitive (leaf) block that implements an atom in the input technology-mapped netlist (eg. LUT, FF, memory slice, etc), then the name of this block must match exactly with the name of the atom in that netlist so that one can later identify that mapping. Second, if this block is not used, then it should be named with the keyword open. In all other situations, the name is arbitrary.

- **instance** The physical block in the FPGA architecture that the current block represents. Should be of format: architecture_instance_name[instance #]. For example, the 5th index BLE in a CLB should have instance="ble[5]"

- **mode** The mode the block is operating in.

A block connects to other blocks via pins which are organized based on a hierarchy. All block tags contains the children tags: inputs, outputs, clocks. Each of these tags in turn contain port tags. Each port tag has an attribute name that matches with the name of a corresponding port in the FPGA architecture. Within each port tag is a list of named connections where the first name corresponds to pin 0, the next to pin 1, and so forth. The names of these connections use the following format:

1. Unused pins are identified with the keyword open.
2. The name of an input pin to a complex logic block is the same as the name of the net using that pin.
3. The name of an output pin of a primitive (leaf block) is the same as the name of the net using that pin.
4. The names of all other pins are specified by describing their immediate drivers. This format is [name_of_immediate_driver_block].[port_name][pin#]->interconnect_name.

For primitives with equivalent inputs VPR may rotate the input pins. The resulting rotation is specified with the <port_rotation_map> tag. For example, consider a netlist contains a 2-input LUT named c, which is implemented in a 5-LUT:
Listing 4.1: Example of `<port_rotation_map>` tag.

```xml
... 
  <block name="c" instance="lut[0]">
    <inputs>
      <port name="in">open open lut5.in[2]->direct:lut5 open lut5.in[4]->direct:lut5</port>
      <port_rotation_map name="in">open open 1 open 0</port_rotation_map>
    </inputs>
    <outputs>
      <port name="out">c</port>
    </outputs>
    <clocks>
  </clocks>
  </block>
...
```

In the original netlist the two LUT inputs were connected to pins at indicies 0 and 1 (the only input pins). However during clustering the inputs were rotated, and those nets now connect to the pins at indicies 2 and 4 (line 4). The `<port_rotation_map>` tag specified the port name it applies to (name attribute), and its contents lists the pin indicies each pin in the port list is associated with in the original netlist (i.e. the pins lut5.in[2]->direct:lut5 and lut5.in[4]->direct:lut5 respectively correspond to indicies 1 and 0 in the original netlist).

**Note:** Use `vpr --net_file` to override the default net file name.

### Packing File Format Example

The following is an example of what a .net file would look like. In this circuit there are 3 inputs (pa, pb, pc) and 4 outputs (out:pd, out:pe, out:pf, out:pg). The io pad is set to inpad mode and is driven by the inpad:

Listing 4.2: Example packed netlist file (trimmed for brevity).

```xml
... 
  <block name="b1.net" instance="FPGA_packed_netlist[0]">
    <inputs>
      pa pb pc
    </inputs>
    <outputs>
      out:pd out:pe out:pf out:pg
    </outputs>
  </block>
  <block name="pa" instance="io[0]" mode="inpad">
    <inputs>
      <port name="outpad">open</port>
    </inputs>
    <outputs>
      <port name="inpad">inpad[0].inpad[0]->inpad</port>
    </outputs>
  </block>
... 
```

(continues on next page)
Note: .net files may be outputted at two stages: - After packing is completed, the packing results will be outputted. The .net file can be loaded as an input for placer, router and analyzer. Note that the file may not represent the final packing results as the analyzer will apply synchronization between packing and routing results. - After analysis is completed, updated packing results will be outputted. This is due to that VPR router may swap pin mapping in packing results for optimizations. In such cases, packing results are synchronized with routing results. The outputted .net file will have a postfix of .post_routing as compared to the original packing results. It could happen that VPR router does not apply any pin swapping and the two .net files are the same. In both cases, the post-analysis .net file should be considered to be the final packing results for downstream tools, e.g., bitstream generator. Users may load the post-routing .net file in VPR’s analysis flow to sign-off the final results.

**Warning:** Currently, the packing result synchronization is only applicable to input pins which may be remapped to different nets during routing optimization. If your architecture defines link_instance_pin_xml_syntax_equivalence for output pins, the packing results still mismatch the routing results!
4.6.6 Placement File Format (.place)

The first line of the placement file lists the netlist (.net) and architecture (.xml) files used to create this placement. This information is used to ensure you are warned if you accidentally route this placement with a different architecture or netlist file later. The second line of the file gives the size of the logic block array used by this placement. All the following lines have the format:

<table>
<thead>
<tr>
<th>block_name</th>
<th>x</th>
<th>y</th>
<th>subtile_number</th>
</tr>
</thead>
</table>

The `block_name` is the name of this block, as given in the input .net formatted netlist. x and y are the row and column in which the block is placed, respectively.

**Note:** The blocks in a placement file can be listed in any order.

Since we can have more than one block in a row or column when the block capacity is set to be greater than 1 in the architecture file, the subtile number specifies which of the several possible subtile locations in row x and column y contains this block. Note that the subtile number used should be in the range 0 to (grid[i][j].capacity - 1). The subtile numbers for a particular x,y location do not have to be used in order.

The placement files output by VPR also include (as a comment) a fifth field: the block number. This is the internal index used by VPR to identify a block – it may be useful to know this index if you are modifying VPR and trying to debug something.

---

**Fig. 4.9:** FPGA co-ordinate system.

**Fig. 4.9** shows the coordinate system used by VPR for a small 2 x 2 CLB FPGA. The number of CLBs in the x and y directions are denoted by nx and ny, respectively. CLBs all go in the area with x between 1 and nx and y between 1 and ny, inclusive. All pads either have x equal to 0 or nx + 1 or y equal to 0 or ny + 1.

**Note:** Use `vpr --place_file` to override the default place file name.
Placement File Format Example

An example placement file is:

Listing 4.3: Example placement file.

```
Netlist file: xor5.net  Architecture file: sample.xml
Array size: 2 x 2 logic blocks

#block name x  y  subblk block number
#---------- -- -- ------- -----------
a  0  1  0 #0 -- NB: block number is a comment.
b  1  0  0 #1
c  0  2  1 #2
d  1  3  0 #3
e  1  3  1 #4
out:xor5  0  2  0 #5
xor5  1  2  0 #6
[1]  1  1  0 #7
```

4.6.7 Routing File Format (.route)

The first line of the routing file gives the array size, nx x ny. The remainder of the routing file lists the global or the detailed routing for each net, one by one. Each routing begins with the word net, followed by the net index used internally by VPR to identify the net and, in brackets, the name of the net given in the netlist file. The following lines define the routing of the net. Each begins with a keyword that identifies a type of routing segment. The possible keywords are SOURCE (the source of a certain output pin class), SINK (the sink of a certain input pin class), OPIN (output pin), IPIN (input pin), CHANX (horizontal channel), and CHANY (vertical channel). Each routing begins on a SOURCE and ends on a SINK. In brackets after the keyword is the (x, y) location of this routing resource. Finally, the pad number (if the SOURCE, SINK, IPIN or OPIN was on an I/O pad), pin number (if the IPIN or OPIN was on a clb), class number (if the SOURCE or SINK was on a clb) or track number (for CHANX or CHANY) is listed – whichever one is appropriate. The meaning of these numbers should be fairly obvious in each case. If we are attaching to a pad, the pad number given for a resource is the subblock number defining to which pad at location (x, y) we are attached. See Fig. 4.9 for a diagram of the coordinate system used by VPR. In a horizontal channel (CHANX) track 0 is the bottommost track, while in a vertical channel (CHANY) track 0 is the leftmost track. Note that if only global routing was performed the track number for each of the CHANX and CHANY resources listed in the routing will be 0, as global routing does not assign tracks to the various nets.

For an N-pin net, we need N-1 distinct wiring “paths” to connect all the pins. The first wiring path will always go from a SOURCE to a SINK. The routing segment listed immediately after the SINK is the part of the existing routing to which the new path attaches.

**Note:** It is important to realize that the first pin after a SINK is the connection into the already specified routing tree; when computing routing statistics be sure that you do not count the same segment several times by ignoring this fact.

**Note:** Use `vpr --route_file` to override the default route file name.
Routing File Format Examples

An example routing for one net is listed below:

```
Listing 4.4: Example routing for a non-global net.

Net 5 (xor5)

Node: 1 SOURCE (1,2) Class: 1 Switch: 1  # Source for pins of class 1.
Node: 2 OPIN (1,2) Pin: 4 clb.0[12] Switch:0  # Output pin the 0 port of clb..
   ---block, pin number 12
Node: 4 CHANX (1,1) to (4,1) Track: 1 Switch: 1
Node: 6 CHANX (4,1) to (7,1) Track: 1 Switch: 1
Node: 8 IPIN (7,1) Pin: 0 clb.I[0] Switch: 2
Node: 9 SINK (7,1) Class: 0 Switch: -1  # Sink for pins of class 0 on a clb.
Node: 4 CHANX (7,1) to (10,1) Track: 1 Switch: 1  # Note: Connection to
   ---existing routing!
Node: 5 CHANY (10,1) to (10,4) Track: 1 Switch: 0
Node: 4 CHANX (10,4) to (13,4) Track: 1 Switch: 1
Node: 10 CHANX (13,4) to (16,4) Track: 1 Switch: 1
Node: 11 IPIN (16,4) Pad: 1 clb.I[1] Switch: 2
Node: 12 SINK (16,4) Pad: 1 Switch: -1  # This sink is an output pad at (16,4),
   ---subblock 1.
```

Nets which are specified to be global in the netlist file (generally clocks) are not routed. Instead, a list of the blocks
(name and internal index) which this net must connect is printed out. The location of each block and the class of the pin
to which the net must connect at each block is also printed. For clbs, the class is simply whatever class was specified
for that pin in the architecture input file. For pads the pinclass is always -1; since pads do not have logically-equivalent
pins, pin classes are not needed. An example listing for a global net is given below.

```
Listing 4.5: Example routing for a global net.

Net 146 (pclk): global net connecting:
Block pclk (#146) at (1,0), pinclass -1
Block pksi_17_ (#431) at (3,26), pinclass 2
Block pksi_185_ (#432) at (5,48), pinclass 2
Block n_n2879 (#433) at (49,23), pinclass 2
```

4.6.8 Routing Resource Graph File Format (.xml)

The routing resource graph (rr graph) file is an XML file that describes the routing resources within the FPGA. This
file is generated through the last stage of the rr graph generation during routing with the final channel width. When
reading in rr graph from an external file, the rr graph is used during the placement and routing section of VPR. The
file is constructed using tags. The top level is the rr_graph tag. This tag contains all the channel, switches, segments,
block, grid, node, and edge information of the FPGA. It is important to keep all the values as high precision as possible.
Sensitive values include capacitance and Tdel. As default, these values are printed out with a precision of 30 digits.
Each of these sections are separated into separate tags as described below.

Note: Use vpr --read_rr_graph to specify an RR graph file to be load.
Note: Use `vpr --write_rr_graph` to specify where the RR graph should be written.

**Top Level Tags**

The first tag in all rr graph files is the `<rr_graph>` tag that contains detailed subtags for each category in the rr graph. Each tag has their subsequent subtags that describes one entity. For example, `<segments>` includes all the segments in the graph where each `<segment>` tag outlines one type of segment.

The `rr_graph` tag contains the following tags:

- `<channels>`
  - `<channel>`
- `<switches>`
  - `<switch>`
- `<segments>`
  - `<segment>`
- `<block_types>`
  - `<block_type>`
- `<grid>`
  - `<grid_loc>`
- `<rr_nodes>`
  - `<node>`
- `<rr_edges>`
  - `<edge>`

**Note:** The rr graph is based on the architecture, so more detailed description of each section of the rr graph can be found at [FPGA architecture description](#).

**Detailed Tag Information**

**Channel**

The channel information is contained within the `channels` subtag. This describes the minimum and maximum channel width within the architecture. Each `channels` tag has the following subtags:

```xml
<channel chan_width_max="int" x_min="int" y_min="int" x_max="int" y_max="int"/>
```

This is a required subtag that contains information about the general channel width information. This stores the channel width between x or y directed channels.

**Required Attributes**

- `chan_width_max` – Stores the maximum channel width value of x or y channels.
- `x_min y_min x_max y_max` – Stores the minimum and maximum value of x and y coordinate within the lists.
These are required subtags that list the contents of an `x_list` and `y_list` array which stores the width of each channel. The `x_list` array size as large as the size of the `y_dimension` of the FPGA itself while the `y_list` has the size of the `x_dimension`. This `x_list` tag is repeated for each index within the array.

**Required Attributes**

- **index** – Describes the index within the array.
- **info** – The width of each channel. The minimum is one track per channel. The input and output channels are `io_rat * maximum_in_interior_tracks` wide. The channel distributions read from the architecture file are scaled by a constant factor.

**Switches**

A `switches` tag contains all the switches and its information within the FPGA. It should be noted that for values such as capacitance, `Tdel`, and `sizing info` all have high precision. This ensures a more accurate calculation when reading in the routing resource graph. Each switch tag has a `switch` subtag.

```xml
<switch id="int" name="unique_identifier" type="{mux|tristate|pass_gate|short|buffer}">
    Required Attributes
    • **id** – A unique identifier for that type of switch.
    • **name** – An optional general identifier for the switch.
    • **type** – See *architecture switch description*.
</switch>

<timing R="float" cin="float" Cout="float" Tdel="float"/>
    This optional subtag contains information used for timing analysis. Without it, the program assumes all subtags to contain a value of 0.
    Optional Attributes
    • **R, Cin, Cout** – The resistance, input capacitance and output capacitance of the switch.
    • **Tdel** – Switch’s intrinsic delay. It can be outlined that the delay through an unloaded switch is `Tdel + R * Cout`.

<sizing mux_trans_size="int" buf_size="float"/>
    The sizing information contains all the information needed for area calculation.
    Required Attributes
    • **mux_trans_size** – The area of each transistor in the segment’s driving mux. This is measured in minimum width transistor units.
    • **buf_size** – The area of the buffer. If this is set to zero, the area is calculated from the resistance.
Segments

The `segments` tag contains all the segments and its information. Note again that the capacitance has a high decimal precision. Each segment is then enclosed in its own `segment` tag.

```xml
<segment id="int" name="unique_identifier">
  Required Attributes
  • `id` – The index of this segment.
  • `name` – The name of this segment.
  <timing R_per_meter="float" C_per_meter="float">
    This optional tag defines the timing information of this segment.
    Optional Attributes
    • `R_per_meter, C_per_meter` – The resistance and capacitance of a routing track, per unit logic block length.
  </timing>
</segment>
```

Blocks

The `block_types` tag outlines the information of a placeable complex logic block. This includes generation, pin classes, and pins within each block. Information here is checked to make sure it corresponds with the architecture. It contains the following subtags:

```xml
<block_type id="int" name="unique_identifier" width="int" height="int">
  This describes generation information about the block using the following attributes:
  Required Attributes
  • `id` – The index of the type of the descriptor in the array. This is used for index referencing
  • `name` – A unique identifier for this type of block. Note that an empty block type must be denoted "EMPTY" without the brackets <> to prevent breaking the xml format. Input and output blocks must be named “io”. Other blocks can have any name.
  • `width, height` – The width and height of a large block in grid tiles.
  <pin_class type="pin_type">
    This optional subtag of `block_type` describes groups of pins in configurable logic blocks that share common properties.
    Required Attributes
    • `type` – This describes whether the pin class is a driver or receiver. Valid inputs are OPEN, OUTPUT, and INPUT.
  </pin_class>
  <pin ptc="block_pin_index">name</pin>
  This required subtag of `pin_class` describes its pins.
  Required Attributes
  • `ptc` – The index of the pin within the `block_type`.
  • `name` – Human readable pin name.
</block_type>
```
Grid

The grid tag contains information about the grid of the FPGA. Information here is checked to make sure it corresponds with the architecture. Each grid tag has one subtag as outlined below:

```xml
<grid_loc x="int" y="int" block_type_id="int" width_offset="int" height_offset="int">
  Required Attributes
  • x, y – The x and y coordinate location of this grid tile.
  • block_type_id – The index of the type of logic block that resides here.
  • width_offset, height_offset – The number of grid tiles reserved based on the width and height of a block.
</grid_loc>
```

Nodes

The rr_nodes tag stores information about each node for the routing resource graph. These nodes describe each wire and each logic block pin as represented by nodes.

```xml
<node id="int" type="unique_type" direction="unique_direction" capacity="int">
  Required Attributes
  • id – The index of the particular routing resource node
  • type – Indicates whether the node is a wire or a logic block. Valid inputs for class types are { CHANX | CHANY | SOURCE | SINK | OPIN | IPIN }. Where CHANX and CHANY describe a horizontal and vertical channel. Sources and sinks describes where nets begin and end. OPIN represents an output pin and IPIN represent an input pin.
  • capacity – The number of routes that can use this node.

  Optional Attributes
  • direction – If the node represents a track (CHANX or CHANY), this field represents its direction as { INC_DIR | DEC_DIR | BI_DIR }. In other cases this attribute should not be specified.
</node>
```

```xml
<loc xlow="int" ylow="int" xhigh="int" yhigh="int" side="{LEFT|RIGHT|TOP|BOTTOM}" ptc="int">
  Contains location information for this node. For pins or segments of length one, xlow = xhigh and ylow = yhigh.
  Required Attributes
  • xlow, xhigh, ylow, yhigh – Integer coordinates of the ends of this routing source.
  • ptc – This is the pin, track, or class number that depends on the rr_node type.

  Optional Attributes
  • side – For IPIN and OPIN nodes specifies the side of the grid tile on which the node is located. Valid values are { LEFT | RIGHT | TOP | BOTTOM }. In other cases this attribute should not be specified.
</loc>
```

```xml
<timing R="float" C="float">
  This optional subtag contains information used for timing analysis
  Required Attributes
  • R – The resistance that goes through this node. This is only the metal resistance, it does not include the resistance of the switch that leads to another routing resource node.
</timing>
```
• C – The total capacitance of this node. This includes the metal capacitance, input capacitance of all the switches hanging off the node, the output capacitance of all the switches to the node, and the connection box buffer capacitances that hangs off it.

<segment segment_id="int">  
This optional subtag describes the information of the segment that connects to the node.  

Required Attributes  
• segment_id – This describes the index of the segment type. This value only applies to horizontal and vertical channel types. It can be left empty, or as -1 for other types of nodes.

Edges

The final subtag is the rr_edges tag that encloses information about all the edges between nodes. Each rr_edges tag contains multiple subtags:

<edge src_node="int" sink_node="int" switch_id="int"/>  
This subtag repeats every edge that connects nodes together in the graph.  

Required Attributes  
• src_node, sink_node – The index for the source and sink node that this edge connects to.  
• switch_id – The type of switch that connects the two nodes.

Node and Edge Metadata

metadata blocks (see Architecture metadata) are supported under both node and edge tags.

Routing Resource Graph Format Example

An example of what a generated routing resource graph file would look like is shown below:

Listing 4.6: Example of a routing resource graph in XML format

```
<rr_graph tool_name="vpr" tool_version="82a3c72" tool_comment="Based on my_arch.xml">
  <channels>
    <channel chan_width_max="2" x_min="2" y_min="2" x_max="2" y_max="2"/>
    <x_list index="1" info="5"/>
    <y_list index="2" info="5"/>
  </channels>
  <switches>
    <switch id="0" name="my_switch" buffered="1">
      <timing R="100" Cin="1233-12" Cout="123e-12" Tdel="1e-9"/>
      <sizing mux_trans_size="2.32" buf_size="23.54"/>
    </switch>
  </switches>
  <segments>
    <segment id="0" name="L4">
```

(continues on next page)
<segment>
  <timing R_per_meter="201.7" C_per_meter="1.10e-15"/>
</segment>
</segments>

<block_types>
  <block_type id="0" name="io" width="1" height="1">
    <pin_class type="input">
      <pin ptc="0">DATIN[0]</pin>
      <pin ptc="1">DATIN[1]</pin>
      <pin ptc="2">DATIN[2]</pin>
      <pin ptc="3">DATIN[3]</pin>
    </pin_class>
    <pin_class type="output">
      <pin ptc="4">DATOUT[0]</pin>
      <pin ptc="5">DATOUT[1]</pin>
      <pin ptc="6">DATOUT[2]</pin>
      <pin ptc="7">DATOUT[3]</pin>
    </pin_class>
  </block_type>
  <block_type id="1" name="buf" width="1" height="1">
    <pin_class type="input">
      <pin ptc="0">IN</pin>
    </pin_class>
    <pin_class type="output">
      <pin ptc="1">OUT</pin>
    </pin_class>
  </block_type>
</block_types>

<grid>
  <grid_loc x="0" y="0" block_type_id="0" width_offset="0" height_offset="0"/>
  <grid_loc x="1" y="0" block_type_id="1" width_offset="0" height_offset="0"/>
</grid>

<rr_nodes>
  <node id="0" type="SOURCE" direction="NONE" capacity="1">
    <loc xlow="0" ylow="0" xhigh="0" yhigh="0" ptc="0"/>
    <timing R="0" C="0"/>
  </node>
  <node id="1" type="CHANX" direction="INC" capacity="1">
    <loc xlow="0" ylow="0" xhigh="2" yhigh="0" ptc="0"/>
    <timing R="100" C="12e-12"/>
    <segment segment_id="0"/>
  </node>
</rr_nodes>

<rr_edges>
  <edge src_node="0" sink_node="1" switch_id="0"/>
  <edge src_node="1" sink_node="2" switch_id="0"/>
</rr_edges>
</rr_graph>
4.6.9 Block types usage summary (.txt .xml or .json)

Block types usage summary is a file written in human or machine readable format. It describes types and the amount of cluster-level FPGA resources that are used by implemented design. This file is generated after the placement step with option: –write_block_usage <filename>. It can be saved as a human readable text file or in XML or JSON file to provide machine readable output. Format is selected based on the extension of the <filename>.

The summary consists of 4 parameters:

• nets number - the amount of created nets
• blocks number - sum of blocks used to implement the design
• input pins - sum of input pins
• output pins - sum of output pins

and a list of block types followed by the number of specific block types that are used in the design.

**TXT**

Presents the information in human readable format, the same as in log output:

```plaintext
Netlist num_nets: <int>
Netlist num_blocks: <int>
Netlist <block_type_name_0> blocks: <int>
Netlist <block_type_name_1> blocks: <int>
...
Netlist <block_type_name_n> blocks: <int>
Netlist inputs pins: <int>
Netlist output pins: <int>
```

**JSON**

One of two available machine readable formats. The information is written as follows:

```json
{
  "num_nets": "<int>",
  "num_blocks": "<int>",
  "input_pins": "<int>",
  "output_pins": "<int>",
  "blocks": {
    ",block_type_name_0\": "<int>",
    ",block_type_name_1\": "<int>",
    ...,
    ",block_type_name_n\": "<int>
  }
}
```
XML

Second machine readable format. The information is written as follows:

Listing 4.9: XML format of block types usage summary

```xml
<?xml version="1.0" encoding="UTF-8"?>
<block_usage_report>
  <nets num="<int>"></nets>
  <blocks num="<int>">
    <block type="<block_type_name_0>" usage="<int>"></block>
    <block type="<block_type_name_1>" usage="<int>"></block>
    ...
    <block type="<block_type_name_n>" usage="<int>"></block>
  </blocks>
  <input_pins num="<int>"></input_pins>
  <output_pins num="<int>"></output_pins>
</block_usage_report>
```

4.6.10 Timing summary (.txt .xml or .json)

Timing summary is a file written in human or machine readable format. It describes final timing parameters of design implemented for the FPGA device. This file is generated after the routing step with option: `--write_timing_summary <filename>`. It can be saved as a human readable text file or in XML or JSON file to provide machine readable output. Format is selected based on the extension of the `<filename>`.

The summary consists of 4 parameters:

- **Critical Path Delay (cpd) [ns]**
- **Max Circuit Frequency (Fmax) [MHz]**
- **setup Worst Negative Slack (sWNS) [ns]**
- **setup Total Negative Slack (sTNS) [ns]**

TXT

Presents the information in human readable format, the same as in log output:
**4.7 Debugging Aids**

**Note:** This section is most relevant to developers modifying VPR

The report_timing.setup.rpt file lists details about the critical path of a circuit, and is very useful for determining why your circuit is so fast or so slow.

To access detailed echo files from VPR’s operation, use the command-line option `--echo_file on`. After parsing the netlist and architecture files, VPR dumps out an image of its internal data structures into echo files (typically ending in `.echo`). These files can be examined to be sure that VPR is parsing the input files as you expect.

You can visualize and control the placement move generator whenever the placement engine is paused in the UI. Run with graphics and VTR_ENABLE_DEBUG_LOGGONG enabled and set a breakpoint to stop placement. The new location of the moving block for each proposed move will be highlighted with GREEN and the old location will be highlighted.
with GOLD. The fanin and fanout blocks will also be highlighted. The move type, move outcome and delta cost will be printed in the status bar. .. warning:: VPR must have been compiled with \texttt{VTR\_ENABLE\_DEBUG\_LOGGING} on to get any debug output from this flag.

If the preprocessor flag \texttt{DEBUG} is defined in \texttt{vpr\_types.h}, some additional sanity checks are performed during a run. \texttt{DEBUG} only slows execution by 1 to 2\%. The major sanity checks are always enabled, regardless of the state of \texttt{DEBUG}. Finally, if \texttt{VERBOSE} is set in \texttt{vpr\_types.h}, a great deal of intermediate data will be printed to the screen as VPR runs. If you set verbose, you may want to redirect screen output to a file.

The initial and final placement costs provide useful numbers for regression testing the netlist parsers and the placer, respectively. VPR generates and prints out a routing serial number to allow easy regression testing of the router.

Finally, if you need to route an FPGA whose routing architecture cannot be described in VPR’s architecture description file, don’t despair! The router, graphics, sanity checker, and statistics routines all work only with a graph that defines all the available routing resources in the FPGA and the permissible connections between them. If you change the routines that build this graph (in \texttt{rr\_graph*.c}) so that they create a graph describing your FPGA, you should be able to route your FPGA. If you want to read a text file describing the entire routing resource graph, call the \texttt{dump\_rr\_graph} subroutine.

### 4.8 Placer and Router Debugger

![Placer and Router Debugger](image)
4.8.1 Overview

It can be very useful to stop the program at a significant point and evaluate the circuit at that stage. This debugger allows setting breakpoints during placement and routing using a variety of variables and operations. For example the user can stop the placer after a certain number of perturbations, temperature changes, or when a specific block is moved. It can also stop after a net is routed in the routing process and other such scenarios. There are multiple ways to set and manipulate breakpoints which are all explained in detail below.

4.8.2 Adding a breakpoint

Currently the user is required to have graphics on in order to set breakpoints. By clocking the “Debug” button, the debugger window opens up and from there the user can enter integer values in the entry fields and set breakpoints. A more advanced option is using expressions which allows a wider variety of settings since the user can incorporate multiple variables and use boolean operators. This option is found by clicking the “Advanced” button in the debugger window. Using an expression is more accurate than the entry fields when setting multiple breakpoints.

4.8.3 Enabling/Disabling a breakpoint

Enabling and disabling breakpoints are done using the checkboxes in front of each breakpoint in the breakpoint list. The breakpoint is enabled when the box is checked and disabled otherwise.

4.8.4 Deleting a breakpoint

Deleting a breakpoint is done using the trash can button in front of each breakpoint in the breakpoint list.

4.8.5 Reaching a breakpoint

Upon reaching a breakpoint, the program will stop, notify the user which breakpoint was encountered, and give a summary of the current variable values. This information is presented through a pop-up window and printed to the terminal as well.

4.8.6 Available Variables

You can also find the variables’ list in the Advanced Settings Window, on the left.

**Placer Variables**

- **move_num**: every placer perturbation counts as a move, so the user can stop the program after a certain number of moves.
- Ex. move_num == 33
- Ex. move_num += 4

• **temp_count**: every time the temperature is updated it counts as an increase to temp_count. This breakpoint can be enabled through the entry field or using an expression.
  - Ex. temp_count == 5
  - Ex. temp_count += 5

• **from_block**: in every placer move one or more blocks are relocated. from_block specifies the first block that is relocated in the placer. This breakpoint can be enabled through the entry field on the main debugger window or using an expression.
  - Ex. from_block == 83

• **in_blocks_affected**: this variable allows you to stop after your specified block was moved. Unlike “from_block” which only tracks the first block, in_blocks_affected tracks the last block that was moved.
  - Ex. in_blocks_affected == 83

**Router Variables**

• **router_iter**: Every pass through the whole netlist (with each unrouted or poorly routed net being re-routed) counts as a router iteration. This breakpoint can be enabled through the entry field on the main debugger window or using an expression.
  - Ex. router_iter == 2

• **route_net_id**: stops after the specified net is rerouted. This breakpoint can be enabled through the entry field on the main debugger window or using an expression.
  - route_net_id == 12

### 4.8.7 Available Operators

• **==**
  - Ex. temp_count == 2

• **>**
  - Ex. move_num > 94

• **<**
  - Ex. move_num < 94

• **>=**
  - Ex. router_iter >= 2

• **<=**
  - Ex. router_iter <= 2

• **&&**
  - Ex. from_block == 83 && move_num > 72

• **||**
  - Ex. in_blocks_affected == 11 || temp_count == 9

• **+=**
  - Ex. move_num += 8
Odin II is used for logic synthesis and elaboration, converting a subset of the Verilog Hardware Description Language (HDL) into a BLIF netlist.

5.1 Quickstart

5.1.1 Prerequisites

- ctags
- bison
- flex
- gcc 5.x
- cmake 3.9 (minimum version)
- time
- cairo

5.1.2 Building

To build you may use the Makefile wrapper in the $VTR_ROOT/ODIN_II make build To build with debug symbols you may use the Makefile wrapper in $VTR_ROOT/ODIN_II make debug

NOTE

ODIN uses CMake as it’s build system. CMake provides a portable cross-platform build systems with many useful features. For unix-like systems we provide a wrapper Makefile which supports the traditional make and make clean commands, but calls CMake behind the scenes.

WARNING

After you build Odin, please run from the $VTR_ROOT/ODIN_II make test. This will simulate and verify all of the included microbenchmark circuits to ensure that Odin is working correctly on your system.
5.1.3 Basic Usage

./odin_II [arguments]
*Requires one and only one of -c, -V, or -b

5.1.4 Example Usage

The following are simple command-line arguments and a description of what they do. It is assumed that they are being performed in the Odin_II directory.

```bash
./odin_II -V <path/to/verilog/File>
```

Passes a verilog HDL file to Odin II where it is synthesized. Warnings and errors may appear regarding the HDL code.

```bash
./odin_II -b <path/to/blif/file>
```

Passes a blif file to Odin II where it is synthesized.

```bash
./odin_II -V <path/to/verilog/File> -a <path/to/arch/file> -o myModel.blif
```

Passes a verilog HDL file and an architecture to Odin II where it is synthesized. Odin will use the architecture to do technology mapping. Odin will output the blif in the current directory at ./myModel.blif. Warnings and errors may appear regarding the HDL code.

5.2 User guide

5.2.1 Synthesis Arguments

5.2.2 Simulation Arguments

To activate simulation you must pass one and only one of the following argument:

- `-g <number of random vector>`
- `-t <input vector file>`

Simulation always produces the following files:

- input_vectors
- output_vectors
- test.do (ModelSim)
5.2.3 Examples

Example for `-p`

NOTE
Matching for `-p` is done via strstr so general strings will match all similar pins and nodes. (Eg: FF_NODE will create a single port with all flipflops)

Example of .xml configuration file for `-c`

```xml
<config>
  <verilog_files>
    <!-- Way of specifying multiple files in a project! -->
    <verilog_file>verilog_file.v</verilog_file>
  </verilog_files>
  <output>
    <!-- These are the output flags for the project -->
    <output_type>blif</output_type>
    <output_path_and_name>/output_file.blif</output_path_and_name>
    <target>
      <!-- This is the target device the output is being built for -->
      <arch_file>fpga_architecture_file.xml</arch_file>
    </target>
  </output>
  <optimizations>
    <!-- This is where the optimization flags go -->
  </optimizations>
  <debug_outputs>
    <!-- Various debug options -->
    <debug_output_path>./debug_output_path</debug_output_path>
    <output_ast_graphs>1</output_ast_graphs>
    <output_netlist_graphs>1</output_netlist_graphs>
  </debug_outputs>
</config>
```

NOTE
Hard blocks can be simulated; given a hardblock named block in the architecture file with an instance of it named instance in the verilog file, write a C method with signature defined in SRC/sim_block.h and compile it with an output filename of block+instance.so in the directory you plan to invoke Odin_II from.

When compiling the file, you’ll need to specify the following arguments to the compiler (assuming that you’re in the SANBOX directory):

`cc -I../../libarchfpga_6/include/ -L../../libarchfpga_6 -lvpr_6 -lm --shared -o block+instance.so block.c`

If the netlist generated by Odin II contains the definition of a hardblock which doesn’t have a shared object file defined for it in the working directory, Odin II will not work if you specify it to use the simulator with the `-g` or `-t` options.

WARNING
Use of static memory within the simulation code necessitates compiling a distinct shared object file for each instance of the block you wish to simulate. The method signature the simulator expects contains only 
int and int[] parameters, leaving the code provided to simulate the hard blokc agnostic of the internal Odin II data structures. However, a cycle parameter is included to provide researchers with the ability to delay results of operations performed by the simulation code.

Examples vector file for -t or -T

```
## Example vector input file
GLOBAL_SIM_BASE_CLK input_1 input_2 input_3 clk_input
## Comment
0 0XA 1011 0XD 0
0 0XB 0011 0XF 1
0 0XC 1100 0X2 0

## Example vector output file
output_1 output_2
## Comment
1011 0Xf
0110 0X4
1000 0X5
```

**NOTE**

Each line represents a vector. Each value must be specified in binary or hex. Comments may be included by placing an # at the start of the line. Blank lines are ignored. Values may be separated by non-newline whitespace. (tabs and spaces) Hex values must be prefixed with 0X or 0x.

Each line in the vector file represents one cycle, or one falling edge and one rising edge. Input vectors are read on a falling edge, while output vectors are written on a rising edge. The input vector file does not have a clock input, it is assumed it is controlled by a single global clock that is why it is necessary to add a GLOBAL_SIM_BASE_CLK to the input. To read more about this please visit here.

Examples using vector files -t and -T

A very useful function of Odin II is to compare the simulated output vector file with the expected output vector file based on an input vector file and a verilog file. To do this the command line should be:

```
./odin_II -V <Path/to/verilog/file> -t <Path/to/Input/Vector/File> -T <Path/to/Output/Vector/File>
```

An error will arise if the output vector files do not match.

Without an expected vector output file the command line would be:

```
./odin_II -V <Path/to/verilog/file> -t <Path/to/Input/Vector/File>
```

The generated output file can be found in the current directory under the name output_vectors.
Example using vector files -g

This function generates N amount of random input vectors for Odin II to simulate with.

```
./odin_II -V <Path/to/verilog/file> -g 10
```

This example will produce 10 autogenerated input vectors. These vectors can be found in the current directory under input_vectors and the resulting output vectors can be found under output_vectors.

### 5.2.4 Getting Help

If you have any questions or concerns there are multiple outlets to express them. There is a [google group](#) for users who have questions that is checked regularly by Odin II team members. If you have found a bug please make an issue in the vtr-verilog-to-routing GitHub repository.

### 5.2.5 Reporting Bugs and Feature Requests

#### Creating an Issue on GitHub

Odin II is still in development and there may be bugs present. If Odin II doesn’t perform as expected or doesn’t adhere to the Verilog Standard, it is important to create a [bug report](#) in the GitHub repository. There is a template included, but make sure to include micro-benchmark(s) that reproduces the bug. This micro-benchmark should be as simple as possible. It is important to link some documentation that provides insight on what Odin II is doing that differs from the Verilog Standard. Linked below is a pdf of the IEEE Standard of Verilog (2005) that could help.

[IEEE Standard for Verilog Hardware Description Language](#)

If unsure, there are several outlets to ask questions in the Help section.

#### Feature Requests

If there are any features that the Odin II system overlooks or would be a great addition, please make a [feature request](#) in the GitHub repository. There is a template provided and be as in-depth as possible.

### 5.3 Verilog Support

#### 5.3.1 Lexicon

Verilog Synthesizable Operators Support

Verilog NON-Synthesizable Operator Support

Verilog Synthesizable Keyword Support

Verilog NON-Synthesizable Keyword Support

C Functions support
Verilog Synthesizable preprocessor Keywords Support

5.3.2 Syntax

inline port declaration in the module declaration i.e:

```verilog
module a(input clk)
...
endmodule
```

5.4 Contributing

The Odin II team welcomes outside help from anyone interested. To fix issues or add a new feature submit a PR or WIP PR following the provided guidelines.

5.4.1 Creating a Pull Request (PR)

**Important** Before creating a Pull Request (PR), if it is a bug you have happened upon and intend to fix make sure you create an issue beforehand.

Pull requests are intended to correct bugs and improve Odin’s performance. To create a pull request, clone the vtr-verilog-to-routing repository and branch from the master. Make changes to the branch that improve Odin II and correct the bug. **Important** In addition to correcting the bug, it is required that test cases (benchmarks) are created that reproduce the issue and are included in the regression tests. An example of a good test case could be the benchmark found in the “Issue” being addressed. The results of these new tests need to be regenerate. See regression tests for further instruction. Push these changes to the cloned repository and create the pull request. Add a description of the changes made and reference the “issue” that it corrects. There is a template provided on GitHub.

Creating a “Work in progress” (WIP) PR

**Important** Before creating a WIP PR, if it is a bug you have happened upon and intend to fix make sure you create an issue beforehand.

A “work in progress” PR is a pull request that isn’t complete or ready to be merged. It is intended to demonstrate that an Issue is being addressed and indicates to other developers that they don’t need to fix it. Creating a WIP PR is similar to a regular PR with a few adjustments. First, clone the vtr-verilog-to-routing repository and branch from the master. Make changes to that branch. Then, create a pull request with that branch and include WIP in the title. This will automatically indicate that this PR is not ready to be merged. Continue to work on the branch, pushing the commits regularly. Like a PR, test cases must be included through the use of benchmarks. See regression tests for further instruction.
5.4.2 Odin II’s Flow

Odin II functions by systematically executing a set of steps determined by the files and arguments passed in. The figure below illustrates the flow of Odin II if a Verilog File is passed, with an optional FPGA Architecture Specification File. The simulator is only activated if an Input Vector file is passed in which creates the Output Vector File.

.. graphviz::

```plaintext
digraph G { 
0[label="Verilog HDL File",shape=plaintext];
1[label="Input Vector File",shape=plaintext];
2[label="Output Vector File",shape=diamond];
3[label="FPGA Architecture Specification File",shape=plaintext];
4[label="Build Abstract Syntax Tree",shape=box];
5[label="Elaborate AST",shape=box];
6[label="Build Netlist",shape=box];
7[label="Partial Mapping",shape=box];
8[label="Simulator",shape=box];
9[label="Output Blif",shape=diamond];
10[label="Build Abstract Syntax Tree",shape=box];
11[label="Output Blif",shape=diamond];

0 -> 5 -> 6 -> 7 -> 8
7->10 [color=purple]
4->8 [style=dotted] [color=purple]
8->11
4->10 [style=dotted] [color=purple]
2->10 [color=purple]
10->3 [color=purple]
```

Currently, BLIF files being passed in are only used for simulation; no partial mapping takes place. The flow is depicted in the figure below.

.. graphviz::

```plaintext
digraph G { 
0[label="Input Blif File",shape=plaintext];
1[label="Read Blif",shape=box];
2[label="Build Netlist",shape=box];
3[label="Output Blif",shape=diamond];
4[label="Output Blif",shape=plaintext];
5[label="Simulator",shape=box];
6[label="FPGA Architecture Specification File",shape=box];
7[label="Input Vector File",shape=plaintext];
8[label="Output Vector File",shape=diamond];

0->1->3
3->5 [color=purple]
3->4
5->8 [color=purple]
7->5 [color=purple]
6->5 [style=dotted] [color=purple]
```

}
Building the Abstract Syntax Tree (AST)

Odin II uses Bison and Flex to parse a passed Verilog file and produce an Abstract Syntax Tree for each module found in the Verilog File. The AST is considered the “front-end” of Odin II. Most of the code for this can be found in verilog_bison.y, verilog_flex.l and parse_making_ast.cpp located in the ODIN_II/SRC directory.

AST Elaboration

In this step, Odin II parses through the ASTs and elaborates specific parts like for loops, function instances, etc. It also will simplify the tree and rid itself of useless parts, such as an unused if statement. It then builds one large AST, incorporating each module. The code for this can mostly be found in ast_elaborate.cpp located in the ODIN_II/SRC directory.

NOTE

These ASTs can be viewed via graphviz using the command -A. The file(s) will appear in the main directory.

Building the Netlist

Once again, Odin II parses through the AST assembling a Netlist. During the Netlist creation, pins are assigned and connected. The code for this can be found in netlist_create_from_ast.cpp located in the ODIN_II/SRC directory.

NOTE

The Netlist can be viewed via graphviz using the command -G. The file will appear in the main directory under net.dot.

Partial Mapping

During partial mapping, Odin II maps the logic using an architecture. If no architecture is passed in, Odin II will create the soft logic and use LUTs for mapping. However, if an architecture is passed, Odin II will map accordingly to the available hard blocks and LUTs. It uses a combination of soft logic and hard logic.

Simulator

The simulator of Odin II takes an Input Vector file and creates an Output Vector file determined by the behaviour described in the Verilog file or BLIF file.

5.4.3 Useful tools of Odin II for Developers

When making improvements to Odin II, there are some features the developer should be aware of to make their job easier. For instance, Odin II has a -A and -G command that prints the ASTs and Netlist viewable with GraphViz. These files can be found in the ODIN_II directory. This is very helpful to visualize what is being created and how everything is related to each other in the Netlist and AST.

Another feature to be aware of is make test. This build runs through all the regression tests and will list all the benchmarks that fail. It is important to run this after every major change implemented to ensure the change only affects benchmarks it was intended to effect (if any). It sheds insight on what needs to be fixed and how close it is to being merged with the master.
5.5 Regression Tests

Regression tests are tests that are repeatedly executed to assess functionality. Each regression test targets a specific function of Odin II. There are two main components of a regression test; benchmarks and a configuration file. The benchmarks are comprised of verilog files, input vector files and output vector files. The configuration file calls upon each benchmark and synthesizes them with different architectures. The current regression tests of Odin II can be found in regression_test/benchmark.

5.5.1 Benchmarks

Benchmarks are used to test the functionality of Odin II and ensure that it runs properly. Benchmarks of Odin II can be found in regression_test/benchmark/verilog/any_folder. Each benchmark is comprised of a verilog file, an input vector file, and an output vector file. They are called upon during regression tests and synthesized with different architectures to be compared against the expected results. These tests are useful for developers to test the functionality of Odin II after implementing changes. The command `make test` runs through all these tests, comparing the results to previously generated results, and should be run through when first installing.

Unit Benchmarks

Unit benchmarks are the simplest of benchmarks. They are meant to isolate different functions of Odin II. The goal is that if it does not function properly, the error can be traced back to the function being tested. This cannot always be achieved as different functions depend on others to work properly. It is ideal that these benchmarks test bit size capacity, erroneous cases, as well as standards set by the IEEE Standard for Verilog® Hardware Description Language - 2005.

Micro Benchmarks

Micro benchmarks are precise, like unit benchmarks, however are more syntactic. They are meant to isolate the behaviour of different functions. They trace the behaviour of functions to ensure they adhere to the IEEE Standard for Verilog® Hardware Description Language - 2005. Like unit benchmarks, they should check erroneous cases and behavioural standards set by the IEEE Standard for Verilog® Hardware Description Language - 2005.

Macro Benchmarks

Macro benchmarks are more realistic tests that incorporate multiple functions of Odin II. They are intended to simulate real-user behaviour to ensure that functions work together properly. These tests are designed to test things like syntax and more complicated standards set by the IEEE Standard for Verilog® Hardware Description Language - 2005.

External Benchmarks

External benchmarks are benchmarks created by outside users to the project. It is possible to pull an outside directory and build them on the fly thus creating a benchmark for Odin II.
5.5.2 Creating Regression Tests

New Regression Test Checklist

- Create benchmarks here
- Create configuration file here
- Create a folder in the task directory for the configuration file here
- Generate the results here
- Add the task to a suite (large suite if generating the results takes longer than 3 minutes, otherwise put in light suite) here
- Update the documentation by providing a summary in Regression Test Summary section and updating the Directory Tree here

New Benchmarks added to Regression Test Checklist

- Create benchmarks and add them to the correct regression test folder found in the benchmark/verilog directory here (There is a description of each regression test here)
- Regenerate the results here

Include

- verilog file
- input vector file
- expected output vector file
- configuration file (conditional)
- architecture file (optional)

Creating Benchmarks

If only a few benchmarks are needed for a PR, simply add the benchmarks to the appropriate set of regression tests. The Regression Test Summary summarizes the target of each regression test which may be helpful.

The standard of naming the benchmarks are as follows:

- verilog file: meaningful_title.v
- input vector file: meaningful_title_input
- output vector file: meaningful_title_output

If the tests needed do not fit in an already existing set of regression tests or need certain architecture(s), create a separate folder in the verilog directory and label appropriately. Store the benchmarks in that folder. Add the architecture (if it isn’t one that already exists) to ../vtr_flow/arch.

NOTE

If a benchmark fails and should pass, include a $display statement in the verilog file in the following format:

$display("Expect::FUNCTION < message >");
The function should be in all caps and state what is causing the issue. For instance, if else if was behaving incorrectly it should read ELSE_IF. The message should illustrate what should happen and perhaps a suggestion in where things are going wrong.

Creating a Configuration File

A configuration file is only necessary if the benchmarks added are placed in a new folder. The configuration file is where architectures and commands are specified for the synthesis of the benchmarks. The configuration file must be named task.conf. The following is an example of a standard task.conf (configuration) file:

```
########################
# <title> benchmarks config
########################

# commands
regression_params=--include_default_arch
script_synthesis_params=--time_limit 3600s
script_simulation_params=--time_limit 3600s
simulation_params= -L reset rst -H we

# setup the architecture (standard architectures already available)
archs_dir=../vtr_flow/arch/timing

arch_list_add=k6_N10_40nm.xml
arch_list_add=k6_N10_mem32K_40nm.xml
arch_list_add=k6_frac_N10_frac_chain_mem32K_40nm.xml

# setup the circuits
circuits_dir=regression_test/benchmark/verilog/

circuit_list_add=<verilog file group>/*.vh
circuit_list_add=<verilog file group>/*.v

synthesis_parse_file=regression_test/parse_result/conf/synth.toml
simulation_parse_file=regression_test/parse_result/conf/sim.toml
```

The following key = value are available for configuration files:

Regression Parameters:

- **--verbose** display error logs after batch of tests
- **--concat_circuit_list** concatenate the circuit list and pass it straight through to odin
- **--generate_bench** generate input and output vectors from scratch
- **--disable_simulation** disable the simulation for this task
- **--disable_parallel_jobs** disable running circuit/task pairs in parallel
- **--randomize** perform a dry run randomly to check the validity of the task and flow
- **--regenerate_expectation** regenerate expectation and override the expected value only if there’s a mismatch
- **--generate_expectation** generate the expectation and override the expectation file

5.5. Regression Tests
Creating a Task

The following diagram illustrates the structure of regression tests. Each regression test needs a corresponding folder in the task directory containing the configuration file. The <task display name> should have the same name as the verilog file group in the verilog directory. This folder is where the synthesis results and simulation results will be stored. The task display name and the verilog file group should share the same title.

```
    ____________
   /               \
  /                 \\      ODIN_II
 /                   \\
|                     |
| regression_test     |
|                     |
|                     |
|                     |
|                     |
|                     |
|____________________|
|                    |
|                     |
|                     |
|                     |
|                     |
|                     |
|                     |
|                     |
|____________________|
|                    |
|                     |
|                     |
|                     |
|                     |
|                     |
|                     |
|____________________|
|                    |
|                     |
|                     |
|                     |
|                     |
|                     |
|____________________|
|                    |
|                     |
|                     |
|                     |
|                     |
|____________________|
|                    |
|                     |
|                     |
|                     |
|                     |
|____________________|
|                    |
|                     |
|                     |
|                     |
|____________________|
|                    |
|                     |
|                     |
|                     |
|____________________|
|                    |
```

Creating a Complicated Task

There are times where multiple configuration files are needed in a regression test due to different commands wanted or architectures. The task cmd_line_args is an example of this. If that is the case, each configuration file will still need its own folder, however these folders should be placed in a parent folder.

```
    ____________
   /               \
  /                 \\      ODIN_II
 /                   \\
|                     |
| regression_test     |
|                     |
|                     |
|                     |
|                     |
|                     |
|____________________|
|                    |
|                     |
|                     |
|                     |
|                     |
|____________________|
|                    |
|                     |
|                     |
|                     |
|                     |
|____________________|
|                    |
|                     |
|                     |
|                     |
|____________________|
|                    |
|                     |
|                     |
|                     |
|____________________|
|                    |
|                     |
|                     |
|                     |
|____________________|
|                    |
```
Creating a Suite

Suites are used to call multiple tasks at once. This is handy for regenerating results for multiple tasks. In the diagram below you can see the structure of the suite. The suite contains a configuration file that calls upon the different tasks named `task_list.conf`.

```
  ─── ODIN_II
   │   ─_regression_test
   │       ─ benchmark
   │           ─ suite
   │               └── < suite display name >
   │                   └── task_list.conf
   │
   │   ─ task
   │       └── < parent task display name >
   │           └── < task display name >
   │               └── [ simulation_result.json ]
   │                   └── [ synthesis_result.json ]
   │                           └── task.conf
   │
   │   └── < task display name >
   │       └── [ simulation_result.json ]
   │           └── [ synthesis_result.json ]
   │                   └── task.conf
   │
   │
   │   └── < task display name >
   │       └── [ simulation_result.json ]
   │           └── [ synthesis_result.json ]
   │                   └── task.conf
   │
   │
   │   ─ verilog
   │       └── < verilog file group >
   │           └── *.v
   │                   └── *.vh
```

In the configuration file all that is required is to list the tasks to be included in the suite with the path. For example, if the wanted suite was to call the binary task and the operators task, the configuration file would be as follows:

```
regression_test/benchmark/task/operators
regression_test/benchmark/task/binary
```

For more examples of `task_list.conf` configuration files look at the already existing configuration files in the suites.

**Regenerating Results**

**WARNING**

**BEFORE** regenerating the result, run `make test` to ensure any changes in the code don’t affect the results of benchmarks beside your own. If they do, the failing benchmarks will be listed.

Regenerating results is necessary if any regression test is changed (added benchmarks), if a regression test is added, or if a bug fix was implemented that changes the results of a regression test. For all cases, it is necessary to regenerate the results of the task corresponding to said change. The following commands illustrate how to do so:

```
make sanitize
```

then: where N is the number of processors in the computer, and the path following `-t` ends with the same name as the folder you placed
NOTE

DO NOT run the make sanitize if regenerating the large test. It is probable that the computer will not have enough ram to do so and it will take a long time. Instead run make build.

For more on regenerating results, refer to the Verify Script section.

5.5.3 Regression Test Summaries

c_functions

This regression test targets c functions supported by Verilog such as clog_2.

cmd_line_args

This is a more complicated regression test that incorporates multiple child tasks. It targets different commands available in Odin II. Although it doesn’t have a dedicated set of benchmarks in the verilog folder, the configuration files call on different preexisting benchmarks.

FIR

FIR is an acronym for “Finite Impulse Response”. These benchmarks were sourced from Layout Aware Optimization of High Speed Fixed Coefficient FIR Filters for FPGAs. They test a method of implementing high speed FIR filters on FPGAs discussed in the paper.

full

The full regression test is designed to test real user behaviour.
It does this by simulating flip flop, muxes and other common uses of Verilog.

large

This regression test targets cases that require a lot of ram and time.

micro

The micro regression test targets hards blocks and pieces that can be easily instantiated in architectures.
mixing_optimization

The mixing optimization regression test targets mixing implementations for operations implementable in hard blocks and their soft logic counterparts that can be easily instantiated in architectures. The tests support extensive command line coverage, as well as provide infrastructure to enable the optimization from an .xml configuration file, require for using the optimization as a part of VTR synthesis flow.

operators

This regression test targets the functionality of different operators. It checks bit size capacity and behaviour.

syntax

The syntax regression test targets syntactic behaviour. It checks that functions work cohesively together and adhere to the verilog standard.

keywords

This regression test targets the function of keywords. It has a folder or child for each keyword containing their respective benchmarks. Some folders have benchmarks for two keywords like task_endtask because they both are required together to function properly.

preprocessor

This set of regression test includes benchmarks targetting compiler directives available in Verilog.

Regression Tests Directory Tree

```
 benchmark
   ├── suite
   │    ├── complex_synthesis_suite
   │    │    └── task_list.conf
   │    ├── full_suite
   │    │    └── task_list.conf
   │    ├── heavy_suite
   │    │    └── task_list.conf
   │    └── light_suite
   │         └── task_list.conf
   │            └── yosys+odin
   └── task
       └── arch_sweep
           └── synthesis_result.json
           └── task.conf
       └── c_functions
           └── clog2
               └── simulation_result.json
               └── synthesis_result.json
                   └── task.conf
               └── cmd_line_args
```
5.5. Regression Tests
5.6 Verify Script

The verify_odin.sh script is designed for generating regression test results.

./verify_odin.sh [args]
5.6.1 Arguments

*The tool requires a task to run hence -t <task directory> must be passed in

5.6.2 Examples

The following examples are being performed in the ODIN_II directory:

Generating Results for a New Task

To generate new results, synthesis_parse_file and simulation_parse_file must be specified in task.conf file. The following commands will generate the results of a new regression test using N processors:

```
make sanitize

./verify_odin.sh --generate_expectation -j N -t <regression_test/benchmark/task/<task_name>
```

A synthesis_result.json and a simulation_result.json will be generated in the task’s folder. The simulation results for each benchmark are only generated if they synthesize correctly (no exit error), thus if none of the benchmarks synthesize there will be no simulation_result.json generated.

Regenerating Results for a Changed Test

The following commands will only generate the results of the changes. If there are new benchmarks it will add to the results. If there are deleted benchmarks or modified benchmarks the results will be updated accordingly.

```
make sanitize

./verify_odin.sh --regenerate_expectation -t <regression_test/benchmark/task/<task_name>
```

Generating Results for a Suite

The following commands generate the results for all the tasks called upon in a suite.

```
make sanitize

./verify_odin.sh --generate_expectation -t <regression_test/benchmark/suite/<suite_name>
```

NOTE

If the suite calls upon the large test DO NOT run make sanitize. Instead run make build.

```
./verify_odin.sh --generate_expectation -t <regression_test/benchmark/suite/<suite_name>
```
Checking the configuration file

The following commands will check if a configuration file is being read properly.

```
make build
./verify_odin.sh --dry_run -t <regression_test/benchmark/<path/to/config_file/directory>
```

Running a subset of tests in a suite

The following commands will run only the tests matching `<test regex>`:

```
./verify_odin.sh -t <regression_test/benchmark/suite/<suite_name> <test regex>
```

You may specify as many test regular expressions as desired and the script will run any test that matches at least one regex.

**NOTE**

This uses grep’s extended regular expression syntax for matching test names.
Test names matched are of the form `<suite_name>/<test_name>/`

## 5.7 TESTING ODIN II

The `verify_odin.sh` script will simulate the microbenchmarks and a larger set of benchmark circuits. These scripts use simulation results which have been verified against ModelSim.

After you build Odin-II, run `make test ELABORATOR=odin` or `make test` to ensure that everything is working correctly on your system. The `verify_odin.sh` also simulates the blif output, as well as simulating the verilog with and without the architecture file.

Before checking in any changes to Odin II, please run both of these scripts to ensure that both of these scripts execute correctly. If there is a failure, use ModelSim to verify that the failure is within Odin II and not a faulty regression test. If it is a faulty regression test, make an issue on GitHub. The Odin II simulator will produce a `test.do` file containing clock and input vector information for ModelSim.

When additional circuits are found to agree with ModelSim, they should be added to the regression tests. When new features are added to Odin II, new microbenchmarks should be developed which test those features for regression. This process is illustrated in the Developer Guide, in the Regression Tests section.

### 5.7.1 USING MODELSIM TO TEST ODIN II

ModelSim may be installed as part of the Quartus II Web Edition IDE. Load the Verilog circuit into a new project in ModelSim. Compile the circuit, and load the resulting library for simulation.

You may use random vectors via the `-g` option, or specify your own input vectors using the `-t` option. When simulation is complete, load the resulting `test.do` file into your ModelSim project and execute it. You may now directly compare the vectors in the `output_vectors` file with those produced by ModelSim.

**NOTE**

For simulation purposes, you may need to handle the `GLOBAL_SIM_BASE_CLK` signal in the `input_vector` by either adding this signal as an input signal to the top module or removing it from the `input_vector` file.
To add the verified vectors and circuit to an existing test set, move the Verilog file (eg: test_circuit.v) to the test set folder. Next, move the input_vectors file to the test set folder, and rename it test_circuit_$ELABORATOR_input ($ELABORATOR: odin, yosys). Finally, move the output_vectors file to the test set folder and rename it test_circuit_$ELABORATOR_output.
Yosys+Odin-II is the combination of Yosys as the elaborator and the Odin-II partial mapper and logic optimizer to perform logic synthesis for a subset of the Verilog Hardware Description Language (HDL) into a BLIF netlist. Odin-II lacks support for the Verilog-2005 standard and SystemVerilog, but it provides complex partial mapping for balancing soft logic and hard blocks such as embedded multipliers, adders, and memories in FPGA architectures. Yosys, an open framework for RTL synthesis, provides extensive support for HDLs. However, the Yosys flow forces the user to decide the discrete circuit implementation manually.

The approach taken by the Yosys synthesizer is to map all discrete components into available hard blocks or to explode them in low-level logic when not available. The Yosys+Odin-II improves device utilization and simplifying the flow by automating hard logic decisions with architecture awareness. Yosys acts as the front-end HDL elaborator, while Odin-II is fed by Yosys generated BLIF files to perform the partial mapping. As a result, hard/soft logic trade-off decisions and heterogeneous logic inference become available for Yosys coarse-grained BLIF files.
6.1 Quickstart

6.1.1 Prerequisites

- ctags
- bison
- flex
- gcc 5.x
- cmake 3.9 (minimum version)
- time
- cairo
- gawk
- xdot
- tcl-dev
- graphviz
- pkg-config
- python3
- libffi-dev
- libreadline-dev
- libboost-system-dev
- libboost-python-dev
- libboost-filesystem-dev
- zlib1g-dev

6.1.2 Building

To build, you may use the Makefile wrapper in the $VTR_ROOT/ODIN_II directory, via the make build ELABORATOR=yosys command. As Yosys is added as an external library to the VTR project, the debug option is only available for Odin-II. To build with debug mode, you may need to pass the debug flag to the CMake parameters using the previous Makefile wrapper, i.e., CMAKE_PARAMS="-DODIN_DEBUG=ON". To ease this process, you can build Yosys+Odin-II with Odin-II in debug mode using the make debug ELABORATOR=yosys command.

The second approach to build the VTR flow with the Yosys+Odin-II front-end is to use the main VTR Makefile. i.e., calling make in the $VTR_ROOT directory. In this approach, the compile flag -DODIN_USE_YOSYS=ON should be passed to the CMake parameters as follows: make CMAKE_PARAMS="-DODIN_USE_YOSYS=ON".

**Note:** Yosys uses Makefile as its build system. Since CMake provides portable, cross-platform build systems with many useful features, we provide a CMake wrapper to successfully embeds the Yosys library into the VTR flow. The Makefile wrapper in the $VTR_ROOT/ODIN_II provides the build support for Unix-like systems but, in fact, calls CMake behind the scenes.
Warning: Once you build Yosys+Odin-II, you would run `make test ELABORATOR=yosys` from the `$VTR_ROOT/ODIN_II` to simulate and verify the light and heavy benchmark suites to ensure that Yosys+Odin-II is working correctly on your system.

### 6.1.3 Basic Usage

```
./odin_II --elaborator yosys [arguments]
```

*Requires one and only one of `-c`, `-V` or `-b`*

<table>
<thead>
<tr>
<th>Arg</th>
<th>Following Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-c</code></td>
<td>XML Configuration File</td>
<td>an XML configuration file dictating the runtime parameters of odin</td>
</tr>
<tr>
<td><code>-V</code></td>
<td>Verilog HDL File</td>
<td>You may specify multiple verilog HDL files</td>
</tr>
<tr>
<td><code>-b</code></td>
<td>BLIF File</td>
<td>You may specify multiple blif files</td>
</tr>
<tr>
<td><code>-o</code></td>
<td>BLIF output file</td>
<td>full output path and file name for the blif output file</td>
</tr>
<tr>
<td><code>-a</code></td>
<td>architecture file</td>
<td>You may specify multiple verilog HDL files for synthesis</td>
</tr>
<tr>
<td><code>-h</code></td>
<td></td>
<td>Print help</td>
</tr>
</tbody>
</table>

### 6.1.4 Example Usage

The following are simple command-line arguments and a description of what they do. It is assumed that they are being performed in the Odin-II directory.

```
./odin_II --elaborator yosys -V <path/to/Verilog/File>
```

Passes a Verilog HDL file to Yosys for elaboration, then Odin-II performs the partial mapping and optimization. Warnings and errors may appear regarding the HDL code by Yosys.

**Note:** The entire log file of the Yosys elaboration for each run is outputted into a file called `elaboration.yosys.log` located in the same directory of the final output BLIF file.

```
./odin_II --elaborator yosys -V <path/to/Verilog/File> -a <path/to/arch/file> -o output.blif
```

Passes a Verilog HDL file and architecture to Yosys+Odin-II, where it is synthesized. Yosys will use the HDL files to perform elaboration. Then, Odin-II will use the architecture to do partial technology mapping, and will output the BLIF in the current directory at `./output.blif`. If the output BLIF file is not specified, `default_out.blif` is considered the output file name, again located in the current directory.

**Note:** Once the elaboration is fully executed, Yosys generates a coarse-grained BLIF file that the Odin-II BLIF reader will read to create a netlist. This file is named `coarsen_netlist.yosys.blif` located in the current directory.

```
./odin_II -S <path/to/Tcl/File> -a <path/to/arch/file> -o myModel.blif
```

Passes a Tcl script file, including commands for the elaboration by Yosys, along with the architecture file.

### 6.1. Quickstart
6.2 User guide

6.2.1 Synthesis Arguments

<table>
<thead>
<tr>
<th>Arg</th>
<th>Following Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-c</td>
<td>XML Configuration File</td>
<td>XML runtime directives for the synthesizer such as the Verilog file, and parametrized synthesis</td>
</tr>
<tr>
<td>-V</td>
<td>Verilog HDL File</td>
<td>You may specify multiple Verilog HDL files for synthesis</td>
</tr>
<tr>
<td>-b</td>
<td>BLIF File</td>
<td>You may utilize additional commands for the Yosys elaborator</td>
</tr>
<tr>
<td>-S/-tcl</td>
<td>Tcl Script File</td>
<td>You may specify multiple Verilog HDL files for synthesis</td>
</tr>
<tr>
<td>-o</td>
<td>BLIF Output File</td>
<td>You may specify multiple Verilog HDL files for synthesis</td>
</tr>
<tr>
<td>-a</td>
<td>Architecture File</td>
<td>Output netlist graph in GraphViz viewable .dot format. (net.dot, opens with dotty)</td>
</tr>
<tr>
<td>-G</td>
<td></td>
<td>Output AST graph in in GraphViz viewable .dot format.</td>
</tr>
<tr>
<td>-A</td>
<td></td>
<td>Print all warnings. (Can be substantial.)</td>
</tr>
<tr>
<td>-W</td>
<td></td>
<td>Print help</td>
</tr>
<tr>
<td>-coarsen`</td>
<td>Coarse-grained BLIF File</td>
<td>Performing the partial mapping for a given coarse-grained netlist in BLIF</td>
</tr>
<tr>
<td>-elaborator`</td>
<td>[odin (default), yosys]</td>
<td>Specify the tool that should perform the HDL elaboration</td>
</tr>
<tr>
<td>-fflegalize`</td>
<td></td>
<td>Converts latches’ sensitivity to the rising edge as required by VPR</td>
</tr>
<tr>
<td>-show_yosys_log`</td>
<td></td>
<td>Showing the Yosys elaboration logs in the console window</td>
</tr>
</tbody>
</table>

6.2.2 Additional Examples using Odin-II with the Yosys elaborator

```
./odin_II --elaborator yosys -V <Path/to/Verilog/file> --fflegalize
```

Passes a Verilog file to Yosys for performing elaboration. The BLIF elaboration and partial mapping phases will be executed on the generated netlist. However, all latches in the Yosys+Odin-II output file will be rising edge.

```
./odin_II -b <Path/to/BLIF/file> --coarsen --fflegalize
```

Performs the BLIF elaboration and partial mapping phases on the given coarse-grained BLIF file.

Note: The coarse-grained BLIF file must follow the same style and syntax as what it would be when the Yosys synthesizer generates it.
Implicitly performs the synthesis by Yosys+Odin-II. The Tcl script should include Yosys commands similar to the following Tcl script to perform the elaboration. Then, the BLIF elaboration and partial mapping phases will be performed on the coarse-grained BLIF netlist.

Example of Tcl script for Yosys+Odin-II

```tcl
# FILE: $VTR_ROOT/ODIN_II/regression_test/tools/synth.tcl #
yosys -import

# the environment variable VTR_ROOT is set by Odin-II.

# Read the hardware description Verilog
read_verilog -nomem2reg -nolatches PATH_TO_VERILOG_FILE.v;
# Check that cells match libraries and find top module
hierarchy -check -auto-top;

# Make name convention more readable
autoname;
# Translate processes to netlist components such as MUXs, FFs and latches
procs; opt;
# Extraction and optimization of finite state machines
fsm; opt;
# Collects memories, their port and create multiport memory cells
memory_collect; memorydff; opt;

# Looking for combinatorial loops, wires with multiple drivers and used wires
check;
# resolve asynchronous dfs

techmap -map $VTR_ROOT/ODIN_II/techlib/adff2dff.v;
techmap -map $VTR_ROOT/ODIN_II/techlib/adffe2dff.v;
# To resolve Yosys internal indexed part-select circuitry

techmap */t:$shift */t:$shiftx;

## Utilizing the "memory_bram" command and the Verilog design provided at "$VTR_ROOT/ODIN_II/techlib/mem_map.v"
## we could map Yosys memory blocks to BRAMs and ROMs before the Odin-II partial mapping phase.
## However, Yosys complains about expression widths more than 24 bits.
## E.g. reg [63:0] memory [18:0] ==> ERROR: Expression width 33554432 exceeds implementation limit of 16777216!
## Although we provided the required design files for this process (located in ODIN_II/techlib), we will handle
## memory blocks in the Odin-II BLIF elaborator and partial mapper.
# memory_bram -rules $VTR_ROOT/ODIN_II/techlib/mem_rules.txt
# techmap -map $VTR_ROOT/ODIN_II/techlib/mem_map.v;

# Transform the design into a new one with single top module
flatten;
# Transforms pmux into trees of regular multiplexers
pmuxtree;
# To possibly reduce words size
```

(continues on next page)
wreduce;
    # "undirven" to ensure there is no wire without drive
    # "opt_muxtree" removes dead branches, "opt_expr" performs constant folding,
    # removes "undef" inputs from mux cells, and replaces muxes with buffers and inverters.
    # "-noff" a potential option to remove all sdff and etc. Only dff will remain
    opt -undriven -full; opt_muxtree; opt_expr -mux_undef -mux_bool -fine;;;
    # Make name convention more readable
    autoname;
    # Print statistics
    stat;
    # Output BLIF
    write_blif -param -impltf TCL_BLIF;

Note: The output BLIF command, i.e., write_blif, is not required except for the user usage. Indeed, Odin-II automatically handles the Yosys outputting process.

### 6.2.3 Simulation Arguments

Note: Yosys+Odin-II makes use of the Odin-II simulator. For more information please see the Odin-II Simulation Arguments.

**Example of .xml configuration file for -c**

```xml
<config>
    <inputs>
        <!-- These are the output flags for the project -->
        <!-- possible types: verilog, verilog_header and blif -->
        <input_type>Verilog</input_type>
        <!-- Way of specifying multiple files in a project -->
        <input_path_and_name>PATH_TO_CIRCUIT.v</input_path_and_name>
    </inputs>
    <output>
        <!-- These are the output flags for the project -->
        <output_type>blif</output_type>
        <output_path_and_name>PATH_TO_OUTPUT_FILE</output_path_and_name>
    </output>
    <optimizations>
        <!-- This is where the optimization flags go -->
        <multiply size="MMM" fixed="1" fracture="0" padding="-1"/>
        <memory split_memory_width="1" split_memory_depth="PPP"/>
        <adder size="0" threshold_size="AAA"/>
    </optimizations>
</config>
```

(continues on next page)
Note: Hard blocks can be simulated; given a hardblock named block in the architecture file with an instance of it named instance in the file. First, a Verilog module including the hard block signature, similar to single_port_ram and dual_port_ram, should be added to the $VTR_ROOT/vtr_flow/primitives.v file. Note, (* keep_hierarchy *) must be defined exactly a line before the hard block module. Then, write a C method with signature defined in SRC/sim_block.h and compile it with an output filename of block+instance.so in the directory you plan to invoke Yosys+Odin_II from.

Note: Additional information regarding how to compile the aforementioned file, what are the restriction for a C method signature, etc. are mentioned in the Odin-II simulation examples.

Examples using input/output vector files

```
./odin_II --elaborator yosys -V <Path/to/verilog/file> -t <Path/to/Input/Vector/File> -T <Path/to/Output/Vector/File>
```

A mismatch error will arise if the output vector files do not match with the benchmark output vector, located in the verilog directory.

6.2.4 Getting Help

Note: For more information please see Odin-II Getting Help.

6.2.5 Reporting Bugs and Feature Requests

Creating an Issue on GitHub

Note: For more information please see Issue on GitHub.

Feature Requests

If there are any features that the Yosys+Odin-II system overlooks or would be a great addition, please make a feature request in the GitHub repository. There is a template provided and be as in-depth as possible.
6.3 Verilog Support

Yosys RTL framework has extensive Verilog IEEE-2005 support. Please see the Yosys GitHub repository for more information on a few unsupported Verilog-2005 features.

6.3.1 Unsupported Verilog-2005 features by Yosys

<table>
<thead>
<tr>
<th>Yosys</th>
<th>Unsupported Features</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Non-synthesizable language features as defined in IEC 62142(E):2005 / IEEE Std. 1364.1(E):2002</td>
</tr>
<tr>
<td></td>
<td>Net Types</td>
</tr>
<tr>
<td></td>
<td>tri</td>
</tr>
<tr>
<td></td>
<td>triand</td>
</tr>
<tr>
<td></td>
<td>trior</td>
</tr>
<tr>
<td></td>
<td>Keywords</td>
</tr>
<tr>
<td></td>
<td>config</td>
</tr>
<tr>
<td></td>
<td>disable</td>
</tr>
<tr>
<td></td>
<td>Library files</td>
</tr>
</tbody>
</table>

6.4 Developers Guide

6.4.1 Contributing

The Odin-II team welcomes outside help from anyone interested. To fix issues or add a new feature submit a PR or WIP PR following the provided guidelines. The instruction for Yosys+Odin-II contribution is similar to Odin-II. For more information about how to contribute and make a WIP pull request please see the Odin-II contribution documentation.

6.4.2 Yosys+Odin-II’s Flow

Yosys+Odin-II functions via two CAD tools, Yosys and Odin-II, by executing a set of steps determined by the files and arguments passed in. The figure below illustrates the synthesis flow of Odin-II, Yosys, and Yosys+Odin-II if a Verilog file is passed, with an optional FPGA Architecture Specification File. The Yosys+Odin-II synthesis process includes: reading the HDL file and performing a coarse-grained synthesis by Yosys, elaborating the generated coarse-grained netlist by Odin-II, and ultimately performing partial technology mapping and unused logic removal with the FPGA architecture awareness again by Odin-II. The simulator is only activated if an input vector file is passed in which creates the output vector file.

Fine-grained BLIF files, usually generated by Odin-II, being passed in are only used for simulation; no partial mapping takes place. However, the partial mapping can be performed on coarse-grained BLIF netlists, specified by the --coarsen command argument. The flow is depicted in the figure below.
Figure 1 - The Odin-II and Yosys+Odin-II Synthesis Flow

```tcl
# FILE: $VTR_ROOT/ODIN_II/regression_test/tools/synth.tcl #
yosys -import

# the environment variable VTR_ROOT is set by Odin-II.
# Feel free to specify file paths using "$env(VTR_ROOT)/ ..."

# Read the hardware decription Verilog
read_verilog -nomem2reg -nolatches PATH_TO_VERILOG_FILE.v;
# Check that cells match libraries and find top module
hierarchy -check -auto-top;

# Make name convention more readable
autoname;
# Translate processes to netlist components such as MUXs, FFs and latches
procs; opt;
# Extraction and optimization of finite state machines
fsm; opt;
# Collects memories, their port and create multiport memory cells
memory_collect; memory_dff; opt;

# Looking for combinatorial loops, wires with multiple drivers and used wires
without any driver.
check;
# resolve asynchronous dffs
  techmap -map $VTR_ROOT/ODIN_II/techlib/adff2dff.v;
  techmap -map $VTR_ROOT/ODIN_II/techlib/adffe2dff.v;
# To resolve Yosys internal indexed part-select circuitry
  techmap */t:$shift */t:$shiftx;

## Utilizing the "memory_bram" command and the Verilog design provided at "$VTR_ROOT/
  ODIN_II/techlib/mem_map.v"
```

(continues on next page)
## we could map Yosys memory blocks to BRAMs and ROMs before the Odin-II partial mapping phase.

## However, Yosys complains about expression widths more than 24 bits.


## Although we provided the required design files for this process (located in ODIN_II/techlib), we will handle

## memory blocks in the Odin-II BLIF elaborator and partial mapper.

# memory_bram -rules $VTR_ROOT/ODIN_II/techlib/mem_rules.txt
# techmap -map $VTR_ROOT/ODIN_II/techlib/mem_map.v;

# Transform the design into a new one with single top module
# flatten;
# Transforms pmux into trees of regular multiplexers
# pmuxtree;
# To possibly reduce words size
# wreduce;
# "undirven" to ensure there is no wire without drive
# "optmuxtree" removes dead branches, "opt_expr" performs constant folding,
# removes "undef" inputs from mux cells, and replaces muxes with buffers and inverters.
# "-noff" a potential option to remove all sdff and etc. Only dff will remain
# opt -undriven -full; opt_muxtree; opt_expr -mux_undef -mux_bool -fine;;;
# Make name convention more readable
# autoname;
# Print statistics
# stat;
# Output BLIF
# write_blif -param -impltf TCL_BLIF;

### Algorithm 1 - The Yosys+Odin-II Tcl Script File

## Yosys Elaboration

Yosys, as an open synthesis suite, reads the input digital circuits and creates the corresponding data structures, such as netlist and Abstract Syntax Tree (AST). As shown in Algorithm 1, the Tcl script, including the step-by-step generic coarse-grained synthesis commands required to be run by Yosys, is available at $VTR_ROOT/ODIN_II/regression_test/tools/synth.tcl. Utilizing these commands for the Yosys API inside the Odin-II codebase, the Yosys synthesizer performs the elaboration of the input digital design. The generic coarse-grained synthesis commands includes:

1. Parsing the hardware description Verilog files. The option \-nomem2reg prevents Yosys from exploding implicit memories to an array of registers. The option \-nolatches is used for both VTR primitives and input circuit design to avoid Yosys generating logic loops.
2. Checking that the design cells match the libraries and detecting the top module using hierarchy.
3. Translating the processes to netlist components such as multiplexers, flip-flops, and latches, by the proc command.
4. Performing extraction and optimization of finite state machines by the fsm command.
5. Collecting memories and their ports, then creating a multiport memory cell, by the memory_collect command.
6. Converting asynchronous memory ports to synchronous ones by merging ports and the related DFFs at their interfaces, using the memory_dff command.
7. Checking for errors like combinatorial loops, wires with multiple drivers and used wires without any driver by the check command.

After performing basic synthesis steps, the techmap command with the input adff2dff transforms DFFs with asynchronous reset to the synchronous form using the design provided by Yosys. The next command follows the same approach but with a modified version of the provided design file for DFFs with asynchronous reset and synchronous data enable signals. The last techmap command is in place to resolve the Yosys internal circuitries designed specifically for indexed part-select Verilog code. Since this step is mainly related to the Verilog elaboration, we ask Yosys to transform the $shift and $shiftx sub-circuits into a more straightforward representation.

The flatten command generates an output netlist with only one module, representing the HDL circuit design’s top module. The pmuxtree pass is used to transforms pmux, a sub-circuit representing parallel cases, into trees of regular multiplexers. The option wreduce performs possible word size reduction for operations to avoid propagating additional signals to the subsequent phases. In the autoname passes, Yosys generates an easy-to-read BLIF file by transforming signal names into a shorter format. This command removes some debugging information, such as the path to the source file, that Yosys inserts in names by default and generally gives easier-to-interpret names.

Then, the optimization pass is called to make the netlist ready for output. The option undriven ensures that all nets without a driver are removed, while the full optimization option is used to remove duplicated inputs in AND, OR and MUX gates. The opt_muxtree removes dead branches, opt_expr performs possible constant folding, in addition to removing undef inputs from mux cells and transforming muxes into buffers and inverters. Ultimately, we use the write_blif command to output the coarse-grained BLIF file. The option param prints some additional information about logic cells into the BLIF file, and the impltf option conceals the definition of primary netlist ports, i.e., VCC, GND and PAD, in the output.

**Note:** As earlier mentioned in User guide, the Yosys BLIF output process, i.e., write_blif, is handled by Yosys embedded API inside the Odin-II codebase. As a result, the last command is not required if a user would like to run the Yosys+Odin-II synthesizer using the Tcl script.

### BLIF Reader and Building the Netlist

In this step, Odin-II reads the Yosys generated coarse-grained BLIF file and creates the corresponding netlist data structure. Previously, the simulation option was only available when a BLIF file was passed to Odin-II. However, the option for performing the partial mapping phase on input BLIF files have become available with the Yosys+Odin-II integration. Using the --elaborator yosys command argument, the Odin-II BLIF reader reads the Yosys generated coarse-grained BLIF file. Additionally, if a coarse-grained BLIF file is already created, the user can perform Odin-II partial mapping on that using the -b design.blif --coarsen command arguments.

**Note:** The netlist can be viewed via graphviz using the command -G. The file will appear in the main directory under net.dot.

### BLIF Elaboration

As depicted in Yosys+Odin-II synthesis flow, the difference between fine-grained and coarse-grained netlists is the BLIF elaboration and partial mapping phases in Odin-II technology mapping flow. Technically, the infrastructure of Odin-II and Yosys differ from each other. As a result, the elaboration phase is performed on the input netlist when the input BLIF file is specified as a coarse-grained design to make it compatible with Odin-II partial mapper. As an example, Yosys generates complex DFFs, such as DFF with synchronous enable and reset, while Odin-II partial mapper only recognizes the simple DFF represented by .latch in BLIF. Therefore, these complex modules are required to be transformed into simpler designs using standard logic cells.
Partial Technology Mapping

During the partial mapping, Odin-II maps the logic using an architecture. If no architecture is passed in, Odin-II will create the soft logic and use LUTs for mapping. However, if an architecture is passed, Odin-II will map accordingly to the available hard blocks and LUTs. It uses a combination of soft logic and hard logic.

With the integration of Yosys+Odin-II, the Odin-II partial mapping features such as hard/soft logic trade-offs become available for a Yosys elaborated circuit. For instance, using optimization command arguments, a user can force the partial mapper to infer at least a user-defined percentage of multipliers in soft logic.

Simulator

The simulator of Odin-II takes an input vector file and creates an output vector file determined by the behaviour described in the Verilog file or BLIF file. This section is comprehensively described in the Yosys+Odin-II User guide and the Odin-II Simulation Arguments.

6.4.3 Regression Tests

Yosys+Odin-II utilizes the same set of benchmarks that Odin-II uses to assess its functionality. The benchmarks are comprised of Verilog files, input vector files and output vector files. Due to different approaches taken by Yosys+Odin-II and Odin-II synthesizers and some unsupported Verilog features by Odin-II, each synthesizer has its output vector files, named using XXX_odin_input/output and XXX_yosys_input/output name convention. In some cases, both output vectors are identical while they differ for the remainder of them. The configuration file calls upon each benchmark and synthesizes them with different architectures. The current regression tests’ Verilog files can be found in regression_test/benchmark/verilog.

Benchmarks

Except for the common benchmarks, located in regression_test/benchmark/verilog/common, which are Yosys basic benchmarks to verify its functionality, Yosys+Odin-II mainly utilizes Odin-II’s benchmarks that can be found in regression_test/benchmark/verilog/any_folder. Yosys+Odin-II has its own tasks and suites which are located in regression_test/benchmark/task/yosys+odin and regression_test/benchmark/suite/yosys+odin directories. Each benchmark is comprised of a Verilog file, an input vector file, and an output vector file. They are called upon during regression tests and synthesized with different architectures to be compared against the expected results. These tests are useful for developers to test the functionality of Yosys+Odin-II after implementing changes. The command make test ELABORATOR=yosys runs through all these tests, comparing the results to previously generated results.

Creating Regression Tests

The regression test creating process for Yosys+Odin-II is similar to that of Odin-II. For more information please see the Odin-II Regression Tests.
Example of a Yosys+Odin-II Configuration File

```verbatim
#######
# <title> benchmarks config
#######

# commands
regression_params=--include_default_arch
synthesis_params=--elaborator yosys --fflegalize --show_yosys_log
simulation_params= -L reset rst -H we
script_synthesis_params=--time_limit 3600s
script_simulation_params=--time_limit 3600s

# setup the architecture (standard architectures already available)
archs_dir=../vtr_flow/arch/timing
arch_list_add=k6_N10_40nm.xml
arch_list_add=k6_N10_mem32K_40nm.xml
arch_list_add=k6_frac_N10_frac_chain_mem32K_40nm.xml

# setup the circuits
circuits_dir=regression_test/benchmark/verilog/
circuit_list_add=<verilog file group>/*.vh
circuit_list_add=<verilog file group>/*.v

synthesis_parse_file=regression_test/parse_result/conf/synth.toml
simulation_parse_file=regression_test/parse_result/conf/sim.toml
```

Yosys+Odin-II Task Structure

The Yosys+Odin-II tasks follow the Odin-II task structure, which can be found at Creating a Task. The following diagram illustrates the structure of Yosys+Odin-II regression tests.

```
(continues on next page)
```
Yosys+Odin-II Suite Structure

The process of creating a suite for Yosys+Odin-II is completely identical to Odin-II. For more information on how Odin-II suites are created please see Creating a Suite. In the diagram below you can see the structure of the suites allocated for the Yosys+Odin-II tasks.

Note: To generate only coarse-grained BLIF files using Yosys elaborator, a new script named `run_yosys.sh`, is created in the `regression_test/tools` directory. The behaviour of this script is similar to the `verfiy_script.sh`. Indeed, it can perform the Yosys elaboration on a task or suite and generate corresponding coarse-grained BLIF files.

Regenerating Results

Note: BEFORE regenerating the result, run `make test ELABORATOR=yosys` to ensure any changes in the code don’t affect the results of benchmarks beside your own. If they do, the failing benchmarks will be listed.

Regenerating results is necessary if any regression test is changed (added benchmarks), if a regression test is added, or if a bug fix was implemented that changes the results of a regression test. The process of regenerating the expectation...
results for Yosys+Odin-II benchmarks using `verify_script.sh` is similar to Odin-II. Please visit Regenerating Results and Verify Script sections in the Odin-II developers guide to see how to regenerate expectation results using the verification script.

If the developer changes require regenerating all regression tests’ expectation results, you can use the makefile option created specifically for this purpose. In the following, you can see an example of how to regenerate the expectation results of all regression tests for a specific synthesizer. It is assumed the command is run in the Odin-II root directory.

```
make regenerate_expectation ELABORATOR=yosys
```

The default value of the ELABORATOR is considered `odin`, which means if you run the command mentioned above without specifying the elaborator, it will regenerate Odin-II expectation results.

### Yosys+Odin-II Regression Tests Directory Tree

```
benchmark
    ├── suite
    │     ├── complex_synthesis_suite
    │     ├── full_suite
    │     └── heavy_suite
    │         └── light_suite
    │             └── yosys+odin
    │                 ├── techmap_heavysuite
    │                 │         └── task.conf
    │                 │             └── synthesis_result.json
    │                 └── techmap_keywords_suite
    │                     └── task.conf
    │                             └── synthesis_result.json
    │                     └── techmap_lightsuite
    └── task
        ├── arch_sweep
        ├── c_functions
        └── FIR
            └── fpu
                └── full
                    └── keywords
                        └── koios
                            └── large
                                └── micro
                                    └── mixing_optimization
                                        └── operators
                                            └── preprocessor
                                                └── syntax
                                                    └── vtr
                                                        └── yosys+odin
                                                            └── arch_sweep
                                                                └── synthesis_result.json
                                                                    └── task.conf
```
FIR
  simulation_result.json
  synthesis_result.json
  task.conf
fpu
  hardlogic
    simulation_result.json
    task.conf
  softlogic
    simulation_result.json
    synthesis_result.json
    task.conf
full
  simulation_result.json
  synthesis_result.json
  task.conf
keywords
  always
  and
  assign
  at_parenthathese
  automatic
  begin_end
  buf
  case_endcase
  default
  defparam
  else
  for
  function_endfunction
  generate
  genvar
  if
  initial
  inout
  input_output
  integer
  localparam
  macromodule
  nand
  negedge
  nor
  not
  or
  parameter
  posedge
  reg
  signed_unsigned
  specify_endspecify
  specparam
  star
  task_endtask
while
wire
xnor
xor
koios
  synthesis_result.json
  task.conf
large
  synthesis_result.json
  task.conf
micro
  simulation_result.json
  synthesis_result.json
  task.conf
mixing_optimization
  config_file_half
    config_file_half.xml
    simulation_result.json
    synthesis_result.json
    task.conf
  mults_auto_full
    simulation_result.json
    synthesis_result.json
    task.conf
  mults_auto_half
    simulation_result.json
    synthesis_result.json
    task.conf
  mults_auto_none
    simulation_result.json
    synthesis_result.json
    task.conf
operators
  simulation_result.json
  synthesis_result.json
  task.conf
preprocessor
  simulation_result.json
  synthesis_result.json
  task.conf
syntax
  simulation_result.json
  synthesis_result.json
  task.conf
vtr
  synthesis_result.json
  task.conf
third_party
verilog
6.4.4 Testing Yosys+Odin-II

The testing process, including the simulation, is similar to Odin-II. For more information on how to test and evaluate the Odin-II and Yosys+Odin-II please visit TESTING ODIN II.
Yosys is a framework for Verilog RTL synthesis, used as one of three VTR front-ends to perform logic synthesis, elaboration, and converting a subset of the Verilog Hardware Description Language (HDL) into a BLIF netlist.

7.1 Quickstart

7.1.1 Prerequisites

- ctags
- bison
- flex
- gcc 5.x
- cmake 3.9 (minimum version)
- time
- cairo
- gawk
7.1.2 Building

To build the VTR flow with the Yosys front-end you may use the VTR Makefile wrapper, by calling the `make CMAKE_PARAMS="-DWITH_YOSYS=ON"` command in the `$VTR_ROOT` directory. The compile flag `-DWITH_YOSYS=ON` should be passed to the CMake parameters to enable Yosys compilation process.

**Note:** Yosys uses Makefile as its build system. Since CMake provides portable, cross-platform build systems with many useful features, we provide a CMake wrapper to successfully embeds the Yosys library into the VTR flow.

7.1.3 Basic Usage

To run the VTR flow with the Yosys front-end, you would need to run the `run_vtr_flow.py` script with the start stage specified as `Yosys`.

```
./run_vtr_flow `PATH_TO_VERILOG_FILE.v` `PATH_TO_ARCH_FILE.xml` -start yosys
```

7.2 Developers Guide

The approach of utilizing Yosys in VTR is mainly driven by what Eddie Hung proposed for the VTR-to-Bitstream (VTB), based upon VTR 7. Although some files, such as `yosys_models.v` and `multiply.v`, are directly copied from the VTB project, the other files have been subjected to a few changes due to significant alterations from VTR 7 to the current version of VTR. Additionally, Hung’s approach was specifically proposed for Xilinx Vertex-6 architecture. As a result, relevant changes to the remainder of Yosys library files have been applied to make them compatible with the current VTR version and support routine architectures used in the VTR regression tests.
7.2.1 What is new compared to the VTB files?

Changes applied to the VTB files are outlined as follows:

1. Replacing Vertix-6 adder black-box *(xadder)* with the conventional adder used in the current version of VTR.
2. If required, performing a recursive depth splitting for memory hard blocks, i.e., *single_port_ram* and *dual_port_ram*, to make them adaptable with the VTR flow configurations.
3. Converting DFFs with asynchronous reset to synchronous form using *adff2dffe.v* and *adffe2dffe.v*.
4. Adding the *dffunmap* command to Yosys synthesis script to transform complex DFF sub-circuits, such as SDFFE (DFF with synchronous reset and enable), to their soft logic implementation, i.e., the combination of multiplexers and latches.
5. Removing the ABC commands from the Yosys synthesis script and letting the VTR flow’s ABC stage performs the technology mapping.

Note: The LUT size is considered the one defined in the architecture file as the same as the regular VTR flow.

7.2.2 How to add new changes?

The Yosys synthesis commands, including the generic synthesis and additional VTR specific configurations, are provided in *synthesis.ys*. To make changes in the overall Yosys synthesis flow, the *synthesis.ys* script is perhaps the first file developers may require to change.

Moreover, the *yosys_models.v* file includes the required definitions for Yosys to how it should infer implicit memories and instantiate arithmetic operations, such as addition, subtraction, and multiplication. Therefore, to alter these behaviours or add more regulations such as how Yosys should behave when facing other arithmetic operations, for example modulo and division, the *yosys_models.v* Verilog file is required to be modified.

Except for *single_port_ram.v* and *dual_port_ram.v* Verilog files that perform the depth splitting process, the other files are defined as black-box, i.e., their declarations are required while no definition is needed. To add new black-box components, developers should first provide the corresponding Verilog files similar to the *adder.v*. Then, a new *read_verilog -lib TTT/NEW_BB.v* command should be added to the Yosys synthesis script. If there is an implicit inference of the new black-box component, the *yosys_models.v* Verilog file must also be modified, as mentioned earlier.

7.2.3 Yosys Synthesis Script File

```plaintext
# XXX (input circuit) is replaced with filename by the run_vtr_flow script
read_verilog -nolatches XXX

# These commands follow the generic 'synth'
# command script inside Yosys
# The -libdir argument allows Yosys to search the current
# directory for any definitions to modules it doesn't know
# about, such as hand-instantiated (not inferred) memories
hierarchy -check -auto-top -libdir .
proc

# Check that there are no combinational loops
scc -select
select -assert-none %
```

(continues on next page)
select -clear

opt_expr
opt_clean
check
opt -nodffe -nosdff
fsm
opt
wreduce
peepopt
opt_clean
share
opt
memory -nomap
opt -full

# Transform all async FFs into synchronous ones
techmap -map +/adff2dff.v
techmap -map TTT/../../../ODIN_II/techlib/adffe2dff.v

# Map multipliers, DSPs, and add/subtracts according to yosys_models.v
techmap -map YYY */t:$mul */t:$mem */t:$sub */t:$add
opt -fast -full
memory_map
# Taking care to remove any undefined muxes that
# are introduced to promote resource sharing
opt -full

# Then techmap all other `complex` blocks into basic
# (lookup table) logic
techmap
opt -fast

# We read the definitions for all the VTR primitives
# as blackboxes
read_verilog -lib TTT/adder.v
read_verilog -lib TTT/multiply.v
read_verilog -lib SSS  #(SSS) will be replaced by single_port_ram.v by python script
read_verilog -lib DDD  #(DDD) will be replaced by dual_port_ram.v by python script

# Rename singlePortRam to single_port_ram
# Rename dualPortRam to dualZ_port_ram
# rename function of Yosys not work here
# since it may outcome hierarchy error
read_verilog SSR  #(SSR) will be replaced by spram_rename.v by python script
read_verilog DDR  #(DDR) will be replaced by dpram_rename.v by python script

# Flatten the netlist
flatten
# Turn all DFFs into simple latches
# Lastly, check the hierarchy for any unknown modules, # and purge all modules (including blackboxes) that # aren't used hierarchy -check -purge_lib tee -o /dev/stdout stat

autoname

# Then write it out as a blif file, remembering to call # the internal `$true`/`$false` signals vcc/gnd, but # switch `-impltf` doesn't output them # ZZZ will be replaced by run_vtr_flow.pl write_blif -true + vcc -false + gnd -undef + unconn -blackbox ZZZ

**Algorithm 1** - The Yosys Tcl Script File

## 7.3 Structure

### 7.3.1 Structure of Yosys Synthesis Files and the Yosys External Library Library

```
$VTR_ROOT
  vtr_flow
  |
  misc
  |
  yosyslib
  |
  adder.v
  |
  dpram_rename.v
  |
  dual_port_ram.v
  |
  multiply.v
  |
  single_port_ram.v
  |
  spram_rename.v
  |
  synthesis.ys
  |
  yosys_models.v

  libs
  |
  EXTERNAL
  |
  libyosys
  |
  backends
  |
  examples
  |
  frontends
  |
  guidelines
  |
  kernel
  |
  libs
  |
  manual
  |
  misc
  |
  passes
  |
  techlibs
  |
  tests
```
CHAPTER EIGHT

ABC

ABC is included with in VTR to perform technology independant logic optimization and technology mapping. ABC is developed at UC Berkeley, see the ABC homepage for details.
9.1 Design Flow Tutorials

These tutorials describe how to run the VTR design flow.

9.1.1 Basic Design Flow Tutorial

The following steps show you how to run the VTR design flow to map a sample circuit to an FPGA architecture containing embedded memories and multipliers:

1. From the $VTR_ROOT$, move to the vtr_flow/tasks/regression_tests/vtr_reg_basic directory, and run:

```
../../../scripts/run_vtr_task.py basic_no_timing
```

or:

```
$VTR_ROOT/vtr_flow/scripts/run_vtr_task.py basic_no_timing
```

The subdirectory regression_tests/vtr_reg_basic contains tests that are to be run before each commit. They check for basic functionality to make sure nothing was extremely out of order. This command runs the VTR flow on a set of circuits and a single architecture. The files generated from the run are stored in basic_no_timing/run[#] where [#] is the number of runs you have done. If this is your first time running the flow, the results will be stored in basic_no_timing/run001. When the script completes, enter the following command:

```
../../../scripts/python_libs/vtr/parse_vtr_task.py basic_no_timing/
```

This parses out the information of the VTR run and outputs the results in a text file called run[#]/parse_results.txt.

More info on how to run the flow on multiple circuits and architectures along with different options later. Before that, we need to ensure that the run that you have done works.

2. The basic_no_timing comes with golden results that you can use to check for correctness. To do this check, enter the following command:

```
../../../scripts/python_libs/vtr/parse_vtr_task.py -check_golden basic_no_timing
```

It should return: basic_no_timing...[Pass]
Note: Due to the nature of the algorithms employed, the measurements that you get may not match exactly with the golden measurements. We included margins in our scripts to account for that noise during the check. We also included runtime estimates based on our machine. The actual runtimes that you get may differ dramatically from these values.

3. To see precisely which see circuits, architecture, and CAD flow was employed by the run, look at vtr_flow/tasks/regression_tests/vtr_reg_basic/config.txt. Inside this directory, the config.txt file contains the circuits and architecture file employed in the run.

Some also contain a golden_results.txt file that is used by the scripts to check for correctness.

The $VTR_ROOT/vtr_flow/scripts/run_vtr_flow.py script describes the CAD flow employed in the test. You can modify the flow by editing this script.

At this point, feel free to run any of the tasks with the prefix vtr_reg. These are regression tests included with the flow that test various combinations of flows, architectures, and benchmarks. Refer to the README for a description what each task aims to test.

4. For more information on how the vtr_flow infrastructure works (and how to add the tests that you want to do to this infrastructure) see Tasks.

9.2 Architecture Modeling

This page provides information on the FPGA architecture description language used by VPR. This page is geared towards both new and experienced users of vpr.

New users may wish to consult the conference paper that introduces the language [LAR11]. This paper describes the motivation behind this new language as well as a short tutorial on how to use the language to describe different complex blocks of an FPGA.

New and experienced users alike should consult the detailed Architecture Reference which serves to documents every property of the language.

Multiple examples of how this language can be used to describe different types of complex blocks are provided as follows:

Complete Architecture Description Walkthrough Examples:

9.2.1 Classic Soft Logic Block Tutorial

The following is an example on how to use the VPR architecture description languange to describe a classical academic soft logic block. First we provide a step-by-step explanation on how to construct the logic block. Afterwards, we present the complete code for the logic block.

Fig. 9.1 shows an example of a classical soft logic block found in academic FPGA literature. This block consists of N Basic Logic Elements (BLEs). The BLE inputs can come from either the inputs to the logic block or from other BLEs within the logic block via a full crossbar. The logic block in this figure has I general inputs, one clock input, and N outputs (where each output corresponds to a BLE). A BLE can implement three configurations: a K-input look-up table (K-LUT), a flip-flop, or a K-LUT followed by a flip-flop. The structure of a classical soft logic block results in a property known as logical equivalence for certain groupings of input/output pins. Logically equivalent pins means that connections to those pins can be swapped without changing functionality. For example, the input to AND gates are logically equivalent while the inputs to a 4-bit adders are not logically equivalent. In the case of a classical soft logic block, all input pins are logically equivalent (due to the fully populated crossbar) and all output pins are logically equivalent (because one can swap any two BLEs without changing functionality). Logical equivalence is important
because it enables the CAD tools to make optimizations especially during routing. We describe a classical soft logic block with \( N = 10, I = 22, \) and \( K = 4 \) below.

First, a complex block `pb_type` called CLB is declared with appropriate input, output and clock ports. Logical equivalence is labelled at ports where it applies:

```xml
<pb_type name="clb">
  <input name="I" num_pins="22" equivalent="full"/>
  <output name="O" num_pins="10" equivalent="instance"/>
  <clock name="clk" equivalent="false"/>
</pb_type>
```

A CLB contains 10 BLEs. Each BLE has 4 inputs, one output, and one clock. A BLE block and its inputs and outputs are specified as follows:

```xml
<pb_type name="ble" num_pb="10">
  <input name="in" num_pins="4"/>
  <output name="out" num_pins="1"/>
  <clock name="clk"/>
</pb_type>
```

A BLE consists of one LUT and one flip-flop (FF). Both of these are primitives. Recall that primitive physical blocks must have a `blif_model` attribute that matches with the model name in the BLIF input netlist. For the LUT, the model is `.names` in BLIF. For the FF, the model is `.latch` in BLIF. The class construct denotes that these are special (common) primitives. The primitives contained in the BLE are specified as:

```xml
<pb_type name="lut_4" blif_model=".names" num_pb="1" class="lut">
  <input name="in" num_pins="4" port_class="lut_in"/>
  <output name="out" num_pins="1" port_class="lut_out"/>
</pb_type>
<pb_type name="ff" blif_model=".latch" num_pb="1" class="flipflop">
  <input name="D" num_pins="1" port_class="D"/>
  <output name="Q" num_pins="1" port_class="Q"/>
  <clock name="clk" port_class="clock"/>
</pb_type>
```

Fig. 9.2 shows the ports of the BLE with the input and output pin sets. The inputs to the LUT and flip-flop are direct connections. The multiplexer allows the BLE output to be either the LUT output or the flip-flop output. The code to specify the interconnect is:

```xml
<interconnect>
  <direct input="lut_4.out" output="ff.D"/>
  <direct input="ble.in" output="lut_4.in"/>
  <mux input="ff.Q lut_4.out" output="ble.out"/>
</interconnect>
```
The CLB interconnect is then modeled (see Fig. 9.1). The inputs to the 10 BLEs (ble[9:0].in) can be connected to any of the CLB inputs (clb.I) or any of the BLE outputs (ble[9:0].out) by using a full crossbar. The clock of the CLB is wired to multiple BLE clocks, and is modeled as a full crossbar. The outputs of the BLEs have direct wired connections to the outputs of the CLB and this is specified using one direct tag. The CLB interconnect specification is:

```
<interconnect>
  <complete input="{clb.I ble[9:0].out}" output="ble[9:0].in"/>
  <complete input="clb.clk" output="ble[9:0].clk"/>
  <direct input="ble[9:0].out" output="clb.O"/>
</interconnect>
```

Finally, we model the connectivity between the CLB and the general FPGA fabric (recall that a CLB communicates with other CLBs and I/Os using general-purpose interconnect). The ratio of tracks that a particular input/output pin of the CLB connects to is defined by fc_in/fc_out. In this example, a fc_in of 0.15 means that each input pin connects to 15% of the available routing tracks in the external-to-CLB routing channel adjacent to that pin. The pinlocations tag is used to associate pins on the CLB with which side of the logic block pins reside on where the pattern spread corresponds to evenly spreading out the pins on all sides of the CLB in a round-robin fashion. In this example, the CLB has a total of 33 pins (22 input pins, 10 output pins, 1 clock pin) so 8 pins are assigned to all sides of the CLB except one side which gets assigned 9 pins.

```
<!-- Describe complex block relation with FPGA -->

<fc_in type="frac">0.150000</fc_in>
<fc_out type="frac">0.125000</fc_out>

<pinlocations pattern="spread"/>
</pb_type>
```

**Classic Soft Logic Block Complete Example**

```
<!--
Example of a classical FPGA soft logic block with
N = 10, K = 4, I = 22, O = 10
BLEs consisting of a single LUT followed by a flip-flop that can be bypassed
-->

<pb_type name="clb"/>
```
<input name="I" num_pins="22" equivalent="full"/>
<output name="O" num_pins="10" equivalent="instance"/>
<clock name="clk" equivalent="false"/>

<pb_type name="ble" num_pb="10">
  <input name="in" num_pins="4"/>
  <output name="out" num_pins="1"/>
  <clock name="clk"/>
</pb_type>

<pb_type name="lut_4" blif_model=".names" num_pb="1" class="lut">
  <input name="in" num_pins="4" port_class="lut_in"/>
  <output name="out" num_pins="1" port_class="lut_out"/>
</pb_type>

<pb_type name="ff" blif_model=".latch" num_pb="1" class="flipflop">
  <input name="D" num_pins="1" port_class="D"/>
  <output name="Q" num_pins="1" port_class="Q"/>
  <clock name="clk" port_class="clock"/>
</pb_type>

<interconnect>
  <direct input="lut_4.out" output="ff.D"/>
  <direct input="ble.in" output="lut_4.in"/>
  <mux input="ff.Q lut_4.out" output="ble.out"/>
  <direct input="ble.clk" output="ff.clk"/>
</interconnect>

<interconnect>
  <complete input="clb.I ble[9:0].out" output="ble[9:0].in"/>
  <complete input="clb.clk" output="ble[9:0].clk"/>
  <direct input="ble[9:0].out" output="clb.0"/>
</interconnect>

<!-- Describe complex block relation with FPGA -->

<fc_in type="frac">0.150000</fc_in>
<fc_out type="frac">0.125000</fc_out>

<pinlocations pattern="spread"/>
9.2.2 Multi-mode Logic Block Tutorial

This tutorial aims to introduce how to build a representative multi-mode logic block by exploiting VPR architecture description language, as well as debugging tips to guarantee each mode of a logic block is functional.

Definition

Modern FPGA logic blocks are designed to operate in various modes, so as to provide best performance for different applications. VPR offers enriched syntax to support highly flexible multi-mode logic block architecture.

Fig. 9.3 shows the physical implementation of a Fracturable Logic Element (FLE), which consists of a fracturable 6-input Look-Up Table (LUT), two Flip-flops (FFs) and routing multiplexers to select between combinational and sequential outputs.

![Fig. 9.3: Schematic of a fracturable logic element](image)

The FLE in Fig. 9.3 can operate in two different modes: (a) dual 5-input LUT mode (see Fig. 9.4); and (b) single 6-input LUT mode (see Fig. 9.5). Note that each operating mode does not change the physical implementation of FLE but uses part of the programmable resources.

Architecture Description

To accurately model the operating modes of the FLE, we will use the syntax `<pb_type>` and `<mode>` in architecture description language.

```
<!-- Multi-mode Fracturable Logic Element definition begin -->
<pb_type
  name="fle" num_pb="10">
  <input
    name="in" num_pins="6"/>
  <output
    name="out" num_pins="2"/>
  <clock
    name="clk" num_pins="1"/>

<!-- Dual 5-input LUT mode definition begin -->
<mode
  name="n2_lut5">
  <!-- Detailed definition of the dual 5-input LUT mode -->
```

(continues on next page)
**Fig. 9.4:** Simplified organization when the FLE in **Fig. 9.3** operates in dual 5-input LUT mode

**Fig. 9.5:** Simplified organization when the FLE in **Fig. 9.3** operates in 6-input LUT mode
In the above XML codes, we define a `<pb_type>` for the FLE by following the port organization in Fig. 9.3. Under the `<pb_type>`, we create two modes, n2_lut5 and n1_lut6, corresponding to the two operating modes as shown in Fig. 9.4 and Fig. 9.5. Note that we focus on operating modes here, which are sufficient to perform architecture evaluation.

Under the dual 5-input LUT mode, we can define `<pb_type>` and `<interconnect>` to model the schematic in Fig. 9.4.

```xml
<!-- Dual 5-input LUT mode definition begin -->
<mode name="n2_lut5">
  <pb_type name="lut5inter" num_pb="1">
    <input name="in" num_pins="5"/>
    <output name="out" num_pins="2"/>
    <clock name="clk" num_pins="1"/>
    <!-- Define two LUT5 + FF pairs (num_pb=2) corresponding to :numref:`fig_frac_lut_le_dual_lut5_mode` -->
    <pb_type name="ble5" num_pb="2">
      <input name="in" num_pins="5"/>
      <output name="out" num_pins="1"/>
      <clock name="clk" num_pins="1"/>
      <!-- Define the LUT -->
      <pb_type name="lut5" blif_model=".names" num_pb="1" class="lut">
        <input name="in" num_pins="5" port_class="lut_in"/>
        <output name="out" num_pins="1" port_class="lut_out"/>
        <!-- LUT timing using delay matrix -->
        <!-- These are the physical delay inputs on a Stratix IV LUT but because VPR cannot do LUT rebalancing, we instead take the average of these numbers to get more stable results
        82e-12
        173e-12
        261e-12
        263e-12
        398e-12
        -->
        <delay_matrix type="max" in_port="lut5.in" out_port="lut5.out">
          235e-12
          235e-12
          235e-12
          235e-12
          235e-12
        </delay_matrix>
      </pb_type>
    </pb_type>
  </pb_type>
</mode>
<!-- Dual 5-input LUT mode definition end -->
```

(continues on next page)
Under the 6-input LUT mode, we can define `<pb_type>` and `<interconnect>` to model the schematic in Fig. 9.5.

```xml
<!-- 6-LUT mode definition begin -->
<mode name="n1_lut6" disable_packing="false">
  <!-- Define 6-LUT mode, consisting of a LUT6 + FF pair (num_pb=1) corresponding to
   numref: fig_frac_lut_le_lut6_mode -->
  <pb_type name="ble6" num_pb="1">
    <input name="in" num_pins="6"/>
    <output name="out" num_pins="1"/>
    <clock name="clk" num_pins="1"/>
  </pb_type>
  <pb_type name="lut6" blif_model=".names" num_pb="1" class="lut">
    <input name="in" num_pins="6" port_class="lut_in"/>
    <output name="out" num_pins="1" port_class="lut_out"/>
  </pb_type>
</mode>
```

(continues on next page)
<!-- These are the physical delay inputs on a Stratix IV LUT but because VPR cannot do LUT rebalancing, we instead take the average of these numbers to get more stable results

82e-12
173e-12
261e-12
263e-12
398e-12
397e-12

-->

<delay_matrix type="max" in_port="lut6.in" out_port="lut6.out">
261e-12
261e-12
261e-12
261e-12
261e-12
261e-12
</delay_matrix>

<!-- Define flip-flop -->

<pb_type name="ff" blif_model=".latch" num_pb="1" class="flipflop">
<input name="D" num_pins="1" port_class="D"/>
<output name="Q" num_pins="1" port_class="Q"/>
<clock name="clk" num_pins="1" port_class="clock"/>
<T_setup value="66e-12" port="ff.D" clock="clk"/>
<T_clock_to_Q max="124e-12" port="ff.Q" clock="clk"/>
</pb_type>

<interconnect>

<direct name="direct1" input="ble6.in" output="lut6[0:0].in"/>
<direct name="direct2" input="lut6.out" output="ff.D"/>
<!-- Advanced user option that tells CAD tool to find LUT+FF pairs in netlist -->
<pack_pattern name="ble6" in_port="lut6.out" out_port="ff.D"/>
</direct>

<direct name="direct3" input="ble6.clk" output="ff.clk"/>
<mux name="mux1" input="ff.Q lut6.out" output="ble6.out">
<!-- LUT to output is faster than FF to output on a Stratix IV -->
<delay_constant max="25e-12" in_port="lut6.out" out_port="ble6.out"/>
<delay_constant max="45e-12" in_port="ff.Q" out_port="ble6.out"/>
</mux>
</interconnect>

<interconnect>

<direct name="direct1" input="fle.in" output="ble6.in"/>
<direct name="direct2" input="ble6.out" output="fle.out[0:0]"/>
<direct name="direct3" input="fle.clk" output="ble6.clk"/>
</interconnect>
</mode>

<!-- 6-LUT mode definition end -->

Full example can be found at link.
Validation in packer

After finishing the architecture description, the next step is to validate that VPR can map logic to each operating mode. Since VPR packer will exhaustively try each operating mode and finally map logic to one of it. As long as there is an operating mode that is feasible for mapping, VPR will complete packing without errors. However, this may shadow the problems for other operating modes. It is entirely possible that an operating mode is not defined correctly and is always dropped by VPR during packing. Therefore, it is necessary to validate the correctness of each operating mode. To efficiently reach the goal, we will temporarily apply the syntax `disable_packing` to specific modes, so as to narrow down the search space.

First, we can disable the dual 5-input LUT mode for packer, by changing

```
<mode name="n2_lut5">
```

to

```
<mode name="n2_lut5" disable_packing="true">
```

As a result, VPR packer will only consider the 6-input LUT mode during packing. We can try a benchmark `mult_2x2.blif` by following the design flow tutorial Basic Design Flow Tutorial. If the flow-run succeed, it means that the 6-input LUT mode is being successfully used by the packer.

Then, we can enable the dual 5-input LUT mode for packer, and disable the 6-input LUT mode, by changing

```
<mode name="n2_lut5" disable_packing="true">
<mode name="n1_lut6">
```

to

```
<mode name="n2_lut5">
<mode name="n1_lut6" disable_packing="true">
```

In this case, VPR packer will consider the dual 5-input LUT mode during packing. We can again try the same benchmark `mult_2x2.blif` by following the design flow tutorial Basic Design Flow Tutorial. If the flow-run succeed, it means that the dual 5-input LUT mode is being successfully used by the packer.

Finally, after having validated that both operating modes are being successfully used by the packer, we can re-enable both operating modes by changing to

```
<mode name="n2_lut5">
<mode name="n1_lut6">
```

Now, VPR packer will try to choose the best operating mode to use.

**Tips for Debugging**

When packing fails on a multi-mode logic block, the following procedures are recommended to quickly spot the bugs.

- Apply `disable_packing` to all the modes, except the one you suspect to be problematic. In the example of this tutorial, you may disable the packing for mode `n2_lut5` and focus on debugging mode `n1_lut6`.

```
<mode name="n2_lut5" disable_packing="true">
<mode name="n1_lut6" disable_packing="false">
```
• Turn on verbose output of packer --pack_verbosity (see details in Packing Options. Recommend to use a higher verbosity number than the default value, e.g., 5. Consider the example blif and architecture in this tutorial, you may execute vpr with

```
vpr k6_frac_N10_40nm.xml mult_2x2.blif --pack_verbosity 5
```

• Packer will show detailed information about why it fails. For example:

```
FAILED Detailed Routing Legality
Placed atom 'p3' (.names) at clb[0][default]/fle[4][n1_lut6]/ble6[0][default]/
  lut6[0][lut6]/lut[0]
(921:cluster-external source (LB_SOURCE)--1:'clb[0].I[1]' (1:'clb[0].I[1]'--62:
  'fle[0].in[1]' (62:'fle[0].in[1]'--123:'ble6[0].in[1]' (123:'ble6[0].in[1]'--
  131:'lut6[0].in[1]' (131:'lut6[0].in[1]'--138:'lut[0].in[1]' (138:'lut[0].in[1]
  -->930:cluster-internal sink (LB_SINK accessible via architecture pins: clb[0]/
  'fle[0]/ble6[0]/lut6[0]/lut[0].in[0], clb[0]/fle[0]/ble6[0]/lut6[0]/lut[0].in[1],
  clb[0]/fle[0]/ble6[0]/lut6[0]/lut[0].in[2], clb[0]/fle[0]/ble6[0]/lut6[0]/lut[0].
  in[3], clb[0]/fle[0]/ble6[0]/lut6[0]/lut[0].in[4], clb[0]/fle[0]/ble6[0]/lut6[0]/
  lut[0].in[5]))
```

Which indicates that input ports of <pb_type name=lut6> in the mode n1_lut6 may be dangling, and thus leads to failures in routing stage of packing.

• You may modify the architecture description and re-run vpr until packing succeeds.

• Move on to the next mode you will debug and repeat from the first step.

The debugging tips are not only applicable to the example showed in this tutorial but rather general to any multi-mode logic block architecture.

### 9.2.3 Configurable Memory Bus-Based Tutorial

**Warning:** The description in this tutorial is not yet supported by CAD tools due to bus-based routing.

**See also:**

*Configurable Memory Block Example* for a supported version.

Configurable memories are found in today’s commercial FPGAs for two primary reasons:

1. Memories are found in a variety of different applications including image processing, soft processors, etc and

2. Implementing memories in soft logic (LUTs and flip-flops) is very costly in terms of area.

Thus it is important for modern FPGA architects be able to describe the specific properties of the configurable memory that they want to investigate. The following is an example on how to use the langauge to describe a configurable memory block. First we provide a step-by-step explanation on how to construct the memory block. Afterwards, we present the complete code for the memory block.

Fig. 9.6 shows an example of a single-ported memory. This memory block can support multiple different width and depth combinations (called aspect ratios). The inputs can be either registered or combinational. Similarly, the outputs can be either registered or combinational. Also, each memory configuration has groups of pins called ports that share common properties. Examples of these ports include address ports, data ports, write enable, and clock. In this example, the block memory has the following three configurations: 2048x1, 1024x2, and 512x4, which will be modeled using modes. We begin by declaring the reconfigurable block RAM along with its I/O as follows:
The input and output registers are defined as 2 sets of bypassable flip-flops at the I/Os of the block RAM. There are a total of 16 inputs that can be registered as a bus so 16 flip-flops (for the 11 address lines, 4 data lines, and 1 write enable), named \textit{ff\_reg\_in}, must be declared. There are 4 output bits that can also be registered, so 4 flip-flops (named \textit{ff\_reg\_out}) are declared:

```
<pb_type name="ff\_reg\_in" blif_model=".latch" num_pb="16" class="flipflop">
  <input name="D" num_pins="1" port_class="D"/>
  <output name="Q" num_pins="1" port_class="Q"/>
  <clock name="clk" port_class="clock"/>
</pb_type>
<pb_type name="ff\_reg\_out" blif_model=".latch" num_pb="4" class="flipflop">
  <input name="D" num_pins="1" port_class="D"/>
  <output name="Q" num_pins="1" port_class="Q"/>
  <clock name="clk" port_class="clock"/>
</pb_type>
```

Each aspect ratio of the memory is declared as a mode within the memory physical block type as shown below. Also, observe that since memories are one of the special (common) primitives, they each have a \textit{class} attribute:

```
<pb_type name="mem\_reconfig" num_pb="1">
  <input name="addr" num_pins="11"/>
  <input name="din" num_pins="4"/>
</pb_type>
```

(continues on next page)
The top-level interconnect structure of the memory SPCB is shown in Fig. 9.8. The inputs of the SPCB can connect to input registers or bypass the registers and connect to the combinational memory directly. Similarly, the outputs of the combinational memory can either be registered or connect directly to the outputs. The description of the interconnect is as follows:

Fig. 9.8: Interconnect within the configurable memory block.
The interconnect for the bypassable registers is complex and so we provide a more detailed explanation. First, consider the input registers. Line 2 shows that the SPCB inputs drive the input flip-flops using direct wired connections. Then, in line 5, the combinational configurable memory inputs \{mem_reconfig.wen \ mem_reconfig.din \ mem_reconfig.addr\} either come from the flip-flops \texttt{ff.reg.in[15:0].Q} or from the SPCB inputs \texttt{block_RAM.wen block_RAM.din[3:0] block_RAM.addr[10:0]} through a 16-bit 2-to-1 bus-based mux. Thus completing the bypassable input register interconnect. A similar scheme is used at the outputs to ensure that either all outputs are registered or none at all. Finally, we model the relationship of the memory block with the general FPGA fabric. The ratio of tracks that a particular input/output pin of the CLB connects to is defined by \texttt{fc_in/fc_out}. The pinlocations describes which side of the logic block pins reside on where the pattern spread describes evenly spreading out the pins on all sides of the CLB in a round-robin fashion.

```xml
<fc_in type="frac">0.150000</fc_in>
<fc_out type="frac">0.125000</fc_out>
<pinlocations pattern="spread"/>
```

**Configurable Memory Bus-Based Complete Example**

```xml
<pb_type name="block_RAM">
  <input name="addr" num_pins="11" equivalent="false"/>
  <input name="din" num_pins="4" equivalent="false"/>
  <input name="wen" num_pins="1" equivalent="false"/>
  <output name="dout" num_pins="4" equivalent="false"/>
  <clock name="clk" equivalent="false"/>
</pb_type>
```

```xml
<pb_type name="ff_reg_in" blif_model=".latch" num_pb="16" class="flipflop">
  <input name="D" num_pins="1" port_class="D"/>
  <output name="Q" num_pins="1" port_class="Q"/>
  <clock name="clk" port_class="clock"/>
</pb_type>
```

```xml
<pb_type name="ff_reg_out" blif_model=".latch" num_pb="4" class="flipflop">
  <input name="D" num_pins="1" port_class="D"/>
  <output name="Q" num_pins="1" port_class="Q"/>
  <clock name="clk" port_class="clock"/>
</pb_type>
```

```xml
<pb_type name="mem_reconfig" num_pb="1">
  <input name="addr" num_pins="11"/>
  <input name="din" num_pins="4"/>
  <input name="wen" num_pins="1"/>
  <output name="dout" num_pins="4"/>
```

(continues on next page)
<!-- Declare a 2048x1 memory type -->
<mode name="mem_2048x1_mode">
    <pb_type name="mem_2048x1" blif_model=".subckt sp_mem" class="memory">
        <input name="addr" num_pins="11" port_class="address"/>
        <input name="din" num_pins="1" port_class="data_in"/>
        <input name="wen" num_pins="1" port_class="write_en"/>
        <output name="dout" num_pins="1" port_class="data_out"/>
    </pb_type>
    <interconnect>
        <direct input="mem_reconfig.addr[10:0]" output="mem_2048x1.addr"/>
        <direct input="mem_reconfig.din[0]" output="mem_2048x1.din"/>
        <direct input="mem_reconfig.wen" output="mem_2048x1.wen"/>
        <direct input="mem_2048x1.dout" output="mem_reconfig.dout[0]"/>
    </interconnect>
</mode>

<!-- Declare a 1024x2 memory type -->
<mode name="mem_1024x2_mode">
    <pb_type name="mem_1024x2" blif_model=".subckt sp_mem" class="memory">
        <input name="addr" num_pins="10" port_class="address"/>
        <input name="din" num_pins="2" port_class="data_in"/>
        <input name="wen" num_pins="1" port_class="write_en"/>
        <output name="dout" num_pins="2" port_class="data_out"/>
    </pb_type>
    <interconnect>
        <direct input="mem_reconfig.addr[9:0]" output="mem_1024x2.addr"/>
        <direct input="mem_reconfig.din[1:0]" output="mem_1024x2.din"/>
        <direct input="mem_reconfig.wen" output="mem_1024x2.wen"/>
        <direct input="mem_1024x2.dout" output="mem_reconfig.dout[1:0]"/>
    </interconnect>
</mode>

<!-- Declare a 512x4 memory type -->
<mode name="mem_512x4_mode">
    <pb_type name="mem_512x4" blif_model=".subckt sp_mem" class="memory">
        <input name="addr" num_pins="9" port_class="address"/>
        <input name="din" num_pins="4" port_class="data_in"/>
        <input name="wen" num_pins="1" port_class="write_en"/>
        <output name="dout" num_pins="4" port_class="data_out"/>
    </pb_type>
    <interconnect>
        <direct input="mem_reconfig.addr[8:0]" output="mem_512x4.addr"/>
        <direct input="mem_reconfig.din[3:0]" output="mem_512x4.din"/>
        <direct input="mem_reconfig.wen" output="mem_512x4.wen"/>
        <direct input="mem_512x4.dout" output="mem_reconfig.dout[3:0]"/>
    </interconnect>
</mode>
</pb_type>
</interconnect>
9.2.4 Fracturable Multiplier Bus-Based Tutorial

**Warning:** The description in this tutorial is not yet supported by CAD tools due to bus-based routing.

See also:

*Fracturable Multiplier Example* for a supported version.

Configurable multipliers are found in today’s commercial FPGAs for two primary reasons:

1. Multipliers are found in a variety of different applications including DSP, soft processors, scientific computing, etc and
2. Implementing multipliers in soft logic is very area expensive.

Thus it is important for modern FPGA architects be able to describe the specific properties of the configurable multiplier that they want to investigate. The following is an example on how to use the VPR architecture description language to describe a common type of configurable multiplier called a fracturable multiplier shown in Fig. 9.9. We first give a step-by-step description on how to construct the multiplier block followed by a complete example.

![Fig. 9.9: Model of a fracturable multiplier block](image)

The large `block_mult` can implement one 36x36 multiplier cluster called a `mult_36x36_slice` or it can implement two divisible 18x18 multipliers. A divisible 18x18 multiplier can implement a 18x18 multiplier cluster called a `mult_18x18_slice` or it can be fractured into two 9x9 multiplier clusters called `mult_9x9_slice`. Fig. 9.10 shows a multiplier slice. Pins belonging to the same input or output port of a multiplier slice must be either all registered or none registered. Pins belonging to different ports or different slices may have different register configurations. A multiplier primitive itself has two input ports (A and B) and one output port (OUT).
First, we describe the `block_mult` complex block as follows:

```xml
<pb_type name="block_mult">
  <input name="A" num_pins="36"/>
  <input name="B" num_pins="36"/>
  <output name="OUT" num_pins="72"/>
  <clock name="clk"/>
</pb_type>
```

The `block_mult` complex block has two modes: a mode containing a 36x36 multiplier slice and a mode containing two fracturable 18x18 multipliers. The mode containing the 36x36 multiplier slice is described first. The mode and slice is declared here:

```xml
<mode name="mult_36x36">
  <pb_type name="mult_36x36_slice" num_pb="1">
    <input name="A_cfg" num_pins="36"/>
    <input name="B_cfg" num_pins="36"/>
    <input name="OUT_cfg" num_pins="72"/>
    <clock name="clk"/>
  </pb_type>
</mode>
```

This is followed by a description of the primitives within the slice. There are two sets of 36 flip-flops for the input ports and one set of 72 flip-flops for the output port. There is one 36x36 multiplier primitive. These primitives are described by four `pb_type`s as follows:

```xml
<pb_type name="reg_36x36_A" blif_model=".latch" num_pb="36" class="flipflop">
  <input name="D" num_pins="1" port_class="D"/>
  <output name="Q" num_pins="1" port_class="Q"/>
  <clock name="clk" port_class="clock"/>
</pb_type>
<pb_type name="reg_36x36_B" blif_model=".latch" num_pb="36" class="flipflop">
  <input name="D" num_pins="1" port_class="D"/>
  <output name="Q" num_pins="1" port_class="Q"/>
  <clock name="clk" port_class="clock"/>
</pb_type>
<pb_type name="reg_36x36_out" blif_model=".latch" num_pb="72" class="flipflop">
  <input name="D" num_pins="1" port_class="D"/>
  <output name="Q" num_pins="1" port_class="Q"/>
  <clock name="clk" port_class="clock"/>
</pb_type>
```

```xml
<pb_type name="mult_36x36" blif_model=".subckt mult" num_pb="1">
  <input name="A" num_pins="36"/>
  <input name="B" num_pins="36"/>
  <output name="OUT" num_pins="72"/>
</pb_type>
```

(continues on next page)
The slice description finishes with a specification of the interconnection. Using the same technique as in the memory example, bus-based multiplexers are used to register the ports. Clocks are connected using the complete tag because there is a one-to-many relationship. Direct tags are used to make simple, one-to-one connections.

```
<interconnect>
    <direct input="mult_36x36_slice.A_cfg" output="reg_36x36_A[35:0].D"/>
    <direct input="mult_36x36_slice.B_cfg" output="reg_36x36_B[35:0].D"/>
    <mux input="mult_36x36_slice.A_cfg reg_36x36_A[35:0].Q" output="mult_36x36.A"/>
    <mux input="mult_36x36_slice.B_cfg reg_36x36_B[35:0].Q" output="mult_36x36.B"/>

    <direct input="mult_36x36.OUT" output="reg_36x36_out[71:0].D"/>
    <mux input="mult_36x36.OUT reg_36x36_out[71:0].Q" output="mult_36x36_slice.OUT_cfg"/>

    <complete input="mult_36x36_slice.clk" output="reg_36x36_A[35:0].clk"/>
    <complete input="mult_36x36_slice.clk" output="reg_36x36_B[35:0].clk"/>
    <complete input="mult_36x36_slice.clk" output="reg_36x36_out[71:0].clk"/>
</interconnect>
```

The mode finishes with a specification of the interconnect between the slice and its parent.

```
<interconnect>
    <direct input="block_mult.A" output="mult_36x36_slice.A_cfg"/>
    <direct input="block_mult.B" output="mult_36x36_slice.A_cfg"/>
    <direct input="mult_36x36_slice.OUT_cfg" output="block_mult.OUT"/>
    <direct input="block_mult.clk" output="mult_36x36_slice.clk"/>
</interconnect>
```

After the mode containing the 36x36 multiplier slice is described, the mode containing two fracturable 18x18 multipliers is described:

```
<mode name="two_divisible_mult_18x18">
    <pb_type name="divisible_mult_18x18" num_pb="2">
        <input name="A" num_pins="18"/>
        <input name="B" num_pins="18"/>
        <input name="OUT" num_pins="36"/>
        <clock name="clk"/>
    </pb_type>
    <interconnect>
        <!-- follows previous pattern for slice definition -->
    </interconnect>
</mode>
```

This mode has two additional modes which are the actual 18x18 multiply block or two 9x9 multiplier blocks. Both follow a similar description as the mult_36x36_slice with just the number of pins halved so the details are not repeated.

```
<mode name="two_divisible_mult_18x18">
    <pb_type name="mult_18x18_slice" num_pb="1">
        <!-- follows previous pattern for slice definition -->
    </pb_type>
    <interconnect>
        <!-- follows previous pattern for slice definition -->
    </interconnect>
</mode>
```
The interconnect for the divisible 18x18 mode is shown in Fig. 9.11. The unique characteristic of this interconnect is that the input and output ports of the parent is split in half, one half for each child. A convenient way to specify this is to use the syntax divisible_mult_18x18[1:0] which will append the pins of the ports of the children together. The interconnect for the fracturable 18x18 mode is described here:

Fracturable Multiplier Bus-Based Complete Example

<!-- Example of a fracturable multiplier whose inputs and outputs may be optionally registered
The multiplier hard logic block can implement one 36x36, two 18x18, or four 9x9 multiplies
-->

<pb_type name="block_mult">
  <input name="A" num_pins="36"/>
  <input name="B" num_pins="36"/>
  <output name="OUT" num_pins="72"/>
  <clock name="clk"/>
</pb_type>
<mode name="mult_36x36">
  <pb_type name="mult_36x36_slice" num_pb="1">
    <input name="A_cfg" num_pins="36" equivalence="false"/>
    <input name="B_cfg" num_pins="36" equivalence="false"/>
    <input name="OUT_cfg" num_pins="72" equivalence="false"/>
    <clock name="clk"/>
    
    <pb_type name="reg_36x36_A" blif_model=".latch" num_pb="36" class="flipflop">
      <input name="D" num_pins="1" port_class="D"/>
      <output name="Q" num_pins="1" port_class="Q"/>
      <clock name="clk" port_class="clock"/>
    </pb_type>
    
    <pb_type name="reg_36x36_B" blif_model=".latch" num_pb="36" class="flipflop">
      <input name="D" num_pins="1" port_class="D"/>
      <output name="Q" num_pins="1" port_class="Q"/>
      <clock name="clk" port_class="clock"/>
    </pb_type>
    
    <pb_type name="reg_36x36_out" blif_model=".latch" num_pb="72" class="flipflop">
      <input name="D" num_pins="1" port_class="D"/>
      <output name="Q" num_pins="1" port_class="Q"/>
      <clock name="clk" port_class="clock"/>
    </pb_type>
    
    <pb_type name="mult_36x36" blif_model=".subckt mult" num_pb="1">
      <input name="A" num_pins="36"/>
      <input name="B" num_pins="36"/>
      <output name="OUT" num_pins="72"/>
    </pb_type>
  </pb_type>
  
  <interconnect>
    <direct input="mult_36x36_slice.A_cfg" output="reg_36x36_A[35:0].D"/>
    <direct input="mult_36x36_slice.B_cfg" output="reg_36x36_B[35:0].D"/>
    <mux input="mult_36x36_slice.A_cfg reg_36x36_A[35:0].Q" output="mult_36x36_A"/>
    <mux input="mult_36x36_slice.B_cfg reg_36x36_B[35:0].Q" output="mult_36x36_B"/>
    <direct input="mult_36x36.OUT" output="reg_36x36_out[71:0].D"/>
    <mux input="mult_36x36.OUT reg_36x36_out[71:0].Q" output="mult_36x36_slice.OUT_cfg"/>
    
    <complete input="mult_36x36_slice.clk" output="reg_36x36_A[35:0].clk"/>
    <complete input="mult_36x36_slice.clk" output="reg_36x36_B[35:0].clk"/>
    <complete input="mult_36x36_slice.clk" output="reg_36x36_out[71:0].clk"/>
  </interconnect>
</mode>

9.2. Architecture Modeling
<mode name="two_divisible_mult_18x18">
  <pb_type name="divisible_mult_18x18" num_pb="2">
    <input name="A" num_pins="18"/>
    <input name="B" num_pins="18"/>
    <input name="OUT" num_pins="36"/>
    <clock name="clk"/>
  </pb_type>
</mode>

<mode name="mult_18x18">
  <pb_type name="mult_18x18_slice" num_pb="1">
    <input name="A_cfg" num_pins="18"/>
    <input name="B_cfg" num_pins="18"/>
    <input name="OUT_cfg" num_pins="36"/>
    <clock name="clk"/>
    <pb_type name="reg_18x18_A" blif_model=".latch" num_pb="18" class="flipflop">
      <input name="D" num_pins="1" port_class="D"/>
      <output name="Q" num_pins="1" port_class="Q"/>
      <clock name="clk" port_class="clock"/>
    </pb_type>
    <pb_type name="reg_18x18_B" blif_model=".latch" num_pb="18" class="flipflop">
      <input name="D" num_pins="1" port_class="D"/>
      <output name="Q" num_pins="1" port_class="Q"/>
      <clock name="clk" port_class="clock"/>
    </pb_type>
    <pb_type name="reg_18x18_out" blif_model=".latch" num_pb="36" class="flipflop">
      <input name="D" num_pins="1" port_class="D"/>
      <output name="Q" num_pins="1" port_class="Q"/>
      <clock name="clk" port_class="clock"/>
    </pb_type>
  </pb_type>
</mode>

<interconnect>
  <direct input="mult_18x18_slice.A_cfg" output="reg_18x18_A[17:0].D"/>
  <direct input="mult_18x18_slice.B_cfg" output="reg_18x18_B[17:0].D"/>
  <mux input="mult_18x18_slice.A_cfg reg_18x18_A[17:0].Q" output="mult_18x18.A"/>
  <mux input="mult_18x18_slice.B_cfg reg_18x18_B[17:0].Q" output="mult_18x18.B"/>
  <direct input="mult_18x18.OUT" output="reg_18x18_out[35:0].D"/>
  <mux input="mult_18x18.OUT reg_18x18_out[35:0].Q" output="mult_18x18_slice.OUT_cfg"/>
  <complete input="mult_18x18_slice.clk" output="reg_18x18_A[17:0].clk"/>
  <complete input="mult_18x18_slice.clk" output="reg_18x18_B[17:0].clk"/>
  <complete input="mult_18x18_slice.clk" output="reg_18x18_out[35:0].clk"/>
</interconnect>
<interconnect>
  <direct input="divisible_mult_18x18.A" output="mult_18x18_slice.A_cfg"/>
  <direct input="divisible_mult_18x18.B" output="mult_18x18_slice.A_cfg"/>
  <direct input="mult_18x18_slice.OUT_cfg" output="divisible_mult_18x18.OUT"/>
  <complete input="divisible_mult_18x18.clk" output="mult_18x18_slice.clk"/>
</interconnect>

</mode>

<mode name="two_mult_9x9">
  <pb_type name="mult_9x9_slice" num_pb="2">
    <input name="A_cfg" num_pins="9"/>
    <input name="B_cfg" num_pins="9"/>
    <input name="OUT_cfg" num_pins="18"/>
    <clock name="clk"/>
  </pb_type>

  <pb_type name="reg_9x9_A" blif_model=".latch" num_pb="9" class="flipflop">
    <input name="D" num_pins="1" port_class="D"/>
    <output name="Q" num_pins="1" port_class="Q"/>
    <clock name="clk" port_class="clock"/>
  </pb_type>

  <pb_type name="reg_9x9_B" blif_model=".latch" num_pb="9" class="flipflop">
    <input name="D" num_pins="1" port_class="D"/>
    <output name="Q" num_pins="1" port_class="Q"/>
    <clock name="clk" port_class="clock"/>
  </pb_type>

  <pb_type name="reg_9x9_out" blif_model=".latch" num_pb="18" class="flipflop">
    <input name="D" num_pins="1" port_class="D"/>
    <output name="Q" num_pins="1" port_class="Q"/>
    <clock name="clk" port_class="clock"/>
  </pb_type>

  <pb_type name="mult_9x9" blif_model=".subckt mult" num_pb="1">
    <input name="A" num_pins="9"/>
    <input name="B" num_pins="9"/>
    <output name="OUT" num_pins="18"/>
  </pb_type>

  <interconnect>
    <direct input="mult_9x9_slice.A_cfg" output="reg_9x9_A[8:0].D"/>
    <direct input="mult_9x9_slice.B_cfg" output="reg_9x9_B[8:0].D"/>
    <mux input="mult_9x9_slice.A_cfg reg_9x9_A[8:0].Q" output="mult_9x9.A"/>
    <mux input="mult_9x9_slice.B_cfg reg_9x9_B[8:0].Q" output="mult_9x9.B"/>
    <direct input="mult_9x9.0UT" output="reg_9x9_out[17:0].D"/>
    <mux input="mult_9x9.0UT reg_9x9_out[17:0].Q" output="mult_9x9_slice.OUT_cfg"/>
  </interconnect>

  <complete input="mult_9x9_slice.clk" output="reg_9x9_A[8:0].clk"/>
  <complete input="mult_9x9_slice.clk" output="reg_9x9_B[8:0].clk"/>
  <complete input="mult_9x9_slice.clk" output="reg_9x9_out[17:0].clk"/>
</mode>

(continues on next page)
Architecture Description Examples:

9.2.5 Fracturable Multiplier Example

A 36x36 multiplier fracturable into 18x18s and 9x9s

```xml
<pb_type name="mult_36" height="3">
  <input name="a" num_pins="36"/>
  <input name="b" num_pins="36"/>
  <output name="out" num_pins="72"/>

  <mode name="two_divisible_mult_18x18">
    <pb_type name="divisible_mult_18x18" num_pb="2">
      <input name="a" num_pins="18"/>
      <input name="b" num_pins="18"/>
      <output name="out" num_pins="36"/>
    </pb_type>

    <mode name="two_mult_9x9">
      <pb_type name="mult_9x9_slice" num_pb="2">
        <input name="A_cfg" num_pins="9"/>
        <input name="B_cfg" num_pins="9"/>
      </pb_type>
    </mode>
  </mode>
</pb_type>
```
<output name="OUT_cfg" num_pins="18"/>

<pb_type name="mult_9x9" blif_model=".subckt multiply" num_pb="1" area="300">
  <input name="a" num_pins="9"/>
  <input name="b" num_pins="9"/>
  <output name="out" num_pins="18"/>
  <delay_constant max="2.03e-13" min="1.89e-13" in_port="{a b}" out_port="out"/>
</pb_type>

<interconnect>
  <direct name="a2a" input="mult_9x9_slice.A_cfg" output="mult_9x9.a">
    <delay_constant max="2.03e-13" min="1.89e-13" in_port="mult_9x9_slice.A_cfg" out_port="mult_9x9.a"/>
    <C_constant C="1.89e-13" in_port="mult_9x9_slice.A_cfg" out_port="mult_9x9.a"/>
  </direct>
  <direct name="b2b" input="mult_9x9_slice.B_cfg" output="mult_9x9.b">
    <delay_constant max="2.03e-13" min="1.89e-13" in_port="mult_9x9_slice.B_cfg" out_port="mult_9x9.b"/>
    <C_constant C="1.89e-13" in_port="mult_9x9_slice.B_cfg" out_port="mult_9x9.b"/>
  </direct>
  <direct name="out2out" input="mult_9x9.out" output="mult_9x9_slice.OUT_cfg">
    <delay_constant max="2.03e-13" min="1.89e-13" in_port="mult_9x9.out" out_port="mult_9x9_slice.OUT_cfg"/>
    <C_constant C="1.89e-13" in_port="mult_9x9.out" out_port="mult_9x9_slice.OUT_cfg"/>
  </direct>
</interconnect>

<interconnect>
  <direct name="a2a" input="divisible_mult_18x18.a" output="mult_9x9_slice[1:0].A_cfg">
    <delay_constant max="2.03e-13" min="1.89e-13" in_port="divisible_mult_18x18.a" out_port="mult_9x9_slice[1:0].A_cfg"/>
    <C_constant C="1.89e-13" in_port="divisible_mult_18x18.a" out_port="mult_9x9_slice[1:0].A_cfg"/>
  </direct>
  <direct name="b2b" input="divisible_mult_18x18.b" output="mult_9x9_slice[1:0].B_cfg">
    <delay_constant max="2.03e-13" min="1.89e-13" in_port="divisible_mult_18x18.b" out_port="mult_9x9_slice[1:0].B_cfg"/>
    <C_constant C="1.89e-13" in_port="divisible_mult_18x18.b" out_port="mult_9x9_slice[1:0].B_cfg"/>
  </direct>
  <direct name="out2out" input="mult_9x9_slice[1:0].OUT_cfg" output="divisible_mult_18x18.out">
    <delay_constant max="2.03e-13" min="1.89e-13" in_port="mult_9x9_slice[1:0].OUT_cfg" out_port="divisible_mult_18x18.out"/>
    <C_constant C="1.89e-13" in_port="mult_9x9_slice[1:0].OUT_cfg" out_port="divisible_mult_18x18.out"/>
  </direct>
</interconnect>

(continued on next page)
<mode name="mult_18x18">
  <pb_type name="mult_18x18_slice" num_pb="1">
    <input name="A_cfg" num_pins="18"/>
    <input name="B_cfg" num_pins="18"/>
    <output name="OUT_cfg" num_pins="36"/>
  </pb_type>
  <pb_type name="mult_18x18" blif_model=".subckt multiply" num_pb="1" area="1000">
    <input name="a" num_pins="18"/>
    <input name="b" num_pins="18"/>
    <output name="out" num_pins="36"/>
    <delay_constant max="2.03e-13" min="1.89e-13" in_port="\{a b\}" out_port="out"/>
  </pb_type>
</mode>

<interconnect>
  <direct name="a2a" input="mult_18x18_slice.A_cfg" output="mult_18x18.a">
    <delay_constant max="2.03e-13" min="1.89e-13" in_port="mult_18x18_slice.A_cfg" out_port="mult_18x18.a"/>
    <C_constant C="1.89e-13" in_port="mult_18x18_slice.A_cfg" out_port="mult_18x18.a"/>
  </direct>
  
  <direct name="b2b" input="mult_18x18_slice.B_cfg" output="mult_18x18.b">
    <delay_constant max="2.03e-13" min="1.89e-13" in_port="mult_18x18_slice.B_cfg" out_port="mult_18x18.b"/>
    <C_constant C="1.89e-13" in_port="mult_18x18_slice.B_cfg" out_port="mult_18x18_slice.B_cfg"/>
  </direct>
  
  <direct name="out2out" input="mult_18x18.out" output="mult_18x18_slice.OUT_cfg">
    <delay_constant max="2.03e-13" min="1.89e-13" in_port="mult_18x18.out" out_port="mult_18x18.out"/>
    <C_constant C="1.89e-13" in_port="mult_18x18.out" out_port="mult_18x18_slice.OUT_cfg"/>
  </direct>
</interconnect>

<interconnect>
  <direct name="a2a" input="divisible_mult_18x18.a" output="mult_18x18_slice.A_cfg">
    <delay_constant max="2.03e-13" min="1.89e-13" in_port="divisible_mult_18x18.a" out_port="mult_18x18_slice.A_cfg"/>
    <C_constant C="1.89e-13" in_port="divisible_mult_18x18.a" out_port="mult_18x18_slice.A_cfg"/>
  </direct>
  
  <direct name="b2b" input="divisible_mult_18x18.b" output="mult_18x18_slice.B_cfg">
    <delay_constant max="2.03e-13" min="1.89e-13" in_port="divisible_mult_18x18.b" out_port="mult_18x18_slice.B_cfg"/>
    <C_constant C="1.89e-13" in_port="divisible_mult_18x18.b" out_port="mult_18x18_slice.B_cfg"/>
  </direct>
</interconnect>
Verilog-to-Routing Documentation, Release 8.1.0-dev

(continued from previous page)

```
<interconnect>
  <direct name="a2a" input="mult_36.a" output="divisible_mult_18x18[1:0].a"/>
  <delay_constant max="2.03e-13" min="1.89e-13" in_port="mult_36.a" out_port="divisible_mult_18x18[1:0].a"/>
  <C_constant C="1.89e-13" in_port="mult_36.a" out_port="divisible_mult_18x18[1:0].a"/>
</interconnect>

<interconnect>
  <direct name="b2b" input="mult_36.b" output="divisible_mult_18x18[1:0].a"/>
  <delay_constant max="2.03e-13" min="1.89e-13" in_port="mult_36.b" out_port="divisible_mult_18x18[1:0].a"/>
  <C_constant C="1.89e-13" in_port="mult_36.b" out_port="divisible_mult_18x18[1:0].a"/>
</interconnect>

<interconnect>
  <direct name="out2out" input="divisible_mult_18x18[1:0].out" output="mult_36.out"/>
  <delay_constant max="2.03e-13" min="1.89e-13" in_port="divisible_mult_18x18[1:0].out" out_port="mult_36.out"/>
  <C_constant C="1.89e-13" in_port="divisible_mult_18x18[1:0].out" out_port="mult_36.out"/>
</interconnect>
```

(continues on next page)
&lt;C_constant C="1.89e-13" in_port="mult_36x36_slice.A_cfg" out_port="mult_36x36.a"/&gt;

&lt;/direct&gt;
&lt;direct name="b2b" input="mult_36x36_slice.B_cfg" output="mult_36x36.b"&gt;
 &lt;delay_constant max="2.03e-13" min="1.89e-13" in_port="mult_36x36_slice.B_cfg" out_port="mult_36x36.b"/&gt;
 &lt;C_constant C="1.89e-13" in_port="mult_36x36_slice.B_cfg" out_port="mult_36x36.b"/&gt;
 &lt;/direct&gt;
&lt;direct name="out2out" input="mult_36x36.out" output="mult_36x36_slice.OUT_cfg"&gt;
 &lt;delay_constant max="2.03e-13" min="1.89e-13" in_port="mult_36x36.out" out_port="mult_36x36_slice.OUT_cfg"/&gt;
 &lt;C_constant C="1.89e-13" in_port="mult_36x36.out" out_port="mult_36x36_slice.OUT_cfg"/&gt;
 &lt;/direct&gt;
 &lt;/interconnect&gt;
 &lt;/pb_type&gt;

&lt;interconnect&gt;
 &lt;direct name="a2a" input="mult_36.a" output="mult_36x36_slice.A_cfg"&gt;
 &lt;delay_constant max="2.03e-13" min="1.89e-13" in_port="mult_36.a" out_port="mult_36x36_slice.A_cfg"/&gt;
 &lt;C_constant C="1.89e-13" in_port="mult_36.a" out_port="mult_36x36_slice.A_cfg"/&gt;
 &lt;/direct&gt;
 &lt;/interconnect&gt;

&lt;direct name="b2b" input="mult_36.b" output="mult_36x36_slice.B_cfg"&gt;
 &lt;delay_constant max="2.03e-13" min="1.89e-13" in_port="mult_36.b" out_port="mult_36x36_slice.B_cfg"/&gt;
 &lt;C_constant C="1.89e-13" in_port="mult_36.b" out_port="mult_36x36_slice.B_cfg"/&gt;
 &lt;/direct&gt;
&lt;/interconnect&gt;
&lt;/mode&gt;

&lt;fc_in type="frac"> 0.15&lt;/fc_in&gt;
&lt;fc_out type="frac"> 0.125&lt;/fc_out&gt;

&lt;pinlocations pattern="spread"/&gt;
&lt;/pb_type&gt;
9.2.6 Configurable Memory Block Example

A memory block with a reconfigurable aspect ratio.

```xml
<pb_type name="memory" height="1">
  <input name="addr1" num_pins="14"/>
  <input name="addr2" num_pins="14"/>
  <input name="data" num_pins="16"/>
  <input name="we1" num_pins="1"/>
  <input name="we2" num_pins="1"/>
  <output name="out" num_pins="16"/>
  <clock name="clk" num_pins="1"/>
</pb_type>

<mode name="mem_1024x16_sp">
  <pb_type name="mem_1024x16_sp" blif_model=".subckt single_port_ram" class="memory">
    <input name="addr" num_pins="10" port_class="address"/>
    <input name="data" num_pins="16" port_class="data_in"/>
    <input name="we" num_pins="1" port_class="write_en"/>
    <output name="out" num_pins="16" port_class="data_out"/>
    <clock name="clk" num_pins="1" port_class="clock"/>
  </pb_type>
  <interconnect>
    <direct name="address1" input="memory.addr1[9:0]" output="mem_1024x16_sp.addr"/>
    <direct name="data1" input="memory.data[15:0]" output="mem_1024x16_sp.data"/>
    <direct name="writeen1" input="memory.we1" output="mem_1024x16_sp.we"/>
    <direct name="dataout1" input="mem_1024x16_sp.out" output="memory.out[15:0]"/>
    <direct name="clk" input="memory.clk" output="mem_1024x16_sp.clk"/>
  </interconnect>
</mode>

<mode name="mem_2048x8_dp">
  <pb_type name="mem_2048x8_dp" blif_model=".subckt dual_port_ram" class="memory">
    <input name="addr1" num_pins="11" port_class="address1"/>
    <input name="addr2" num_pins="11" port_class="address2"/>
    <input name="data1" num_pins="8" port_class="data_in1"/>
    <input name="data2" num_pins="8" port_class="data_in2"/>
    <input name="we1" num_pins="1" port_class="write_en1"/>
    <input name="we2" num_pins="1" port_class="write_en2"/>
    <output name="out1" num_pins="8" port_class="data_out1"/>
    <output name="out2" num_pins="8" port_class="data_out2"/>
    <clock name="clk" num_pins="1" port_class="clock"/>
  </pb_type>
  <interconnect>
    <direct name="address1" input="memory.addr1[10:0]" output="mem_2048x8_dp.addr1"/>
    <direct name="address2" input="memory.addr2[10:0]" output="mem_2048x8_dp.addr2"/>
  </interconnect>
</mode>
```

(continues on next page)
<direct name="data1" input="memory.data[7:0]" output="mem_2048x8_dp.data1">
</direct>
<direct name="data2" input="memory.data[15:8]" output="mem_2048x8_dp.data2">
</direct>
<direct name="writeen1" input="memory.we1" output="mem_2048x8_dp.we1">
</direct>
<direct name="writeen2" input="memory.we2" output="mem_2048x8_dp.we2">
</direct>
<direct name="dataout1" input="mem_2048x8_dp.out1" output="memory.out[7:0]">
</direct>
<direct name="dataout2" input="mem_2048x8_dp.out2" output="memory.out[15:8]">
</direct>
<direct name="clk" input="memory.clk" output="mem_2048x8_dp.clk">
</direct>
</interconnect>

</mode>

<mode name="mem_2048x8_sp">
<pb_type name="mem_2048x8_sp" blif_model=".subckt single_port_ram" class="memory">
  <input name="addr" num_pins="11" port_class="address"/>
  <input name="data" num_pins="8" port_class="data_in"/>
  <input name="we" num_pins="1" port_class="write_en"/>
  <output name="out" num_pins="8" port_class="data_out"/>
  <clock name="clk" num_pins="1" port_class="clock"/>
</pb_type>
<interconnect>
<direct name="address1" input="memory.addr1[10:0]" output="mem_2048x8_sp.addr">
</direct>
<direct name="data1" input="memory.data[7:0]" output="mem_2048x8_sp.data">
</direct>
<direct name="writeen1" input="memory.we1" output="mem_2048x8_sp.we">
</direct>
<direct name="dataout1" input="mem_2048x8_sp.out1" output="memory.out[7:0]">
</direct>
<direct name="clk" input="memory.clk" output="mem_2048x8_sp.clk">
</direct>
</interconnect>
</mode>

<mode name="mem_4096x4_dp">
<pb_type name="mem_4096x4_dp" blif_model=".subckt dual_port_ram" class="memory">
  <input name="addr1" num_pins="12" port_class="address1"/>
  <input name="addr2" num_pins="12" port_class="address2"/>
  <input name="data1" num_pins="4" port_class="data_in1"/>
  <input name="data2" num_pins="4" port_class="data_in2"/>
  <input name="we1" num_pins="1" port_class="write_en1"/>
  <input name="we2" num_pins="1" port_class="write_en2"/>
  <output name="out1" num_pins="4" port_class="data_out1"/>
  <output name="out2" num_pins="4" port_class="data_out2"/>
  <clock name="clk" num_pins="1" port_class="clock"/>
</pb_type>
</mode>
<interconnect>
  <direct name="address1" input="memory.addr1[11:0]" output="mem_4096x4_dp.addr1">
  </direct>
  <direct name="address2" input="memory.addr2[11:0]" output="mem_4096x4_dp.addr2">
  </direct>
  <direct name="data1" input="memory.data[3:0]" output="mem_4096x4_dp.data1">
  </direct>
  <direct name="data2" input="memory.data[7:4]" output="mem_4096x4_dp.data2">
  </direct>
  <direct name="writeen1" input="memory.we1" output="mem_4096x4_dp.we1">
  </direct>
  <direct name="writeen2" input="memory.we2" output="mem_4096x4_dp.we2">
  </direct>
  <direct name="dataout1" input="mem_4096x4_dp.out1" output="memory.out[3:0]">
  </direct>
  <direct name="dataout2" input="mem_4096x4_dp.out2" output="memory.out[7:4]">
  </direct>
  <direct name="clk" input="memory.clk" output="mem_4096x4_dp.clk">
  </direct>
</interconnect>

</mode>

<mode name="mem_4096x4_sp">
  <pb_type name="mem_4096x4_sp" blif_model="subckt single_port_ram" class="memory" num_pb="1" area="1000">
    <input name="addr" num_pins="12" port_class="address" />
    <input name="data" num_pins="4" port_class="data_in" />
    <input name="we" num_pins="1" port_class="write_en" />
    <output name="out" num_pins="4" port_class="data_out" />
    <clock name="clk" num_pins="1" port_class="clock" />
  </pb_type>
  <interconnect>
    <direct name="address1" input="memory.addr1[11:0]" output="mem_4096x4_sp.addr1">
    </direct>
    <direct name="data1" input="memory.data[3:0]" output="mem_4096x4_sp.data1">
    </direct>
    <direct name="writeen1" input="memory.we1" output="mem_4096x4_sp.we1">
    </direct>
    <direct name="dataout1" input="mem_4096x4_sp.out1" output="memory.out[3:0]">
    </direct>
    <direct name="clk" input="memory.clk" output="mem_4096x4_sp.clk">
    </direct>
  </interconnect>
</mode>

<mode name="mem_8192x2_dp">
  <pb_type name="mem_8192x2_dp" blif_model="subckt dual_port_ram" class="memory" num_pb="1" area="1000">
    <input name="addr1" num_pins="13" port_class="address1" />
    <input name="addr2" num_pins="13" port_class="address2" />
    <input name="data1" num_pins="2" port_class="data_in1" />
    <input name="data2" num_pins="2" port_class="data_in2" />
    <input name="we1" num_pins="1" port_class="write_en1" />
  </pb_type>
  <interconnect>
    <direct name="address1" input="memory.addr1[11:0]" output="mem_8192x2_dp.addr1">
    </direct>
    <direct name="data1" input="memory.data[3:0]" output="mem_8192x2_dp.data1">
    </direct>
    <direct name="writeen1" input="memory.we1" output="mem_8192x2_dp.we1">
    </direct>
    <direct name="dataout1" input="mem_8192x2_dp.out1" output="memory.out[3:0]">
    </direct>
    <direct name="clk" input="memory.clk" output="mem_8192x2_dp.clk">
    </direct>
  </interconnect>
</mode>
<input name="we2" num_pins="1" port_class="write_en2"/>
<output name="out1" num_pins="2" port_class="data_out1"/>
<output name="out2" num_pins="2" port_class="data_out2"/>
<clock name="clk" num_pins="1" port_class="clock"/>
</pb_type>

<interconnect>
  <direct name="address1" input="memory.addr1[12:0]" output="mem_8192x2_dp.addr1">
  </direct>
  <direct name="address2" input="memory.addr2[12:0]" output="mem_8192x2_dp.addr2">
  </direct>
  <direct name="data1" input="memory.data[1:0]" output="mem_8192x2_dp.data1">
  </direct>
  <direct name="data2" input="memory.data[3:2]" output="mem_8192x2_dp.data2">
  </direct>
  <direct name="writeen1" input="memory.we1" output="mem_8192x2_dp.we1">
  </direct>
  <direct name="writeen2" input="memory.we2" output="mem_8192x2_dp.we2">
  </direct>
  <direct name="dataout1" input="mem_8192x2_dp.out1" output="memory.out[1:0]">
  </direct>
  <direct name="dataout2" input="mem_8192x2_dp.out2" output="memory.out[3:2]">
  </direct>
  <direct name="clk" input="memory.clk" output="mem_8192x2_dp.clk">
  </direct>
</interconnect>

<mode name="mem_8192x2_sp">
  <pb_type name="mem_8192x2_sp" blif_model=".subckt single_port_ram" class="memory" num_pb="1" area="1000">
    <input name="addr" num_pins="13" port_class="address"/>
    <input name="data" num_pins="2" port_class="data_in"/>
    <input name="we" num_pins="1" port_class="write_en"/>
    <output name="out" num_pins="2" port_class="data_out"/>
    <clock name="clk" num_pins="1" port_class="clock"/>
  </pb_type>

  <interconnect>
    <direct name="address1" input="memory.addr1[12:0]" output="mem_8192x2_sp.addr"/>
    <direct name="data1" input="memory.data[1:0]" output="mem_8192x2_sp.data"/>
    <direct name="writeen1" input="memory.we1" output="mem_8192x2_sp.we"/>
    <direct name="dataout1" input="mem_8192x2_sp.out" output="memory.out[1:0]"/>
    <direct name="clk" input="memory.clk" output="mem_8192x2_sp.clk"/>
  </interconnect>
</mode>

<mode name="mem_16384x1_dp">
  <pb_type name="mem_16384x1_dp" blif_model=".subckt dual_port_ram" class="memory" num_pb="1" area="1000">
    <input name="addr" num_pins="13" port_class="address"/>
    <input name="data" num_pins="2" port_class="data_in"/>
    <input name="we" num_pins="1" port_class="write_en"/>
    <output name="out" num_pins="2" port_class="data_out"/>
    <clock name="clk" num_pins="1" port_class="clock"/>
  </pb_type>

  <interconnect>
    <direct name="address1" input="memory.addr1[12:0]" output="mem_16384x1_dp.addr"/>
    <direct name="data1" input="memory.data[1:0]" output="mem_16384x1_dp.data"/>
    <direct name="writeen1" input="memory.we1" output="mem_16384x1_dp.we"/>
    <direct name="dataout1" input="mem_16384x1_dp.out" output="memory.out[1:0]"/>
    <direct name="clk" input="memory.clk" output="mem_16384x1_dp.clk"/>
  </interconnect>
</mode>

(continues on next page)
9.2.7 Virtex 6 like Logic Slice Example

In order to demonstrate the expressiveness of the architecture description language, we use it to describe a section of a commercial logic block. In this example, we describe the Xilinx Virtex-6 FPGA logic slice [Xilinx Inc12], shown in Fig. 9.12, as follows:

![Commercial FPGA logic block slice (Xilinx Virtex-6)](image)

```xml
<pb_type name="v6_lslice">
    <input name="AX" num_pins="1" equivalent="false"/>
    <input name="A" num_pins="5" equivalent="false"/>
    <input name="AI" num_pins="1" equivalent="false"/>
    <input name="BX" num_pins="1" equivalent="false"/>
    <input name="B" num_pins="5" equivalent="false"/>
    <input name="BI" num_pins="1" equivalent="false"/>
    <input name="CX" num_pins="1" equivalent="false"/>
    <input name="C" num_pins="5" equivalent="false"/>
    <input name="CI" num_pins="1" equivalent="false"/>
    <input name="DX" num_pins="1" equivalent="false"/>
    <input name="D" num_pins="5" equivalent="false"/>
    <input name="DI" num_pins="1" equivalent="false"/>
    <input name="SR" num_pins="1" equivalent="false"/>
    <input name="CIN" num_pins="1" equivalent="false"/>
    <input name="CE" num_pins="1" equivalent="false"/>
    <output name="AMUX" num_pins="1" equivalent="false"/>
    <output name="Aout" num_pins="1" equivalent="false"/>
    <output name="AQ" num_pins="1" equivalent="false"/>
    <output name="BMUX" num_pins="1" equivalent="false"/>
    <output name="Bout" num_pins="1" equivalent="false"/>
    <output name="BQ" num_pins="1" equivalent="false"/>
    <output name="CMUX" num_pins="1" equivalent="false"/>
</pb_type>
```

(continues on next page)
<output name="Cout" num_pins="1" equivalent="false"/>
<output name="CQ" num_pins="1" equivalent="false"/>
<output name="DMUX" num_pins="1" equivalent="false"/>
<output name="Dout" num_pins="1" equivalent="false"/>
<output name="DQ" num_pins="1" equivalent="false"/>
<output name="COUT" num_pins="1" equivalent="false"/>

<clock name="CLK"/>

<!-- For the purposes of this example, the Virtex-6 fracturable LUT will be specified as a primitive. If the architect wishes to explore the Xilinx Virtex-6 further, add more detail into this pb_type. Similar convention for flip-flops -->
<pb_type name="fraclut" num_pb="4" blif_model=".subckt vfraclut">
  <input name="A" num_pins="5"/>
  <input name="W" num_pins="5"/>
  <input name="DI1" num_pins="1"/>
  <input name="DI2" num_pins="1"/>
  <output name="MC31" num_pins="1"/>
  <output name="O6" num_pins="1"/>
  <output name="O5" num_pins="1"/>
</pb_type>

<pb_type name="carry" num_pb="4" blif_model=".subckt carry">
  <!-- This is actually the carry-chain but we don't have a special way to specify chain logic yet in UTFAL so it needs to be specified as regular gate logic, the xor gate and the two muxes to the left of it that are shaded grey comprise the logic gates representing the carry logic -->
  <input name="xor" num_pins="1"/>
  <input name="cmuxxor" num_pins="1"/>
  <input name="cmux" num_pins="1"/>
  <input name="cmux_select" num_pins="1"/>
  <input name="mmux" num_pins="2"/>
  <input name="mmux_select" num_pins="1"/>
  <output name="xor_out" num_pins="1"/>
  <output name="cmux_out" num_pins="1"/>
  <output name="mmux_out" num_pins="1"/>
</pb_type>

<pb_type name="ff_small" num_pb="4" blif_model=".subckt vffs">
  <input name="D" num_pins="1"/>
  <input name="CE" num_pins="1"/>
  <input name="SR" num_pins="1"/>
  <output name="Q" num_pins="1"/>
  <clock name="CK" num_pins="1"/>
</pb_type>

<pb_type name="ff_big" num_pb="4" blif_model=".subckt vffb">
  <input name="D" num_pins="1"/>
  <input name="CE" num_pins="1"/>
  <input name="SR" num_pins="1"/>
</pb_type>

(continues on next page)
<output name="Q" num_pins="1"/>
<clock name="CK" num_pins="1"/>
</pb_type>

<interconnect>
  <direct name="fraclutDI2" input="{v6_lslice.AX v6_lslice.BX v6_lslice.CX v6_lslice.DX}" output="fraclut.DI2"/>
  <direct name="DfraclutDI1" input="v6_lslice.DI" output="fraclut[3].DI1"/>
  <direct name="carry06" input="fraclut.06" output="carry.xor"/>
  <direct name="carrymuxxor" input="carry[2:0].cmux_out" output="carry[3:1].cmuxxor"/>
  <direct name="carrymux" input="{fraclut[3].06 fraclut[2].06 fraclut[2].06 fraclut[1].06 fraclut[0].06}" output="carry[2:0].mmux"/>
  <direct name="carrymux_select" input="{v6_lslice.AX v6_lslice.BX v6_lslice.CX}" output="carry[2:0].mmux_select"/>
  <direct name="cout" input="carry[3].mmux_out" output="v6_lslice.COUT"/>
  <direct name="Q" input="ff_big.Q" output="{DQ CQ BQ AQ}"/>
  <mux name="ff_smallA" input="v6_lslice.AX fraclut[0].05" output="ff_small[0].D"/>
  <mux name="ff_smallB" input="v6_lslice.BX fraclut[1].05" output="ff_small[1].D"/>
  <mux name="ff_smallC" input="v6_lslice.CX fraclut[2].05" output="ff_small[2].D"/>
  <mux name="ff_smallD" input="v6_lslice.DX fraclut[3].05" output="ff_small[3].D"/>
  <mux name="ff_bigA" input="fraclut[0].05 fraclut[0].06 carry[0].cmux_out carry[0].xor_out" output="ff_big[0].D"/>
  <mux name="ff_bigB" input="fraclut[1].05 fraclut[1].06 carry[1].cmux_out carry[1].xor_out" output="ff_big[1].D"/>
  <mux name="AMUX" input="fraclut[0].05 fraclut[0].06 carry[0].cmux_out carry[0].mmux_out carry[0].xor_out" output="AMUX"/>
  <mux name="BMUX" input="fraclut[1].05 fraclut[1].06 carry[1].cmux_out carry[1].mmux_out carry[1].xor_out" output="BMUX"/>
  <mux name="CfraclutDI1" input="v6_lslice.CI v6_lslice.DI fraclut[3].MC31" output="fraclut[2].DI1"/>
  <mux name="BfraclutDI1" input="v6_lslice.BI v6_lslice.DI fraclut[2].MC31" output="fraclut[1].DI1"/>
9.2.8 Equivalent Sites tutorial

This tutorial aims at providing information to the user on how to model the equivalent sites to enable equivalent placement in VPR.

Equivalent site placement allows the user to define complex logical blocks (top-level pb_types) that can be used in multiple physical location types of the FPGA device grid. In the same way, the user can define many physical tiles that have different physical attributes that can implement the same logical block.

The first case (multiple physical grid location types for one complex logical block) is explained below. The device has at disposal two different Configurable Logic Blocks (CLB), SLICEL and SLICEM. In this case, the SLICEM CLB is a superset that implements additional features w.r.t. the SLICEL CLB. Therefore, the user can decide to model the architecture to be able to place the SLICEL Complex Block in a SLICEM physical tile, being it a valid grid location. This behavior can lead to the generation of more accurate and better placement results, given that a Complex Logic Block is not bound to only one physical location type.

Below the user can find the implementation of this situation starting from an example that does not make use of the equivalent site placement:

```
<tiles>
  <tile name="SLICEL_TILE">
    <input name="IN_A" num_pins="6"/>
    <input name="AX" num_pins="1"/>
    <input name="SR" num_pins="1"/>
    <input name="CE" num_pins="1"/>
    <input name="CIN" num_pins="1"/>
    <clock name="CLK" num_pins="1"/>
    <output name="A" num_pins="1"/>
    <output name="AMUX" num_pins="1"/>
    <output name="AQ" num_pins="1"/>

    <equivalent_sites>
      <site pb_type="SLICEL_SITE" pin_mapping="direct"/>
    </equivalent_sites>
  </tile>
</tiles>
```
As the user can see, SLICEL and SLICEM are treated as two different entities, even though they seem to be similar one to another. To have the possibility to make VPR choose a SLICEM location when placing a SLICEL_SITE pb_type, the user needs to change the SLICEM tile accordingly, as shown below:

<tile name="SLICEM_TILE">
  <input name="IN_A" num_pins="6"/>
  <input name="AX" num_pins="1"/>
  <input name="AI" num_pins="1"/>
  <input name="SR" num_pins="1"/>
  <input name="WE" num_pins="1"/>
  <input name="CE" num_pins="1"/>
  <input name="CIN" num_pins="1"/>
  <clock name="CLK" num_pins="1"/>
  <output name="A" num_pins="1"/>
  <output name="AMUX" num_pins="1"/>
  <output name="AQ" num_pins="1"/>

  <equivalent_sites>
    <site pb_type="SLICEM_SITE" pin_mapping="direct">
      <direct from="SLICEM_TILE.IN_A" to="SLICEL_SITE.IN_A"/>
      <direct from="SLICEM_TILE.AX" to="SLICEL_SITE.AX"/>
      <direct from="SLICEM_TILE.SR" to="SLICEL_SITE.SR"/>
      <direct from="SLICEM_TILE.CE" to="SLICEL_SITE.CE"/>
      <direct from="SLICEM_TILE.CIN" to="SLICEL_SITE.CIN"/>
      <direct from="SLICEM_TILE.CLK" to="SLICEL_SITE.CLK"/>
      <direct from="SLICEM_TILE.A" to="SLICEL_SITE.A"/>
      <direct from="SLICEM_TILE.AMUX" to="SLICEL_SITE.AMUX"/>
      <direct from="SLICEM_TILE.AQ" to="SLICEL_SITE.AQ"/>
    </site>
    <site pb_type="SLICEL_SITE" pin_mapping="custom">
      <!-- Custom pin mapping definitions here -->
    </site>
  </equivalent_sites>
</tile>

With the above description of the SLICEM tile, the user can now have the SLICEL sites to be placed in SLICEM physical locations. One thing to notice is that not all the pins have been mapped for the SLICEL_SITE. For instance, the WE and AI port are absent from the SLICEL_SITE definition, hence they cannot appear in the pin mapping between physical tile and logical block.

The second case described in this tutorial refers to the situation for which there are multiple different physical location types in the device grid that are used by one complex logical blocks. Imagine the situation for which the device has left and right I/O tile types which have different pinlocations, hence they need to be defined in two different ways. With equivalent site placement, the user doesn’t need to define multiple different pb_types that implement the same functionality.

Below the user can find the implementation of this situation starting from an example that does not make use of the equivalent site placement:
To avoid duplicating the complex logic blocks in LEFT and RIGHT IOPADS, the user can describe the pb_type only once and add it to the equivalent sites tag of the two different tiles, as follows:

```xml
<tiles>
  <tile name="LEFT_IOPAD_TILE">
    <input name="INPUT" num_pins="1"/>
    <output name="OUTPUT" num_pins="1"/>

    <equivalent_sites>
      <site pb_type="LEFT_IOPAD_SITE" pin_mapping="direct"/>
    </equivalent_sites>

    <fc />
    <pinlocations pattern="custom">
      <loc side="left">LEFT_IOPAD_TILE.INPUT</loc>
      <loc side="right">LEFT_IOPAD_TILE.OUTPUT</loc>
    </pinlocations>
  </tile>

  <tile name="RIGHT_IOPAD_TILE">
    <input name="INPUT" num_pins="1"/>
    <output name="OUTPUT" num_pins="1"/>

    <equivalent_sites>
      <site pb_type="RIGHT_IOPAD_SITE" pin_mapping="direct"/>
    </equivalent_sites>

    <fc />
    <pinlocations pattern="custom">
      <loc side="right">RIGHT_IOPAD_TILE.INPUT</loc>
      <loc side="left">RIGHT_IOPAD_TILE.OUTPUT</loc>
    </pinlocations>
  </tile>
</tiles>

<complexblocklist>
  <pb_type name="LEFT_IOPAD_SITE">
    <input name="INPUT" num_pins="1"/>
    <output name="OUTPUT" num_pins="1"/>
    <mode />
  </pb_type>

  <pb_type name="RIGHT_IOPAD_SITE">
    <input name="INPUT" num_pins="1"/>
    <output name="OUTPUT" num_pins="1"/>
    <mode />
  </pb_type>
</complexblocklist>
```
With this implementation, the IOPAD_SITE can be placed both in the LEFT and RIGHT physical location types. Note that the pin_mapping is set as direct, given that the physical tile and the logical block share the same IO pins.

The two different cases can be mixed to have a N to M mapping of physical tiles/logical blocks.

### 9.2.9 Heterogeneous tiles tutorial

This tutorial aims at providing information to the user on how to model sub tiles to enable heterogeneous tiles in VPR.

An heterogeneous tile is a tile that includes two or more site types that may differ in the following aspects:

- **Block types** (pb_type)
- **Fc definition**
- **Pin locations** definition
- **IO ports** definition
As a result, an *heterogeneous tile* has the possibility of having multiple block types at the same \((x, y)\) location in the grid. This comes with the introduction of a third spatial coordinate (sub-block) that identifies the placement of the block type within the \(x\) and \(y\) grid coordinate.

Moreover, the placer can choose and assign different locations for each block type within the same coordinates as well.

Fig. 9.13: Device grid, with \((x, y, \text{sub-block})\) coordinates. Each block can be moved by the placer in all the three spatial dimensions.

To correctly model an architecture, each *Physical Tiles* requires at least one sub tile definition. This represents a default homogeneous architecture, composed of one or many instances of the sub tile within the physical tile (the number of such sub-tiles is referred to as the *capacity*).

To enhance the expressivity of VPR architecture, additional sub tiles can be inserted alongside with the default sub tile. This enables the definition of the *heterogeneous tiles*.

With this new capability, the device grid of a given architecture does include a new sub-block coordinate that identifies the type of sub tile used and its actual location, in case the capacity is greater than 1.

**Heterogeneous tiles examples**

Following, there are two examples to illustrate some potential use cases of the *heterogeneous tiles*, that might be of interest to the reader.

*Note:* The examples below are a simplified versions of the real architectural specification.
Sub-tiles with different pin locations

The Xilinx Series 7 Clock tile is composed of 16 BUFGCTRL sites (pg. 36 of the 7 Series FPGAs Clocking Resources guide). Even though they are equivalent regarding the ports and Fc definition, some of the sites differ in terms of pin locations, as depicted by the simplified representation of the Clock tile in Fig. 9.14.

Heterogeneous tiles come in hand to model this kind of tiles and an example is the following:

```xml
<tiles>
  <tile name="BUFG_TILE">
    <sub_tile name="BUFG_SUB_TILE_0" capacity="1">
      <clock name="I0" num_pins="1"/>
      <clock name="I1" num_pins="1"/>
      <input name="CE0" num_pins="1"/>
      <input name="CE1" num_pins="1"/>
      <input name="IGNORE0" num_pins="1"/>
      <input name="IGNORE1" num_pins="1"/>
      <input name="S0" num_pins="1"/>
      <input name="S1" num_pins="1"/>
      <output name="O" num_pins="1"/>
      <fc in_type="abs" in_val="2" out_type="abs" out_val="2"/>
      <pinlocations pattern="custom">
        <loc side="top">BUFG_SUB_TILE_0.I1 BUFG_SUB_TILE_0.I0 BUFG_SUB_TILE_0.
        CE0 BUFG_SUB_TILE_0.S0 BUFG_SUB_TILE_0.IGNORE1 BUFG_SUB_TILE_0.CE1 BUFG_SUB_TILE_0.
        IGNORE0 BUFG_SUB_TILE_0.S1</loc>
      </pinlocations>
      <equivalent_sites>
        <site pb_type="BUFGCTRL" pin_mapping="direct"/>
      </equivalent_sites>
    </sub_tile>
    <sub_tile name="BUFG_SUB_TILE_1" capacity="14">
      <clock name="I0" num_pins="1"/>
      <clock name="I1" num_pins="1"/>
      <input name="CE0" num_pins="1"/>
      <input name="CE1" num_pins="1"/>
      <input name="IGNORE0" num_pins="1"/>
      <input name="IGNORE1" num_pins="1"/>
      <input name="S0" num_pins="1"/>
      <input name="S1" num_pins="1"/>
      <output name="O" num_pins="1"/>
      <fc in_type="abs" in_val="2" out_type="abs" out_val="2"/>
      <pinlocations pattern="custom">
        <loc side="top">BUFG_SUB_TILE_1.S1 BUFG_SUB_TILE_1.I0 BUFG_SUB_TILE_1.
        CE1 BUFG_SUB_TILE_1.I1 BUFG_SUB_TILE_1.IGNORE1 BUFG_SUB_TILE_1.IGNORE0 BUFG_SUB_TILE_1.
        CE0 BUFG_SUB_TILE_1.S0</loc>
      </pinlocations>
      <equivalent_sites>
        <site pb_type="BUFGCTRL" pin_mapping="direct"/>
      </equivalent_sites>
    </sub_tile>
  </tile>
</tiles>

(continues on next page)
Fig. 9.14: Simplified view of the Clock tile of the Xilinx Series 7 fabric.
The above `BUFG_TILE` contains three types of sub-tiles (`BUFG_SUB_TILE_0`, `BUFG_SUB_TILE_1` and `BUFG_SUB_TILE_2`).

While each sub-tile type contains the same `pb_type` (equivalent_sites of `BUFGCTRL`), they differ in two ways:

1. Each sub-tile has different pin locations. For example `BUFG_SUB_TILE_0` has the `I1` pins on the top side of the tile, while `BUFG_SUB_TILE_1` and `BUFG_SUB_TILE_2` have them on the right and left sides respectively.

2. Each sub-tile has a different ‘capacity’ (i.e. a different number of sites). `BUFG_SUB_TILE_1` and `BUFG_SUB_TILE_2` have capacity 1, while `BUFG_SUB_TILE_1` has capacity 14. As a result the `BUFG_TILE` can implement a total of 16 `BUFGCTRL` blocks.
Sub-tiles containing different block types

As another example taken from the Xilinx Series 7 fabric, the HCLK_IOI tile is composed of three different block types, namely BUFIO, BUFR and IDELAYCTRL.

**HCLK_IOI3 TILE**

![HCLK_IOI tile diagram]

Fig. 9.15: Simplified view of the HCLK_IOI tile in the Xilinx Series 7 fabric.

The reader might think that it is possible to model this situation using the *Complex Blocks* to model this situation, with a `<pb_type>` containing the various blocks.

Indeed, this could be done, but, for some architectures, the placement location of a sub block is particularly relevant, hence the need of leaving this choice to the placement algorithm instead of the packer one.

Each one of these site types has different IO pins as well as pin locations.

```xml
<tile name="HCLK_IOI">
  <sub_tile name="BUFIO" capacity="4">
    <clock name="I" num_pins="1"/>
    <output name="O" num_pins = "1"/>
    <equivalent_sites>
      <site pb_type="BUFIO_SITE" pin_mapping="direct"/>
    </equivalent_sites>
    <fc />
    <pinlocations />
  </sub_tile>
  <sub_tile name="BUFR" capacity="4">
    <clock name="I" num_pins="1"/>
    <input name="CE" num_pins="1"/>
    <output name="O" num_pins = "1"/>
    <equivalent_sites>
      <site pb_type="BUFR_SITE" pin_mapping="direct"/>
    </equivalent_sites>
    <fc />
    <pinlocations />
  </sub_tile>
  <sub_tile name="IDELAYCTRL" capacity="1">
    <clock name="REFCLK" num_pins="1"/>
    <output name="RDY" num_pins="1"/>
  </sub_tile>
</tile>
```

(continues on next page)
Each HCLK_IOI tile contains three sub-tiles, each containing a different type of pb_type:

- the BUFIO sub-tile supports 4 instances (capacity = 4) of pb_type BUFIO_SITE
- the BUFR sub-tile supports 4 instances of BUFR_SITE pb_types
- the IDELAYCTRL sub-tile supports 1 instances of the IDELAYCTRL_SITE

**Modeling Guides:**

### 9.2.10 Primitive Block Timing Modeling Tutorial

To accurately model an FPGA, the architect needs to specify the timing characteristics of the FPGA’s primitives blocks. This involves two key steps:

1. Specifying the logical timing characteristics of a primitive including:
   - whether primitive pins are sequential or combinational, and
   - what the timing dependencies are between the pins.

2. Specifying the physical delay values

These two steps separate the logical timing characteristics of a primitive, from the physically dependant delays. This enables a single logical netlist primitive type (e.g. Flip-Flop) to be mapped into different physical locations with different timing characteristics.

The *FPGA architecture description* describes the logical timing characteristics in the *models section*, while the physical timing information is specified on *pb_types* within *complex block*.

The following sections illustrate some common block timing modeling approaches.

**Combinational block**

A typical combinational block is a full adder, where a, b and cin are combinational inputs, and sum and cout are combinational outputs.

We can model these timing dependencies on the model with the *combinational_sink_ports*, which specifies the output ports which are dependant on an input port:
The physical timing delays are specified on any pb_type instances of the adder model. For example:

```xml
<pb_type name="adder" blif_model=".subckt adder" num_pb="1">
    <input name="a" num_pins="1"/>
    <input name="b" num_pins="1"/>
    <input name="cin" num_pins="1"/>
    <output name="cout" num_pins="1"/>
    <output name="sum" num_pins="1"/>
    <delay_constant max="300e-12" in_port="adder.a" out_port="adder.sum"/>
    <delay_constant max="300e-12" in_port="adder.b" out_port="adder.sum"/>
    <delay_constant max="300e-12" in_port="adder.cin" out_port="adder.sum"/>
    <delay_constant max="300e-12" in_port="adder.a" out_port="adder.cout"/>
    <delay_constant max="300e-12" in_port="adder.b" out_port="adder.cout"/>
    <delay_constant max="10e-12" in_port="adder.cin" out_port="adder.cout"/>
</pb_type>
```

specifies that all the edges of 300ps delays, except to cin to cout edge which has a delay of 10ps.

**Sequential block (no internal paths)**

A typical sequential block is a D-Flip-Flop (DFF). DFFs have no internal timing paths between their input and output ports.

**Note:** If you are using BLIF’s .latch directive to represent DFFs there is no need to explicitly provide a <model> definition, as it is supported by default.

Sequential model ports are specified by providing the clock="<name>" attribute, where <name> is the name of the associated clock ports. The associated clock port must have is_clock="1" specified to indicate it is a clock.

```xml
<model name="dff">
    <input_ports>
```

(continues on next page)
The physical timing delays are specified on any pb_type instances of the model. In the example below the setup-time of the input is specified as 66ps, while the clock-to-q delay of the output is set to 124ps.

```xml
<model name="ff" blif_model=".subckt dff" num_pb="1">
  <input name="D" num_pins="1"/>
  <output name="Q" num_pins="1"/>
  <clock name="clk" num_pins="1"/>
  <T_setup value="66e-12" port="ff.D" clock="clk"/>
  <T_clock_to_Q max="124e-12" port="ff.Q" clock="clk"/>
</model>
```

Mixed Sequential/Combinational Block

It is possible to define a block with some sequential ports and some combinational ports. In the example below, the single_port_ram_mixed has sequential input ports: we, addr and data (which are controlled by clk).

However the output port (out) is a combinational output, connected internally to the we, addr and data input registers.

```xml
<model name="single_port_ram_mixed">
  <input_ports>
    <port name="we" clock="clk" combinational_sink_ports="out"/>
    <port name="addr" clock="clk" combinational_sink_ports="out"/>
    <port name="data" clock="clk" combinational_sink_ports="out"/>
    <port name="clk" is_clock="1"/>
  </input_ports>
</model>
```
In the `pb_type` we define the external setup time of the input registers (50ps) as we did for `Sequential block (no internal paths)`. However, we also specify the following additional timing information:

- The internal clock-to-q delay of the input registers (200ps)
- The combinational delay from the input registers to the `out` port (800ps)

```xml
<pb_type name="mem_sp" blif_model=".subckt single_port_ram_mixed" num_pb="1">
  <input name="addr" num_pins="9"/>
  <input name="data" num_pins="64"/>
  <input name="we" num_pins="1"/>
  <output name="out" num_pins="64"/>
  <clock name="clk" num_pins="1"/>
</model>

<!-- External input register timing -->
<T_setup value="50e-12" port="mem_sp.addr" clock="clk"/>
<T_setup value="50e-12" port="mem_sp.data" clock="clk"/>
<T_setup value="50e-12" port="mem_sp.we" clock="clk"/>

<!-- Internal input register timing -->
<T_clock_to_Q max="200e-12" port="mem_sp.addr" clock="clk"/>
```

(continues on next page)
Sequential block (with internal paths)

Some primitives represent more complex architecture primitives, which have timing paths contained completely within the block.

The model below specifies a sequential single-port RAM. The ports `we`, `addr`, and `data` are sequential inputs, while the port `out` is a sequential output. `clk` is the common clock.

![Sequential single port RAM diagram](image-url)

Fig. 9.19: Sequential single port ram
Similarly to Mixed Sequential/Combinational Block the pb_type defines the input register timing:

- external input register setup time (50ps)
- internal input register clock-to-q time (200ps)

Since the output port out is sequential we also define the:

- internal output register setup time (60ps)
- external output register clock-to-q time (300ps)

The combinational delay between the input and output registers is set to 740ps.

Note the internal path from the input to output registers can limit the maximum operating frequency. In this case the internal path delay is 1ns (200ps + 740ps + 60ps) limiting the maximum frequency to 1 GHz.

```xml
<pb_type name="mem_sp" blif_model=".subckt single_port_ram_seq" num_pb="1">
  <input name="addr" num_pins="9"/>
  <input name="data" num_pins="64"/>
  <input name="we" num_pins="1"/>
  <output name="out" num_pins="64"/>
  <clock name="clk" num_pins="1"/>

  <!-- External input register timing -->
  <T_setup value="50e-12" port="mem_sp.addr" clock="clk"/>
  <T_setup value="50e-12" port="mem_sp.data" clock="clk"/>
  <T_setup value="50e-12" port="mem_sp.we" clock="clk"/>

  <!-- Internal input register timing -->
  <T_clock_to_Q max="200e-12" port="mem_sp.addr" clock="clk"/>
  <T_clock_to_Q max="200e-12" port="mem_sp.data" clock="clk"/>
  <T_clock_to_Q max="200e-12" port="mem_sp.we" clock="clk"/>

  <!-- Internal combinational delay -->
  <delay_constant max="740e-12" in_port="mem_sp.addr" out_port="mem_sp.out"/>
  <delay_constant max="740e-12" in_port="mem_sp.data" out_port="mem_sp.out"/>
  <delay_constant max="740e-12" in_port="mem_sp.we" out_port="mem_sp.out"/>

  <!-- Internal output register timing -->
  <T_setup value="60e-12" port="mem_sp.out" clock="clk"/>

  <!-- External output register timing -->
  <T_clock_to_Q max="300e-12" port="mem_sp.out" clock="clk"/>
</pb_type>
```
Sequential block (with internal paths and combinational input)

A primitive may have a mix of sequential and combinational inputs.

The model below specifies a mostly sequential single-port RAM. The ports `addr` and `data` are sequential inputs, while the port `we` is a combinational input. The port `out` is a sequential output. `clk` is the common clock.

![Sequential single port ram with a combinational input](image)

We use register delays similar to *Sequential block (with internal paths)*. However we also specify the purely combinational delay between the combinational `we` input and sequential output `out` (800ps). Note that the setup time of the output register still affects the `we` to `out` path for an effective delay of 860ps.

```xml
<model name="single_port_ram_seq_comb">
  <input_ports>
    <port name="we" combinational_sink_ports="out"/>
    <port name="addr" clock="clk" combinational_sink_ports="out"/>
    <port name="data" clock="clk" combinational_sink_ports="out"/>
    <port name="clk" is_clock="1"/>
  </input_ports>
  <output_ports>
    <port name="out" clock="clk"/>
  </output_ports>
</model>
```

```xml
<pb_type name="mem_sp" blif_model=".subckt single_port_ram_seq_comb" num_pb="1">
  <input name="addr" num_pins="9"/>
  <input name="data" num_pins="64"/>
  <input name="we" num_pins="1"/>
  <output name="out" num_pins="64"/>
  <clock name="clk" num_pins="1"/>
</pb_type>
```

(continues on next page)
Multi-clock Sequential block (with internal paths)

It is also possible for a sequential primitive to have multiple clocks. The following model represents a multi-clock simple dual-port sequential RAM with:

- one write port (addr1 and data1, we1) controlled by clk1, and
- one read port (addr2 and data2) controlled by clk2.

```xml
<model name="multiclock_dual_port_ram">
  <input_ports>
    <!-- Write Port -->
    <port name="we1" clock="clk1" combinational_sink_ports="data2"/>
    <port name="addr1" clock="clk1" combinational_sink_ports="data2"/>
    <port name="data1" clock="clk1" combinational_sink_ports="data2"/>
    <port name="clk1" is_clock="1"/>

    <!-- Read Port -->
    <port name="addr2" clock="clk2" combinational_sink_ports="data2"/>
    <port name="clk2" is_clock="1"/>
  </input_ports>
  <output_ports>
    <!-- Read Port -->
    <port name="data2" clock="clk2" combinational_sink_ports="data2"/>
  </output_ports>
</model>
```

On the pb_type the input and output register timing is defined similarly to Sequential block (with internal paths), except multiple clocks are used.
Fig. 9.21: Multi-clock sequential simple dual port ram

```xml
<pb_type name="mem_dp" blif_model=".subckt multiclock_dual_port_ram" num_pb="1">
    <input name="addr1" num_pins="9"/>
    <input name="data1" num_pins="64"/>
    <input name="we1" num_pins="1"/>
    <input name="addr2" num_pins="9"/>
    <output name="data2" num_pins="64"/>
    <clock name="clk1" num_pins="1"/>
    <clock name="clk2" num_pins="1"/>

    <!-- External input register timing -->
    <T_setup value="50e-12" port="mem_dp.addr1" clock="clk1"/>
    <T_setup value="50e-12" port="mem_dp.data1" clock="clk1"/>
    <T_setup value="50e-12" port="mem_dp.we1" clock="clk1"/>
    <T_setup value="50e-12" port="mem_dp.addr2" clock="clk2"/>

    <!-- Internal input register timing -->
    <T_clock_to_Q max="200e-12" port="mem_dp.addr1" clock="clk1"/>
    <T_clock_to_Q max="200e-12" port="mem_dp.data1" clock="clk1"/>
    <T_clock_to_Q max="200e-12" port="mem_dp.we1" clock="clk1"/>
    <T_clock_to_Q max="200e-12" port="mem_dp.addr2" clock="clk2"/>

    <!-- Internal combinational delay -->
    <delay_constant max="740e-12" in_port="mem_dp.addr1" out_port="mem_dp.data2"/>
    <delay_constant max="740e-12" in_port="mem_dp.data1" out_port="mem_dp.data2"/>
```

(continues on next page)
Clock Generators

Some blocks (such as PLLs) generate clocks on-chip. To ensure that these generated clocks are identified as clock sources, the associated model output port should be marked with `is_clock="1"`.

As an example consider the following simple PLL model:

```verbatim
<model name="simple_pll">
    <input_ports>
        <port name="in_clock" is_clock="1"/>
    </input_ports>
    <output_ports>
        <port name="out_clock" is_clock="1"/>
    </output_ports>
</model>
```

The port named `in_clock` is specified as a clock sink, since it is an input port with `is_clock="1"` set.

The port named `out_clock` is specified as a clock generator, since it is an output port with `is_clock="1"` set.

**Note:** Clock generators should not be the combinational sinks of primitive input ports.

Consider the following example netlist:

```verbatim
.subckt simple_pll in_clock=clk out_clock=clk_pll
```

Since we have specified that `simple_pll.out_clock` is a clock generator (see above), the user must specify what the clock relationship is between the input and output clocks. This information must be either specified in the SDC file (if no SDC file is specified VPR's default timing constraints will be used instead).

**Note:** VPR has no way of determining what the relationship is between the clocks of a black-box primitive.

Consider the case where the `simple_pll` above creates an output clock which is 2 times the frequency of the input clock. If the input clock period was 10ns then the SDC file would look like:

```verbatim
create_clock clk -period 10
create_clock clk_pll -period 5  #Twice the frequency of clk
```

It is also possible to specify in SDC that there is a phase shift between the two clocks.
Clock Buffers & Muxes

Some architectures contain special primitives for buffering or controlling clocks. VTR supports modelling these using the `is_clock` attribute on the model to differentiate between 'data' and 'clock' signals, allowing users to control how clocks are traced through these primitives.

When VPR traces through the netlist it will propagate clocks from clock inputs to the downstream combinationally connected pins.

Clock Buffers/Gates

Consider the following black-box clock buffer with an enable:

```verilog
.subckt clkbufce 
  in=clk3 
  enable=clk3_enable 
  out=clk3_buf
```

We wish to have VPR understand that the `in` port of the `clkbufce` connects to the `out` port, and that as a result the nets `clk3` and `clk3_buf` are equivalent.

This is accomplished by tagging the `in` port as a clock (`is_clock="1"`), and combinationally connecting it to the `out` port (`combinational_sink_ports="out"`):

```xml
<model name="clkbufce">
  <input_ports>
    <port name="in" combinational_sink_ports="out" is_clock="1"/>
    <port name="enable" combinational_sink_ports="out"/>
  </input_ports>
  <output_ports>
    <port name="out"/>
  </output_ports>
</model>
```

With the corresponding `pb_type`:

```xml
<pb_type name="clkbufce" blif_model="clkbufce" num_pb="1">
  <clock name="in" num_pins="1"/>
  <input name="enable" num_pins="1"/>
  <output name="out" num_pins="1"/>
  <delay_constant max="10e-12" in_port="clkbufce.in" out_port="clkbufce.out"/>
  <delay_constant max="5e-12" in_port="clkbufce.enable" out_port="clkbufce.out"/>
</pb_type>
```

Notably, although the `enable` port is combinationally connected to the `out` port it will not be considered as a potential clock since it is not marked with `is_clock="1"`.  

9.2. Architecture Modeling
Clock Muxes

Another common clock control block is a clock mux, which selects from one of several potential clocks.

For instance, consider:

```
.subckt clkmux 
  clk1=clka 
  clk2=clkb 
  sel=select 
  clk_out=clk_downstream
```

which selects one of two input clocks (clk1 and clk2) to be passed through to (clk_out), controlled on the value of sel.

This could be modelled as:

```
<model name="clkmux">
  <input_ports>
    <port name="clk1" combinational_sink_ports="clk_out" is_clock="1"/>
    <port name="clk2" combinational_sink_ports="clk_out" is_clock="1"/>
    <port name="sel" combinational_sink_ports="clk_out"/>
  </input_ports>
  <output_ports>
    <port name="clk_out"/>
  </output_ports>
</model>
```

where both input clock ports clk1 and clk2 are tagged with is_clock="1" and combinationally connected to the clk_out port. As a result both nets clka and clkb in the netlist would be identified as independent clocks feeding clk_downstream.

**Note:** Clock propagation is driven by netlist connectivity so if one of the input clock ports (e.g. clk1) was disconnected in the netlist no associated clock would be created/considered.
Clock Mux Timing Constraints

For the clock mux example above, if the user specified the following SDC timing constraints:

```bash
create_clock -period 3 clka
create_clock -period 2 clkb
```

VPR would propagate both clka and clkb through the clock mux. Therefore the logic connected to clk_downstream would be analyzed for both the clka and clkb constraints.

Most likely (unless clka and clkb are used elsewhere) the user should additionally specify:

```bash
set_clock_groups -exclusive -group clka -group clkb
```

Which avoids analyzing paths between the two clocks (i.e. clka -> clkb and clkb -> clka) which are not physically realizable. The muxing logic means only one clock can drive clk_downstream at any point in time (i.e. the mux enforces that clka and clkb are mutually exclusive). This is the behaviour of VPR's default timing constraints.

9.3 Running the Titan Benchmarks

This tutorial describes how to run the Titan benchmarks with VTR.

9.3.1 Integrating the Titan benchmarks into VTR

The Titan benchmarks take up a large amount of disk space and are not distributed directly with VTR.

The Titan benchmarks can be automatically integrated into the VTR source tree by running the following from the root of the VTR source tree:

```bash
$ make get_titan_benchmarks
```

which downloads and extracts the benchmarks into the VTR source tree:

Warning: A typical Titan release is a ~1GB download, and uncompresses to ~10GB.
Starting download in 15 seconds...
Downloading http://www.eecg.utoronto.ca/~kmurray/titan/titan_release_1.1.0.tar.gz
...........................................................................................
→
Download http://www.eecg.utoronto.ca/~kmurray/titan/titan_release_1.1.0.md5
Verifying checksum
OK
Searching release for benchmarks and architectures...
Extracting titan_release_1.1.0/benchmarks/titan23/sparcT2_core/netlists/sparcT2_core_stratixiv_arch_timing.blif to ./vtr_flow/benchmarks/titan_blif/sparcT2_core_stratixiv_arch_timing.blif
Extracting titan_release_1.1.0/benchmarks/titan23/LU230/netlists/LU230_stratixiv_arch_timing.blif to ./vtr_flow/benchmarks/titan_blif/LU230_stratixiv_arch_timing.blif
Extracting titan_release_1.1.0/benchmarks/titan23/segmentation/netlists/segmentation_stratixiv_arch_timing.blif to ./vtr_flow/benchmarks/titan_blif/segmentation_stratixiv_arch_timing.blif
Extracting titan_release_1.1.0/benchmarks/titan23/openCV/netlists/openCV_stratixiv_arch_timing.blif to ./vtr_flow/benchmarks/titan_blif/openCV_stratixiv_arch_timing.blif
Extracting `titan_release_1.1.0/benchmarks/titan23/bitcoin_miner/netlists/bitcoin_miner_stratixiv_arch_timing.blif` to `./vtr_flow/benchmarks/titan_blif/bitcoin_miner_stratixiv_arch_timing.blif`

Extracting `titan_release_1.1.0/benchmarks/titan23/sparcT1_chip2/netlists/sparcT1_chip2_stratixiv_arch_timing.blif` to `./vtr_flow/benchmarks/titan_blif/sparcT1_chip2_stratixiv_arch_timing.blif`

Extracting `titan_release_1.1.0/benchmarks/titan23/mes_noc/netlists/mes_noc_stratixiv_arch_timing.blif` to `./vtr_flow/benchmarks/titan_blif/mes_noc_stratixiv_arch_timing.blif`

Extracting `titan_release_1.1.0/benchmarks/titan23/bitonic_mesh/netlists/bitonic_mesh_stratixiv_arch_timing.blif` to `./vtr_flow/benchmarks/titan_blif/bitonic_mesh_stratixiv_arch_timing.blif`

Extracting `titan_release_1.1.0/benchmarks/titan23/dart/netlists/dart_stratixiv_arch_timing.blif` to `./vtr_flow/benchmarks/titan_blif/dart_stratixiv_arch_timing.blif`

Extracting `titan_release_1.1.0/benchmarks/titan23/cholesky_bdti/netlists/cholesky_bdti_stratixiv_arch_timing.blif` to `./vtr_flow/benchmarks/titan_blif/cholesky_bdti_stratixiv_arch_timing.blif`

Extracting `titan_release_1.1.0/benchmarks/titan23/stereo_vision/netlists/stereo_vision_stratixiv_arch_timing.blif` to `./vtr_flow/benchmarks/titan_blif/stereo_vision_stratixiv_arch_timing.blif`

Extracting `titan_release_1.1.0/benchmarks/titan23/neuron/netlists/neuron_stratixiv_arch_timing.blif` to `./vtr_flow/benchmarks/titan_blif/neuron_stratixiv_arch_timing.blif`

Extracting `titan_release_1.1.0/benchmarks/titan23/gaussianblur/netlists/gaussianblur_stratixiv_arch_timing.blif` to `./vtr_flow/benchmarks/titan_blif/gaussianblur_stratixiv_arch_timing.blif`

Extracting `titan_release_1.1.0/benchmarks/titan23/gsm_switch/netlists/gsm_switch_stratixiv_arch_timing.blif` to `./vtr_flow/benchmarks/titan_blif/gsm_switch_stratixiv_arch_timing.blif`

Extracting `titan_release_1.1.0/benchmarks/titan23/sparcT1_core/netlists/sparcT1_core_stratixiv_arch_timing.blif` to `./vtr_flow/benchmarks/titan_blif/sparcT1_core_stratixiv_arch_timing.blif`

Extracting `titan_release_1.1.0/benchmarks/titan23/des90/netlists/des90_stratixiv_arch_timing.blif` to `./vtr_flow/benchmarks/titan_blif/des90_stratixiv_arch_timing.blif`

Extracting `titan_release_1.1.0/benchmarks/titan23/LU_Network/netlists/LU_Network_stratixiv_arch_timing.blif` to `./vtr_flow/benchmarks/titan_blif/LU_Network_stratixiv_arch_timing.blif`

Extracting `titan_release_1.1.0/benchmarks/titan23/denoise/netlists/denoise_stratixiv_arch_timing.blif` to `./vtr_flow/benchmarks/titan_blif/denoise_stratixiv_arch_timing.blif`

Extracting `titan_release_1.1.0/benchmarks/titan23/stap_qrd/netlists/stap_qrd_stratixiv_arch_timing.blif` to `./vtr_flow/benchmarks/titan_blif/stap_qrd_stratixiv_arch_timing.blif`

Extracting `titan_release_1.1.0/benchmarks/titan23/directrf/netlists/directrf_stratixiv_arch_timing.blif` to `./vtr_flow/benchmarks/titan_blif/directrf_stratixiv_arch_timing.blif`

Extracting `titan_release_1.1.0/benchmarks/titan23/SLAM_spheric/netlists/SLAM_spheric_stratixiv_arch_timing.blif` to `./vtr_flow/benchmarks/titan_blif/SLAM_spheric_stratixiv_arch_timing.blif`

Extracting `titan_release_1.1.0/benchmarks/titan23/minres/netlists/minres_stratixiv_arch_timing.blif` to `./vtr_flow/benchmarks/titan_blif/minres_stratixiv_arch_timing.blif`

Extracting `titan_release_1.1.0/arch/stratixiv_arch.timing.no_pack_patterns.xml` to `./vtr_flow/arch/titan/stratixiv_arch.timing.no_pack_patterns.xml`
Extracting titan_release_1.1.0/arch/stratixiv_arch.timing.xml to ./vtr_flow/arch/titan/
   --stratixiv_arch.timing.xml
Extracting titan_release_1.1.0/arch/stratixiv_arch.timing.no_directlink.xml to ./vtr_
   --flow/arch/titan/stratixiv_arch.timing.no_directlink.xml
Extracting titan_release_1.1.0/arch/stratixiv_arch.timing.no_chain.xml to ./vtr_flow/
   --arch/titan/stratixiv_arch.timing.no_chain.xml
Done
Titan architectures: vtr_flow/arch/titan
Titan benchmarks: vtr_flow/benchmarks/titan_blif

Once completed all the Titan benchmark BLIF netlists can be found under $VTR_ROOT/vtr_flow/benchmarks/titan_blif, and the Titan architectures under $VTR_ROOT/vtr_flow/arch/titan.

Note: $VTR_ROOT corresponds to the root of the VTR source tree.

9.3.2 Running benchmarks manually

Once the benchmarks have been integrated into VTR they can be run manually.

For example, the follow uses VPR to implement the neuron benchmark onto the startixiv_arch.timing.xml architecture at a channel width of 300 tracks:

$ vpr $VTR_ROOT/vtr_flow/arch/titan/stratixiv_arch.timing.xml $VTR_ROOT/vtr_flow/
   benchmarks/titan_blif/neuron_stratixiv_arch_timing.blif --route_chan_width 300

9.4 Post-Implementation Timing Simulation

This tutorial describes how to simulate a circuit which has been implemented by VPR with back-annotated timing delays.

Back-annotated timing simulation is useful for a variety of reasons:

• Checking that the circuit logic is correctly implemented
• Checking that the circuit behaves correctly at speed with realistic delays
• Generating VCD (Value Change Dump) files with realistic delays (e.g. for power estimation)

9.4.1 Generating the Post-Implementation Netlist

For the purposes of this tutorial we will be using the stereovision3 benchmark, and will target the k6_N10_40nm architecture.

First lets create a directory to work in:

$ mkdir timing_sim_tut
$ cd timing_sim_tut

Next we'll copy over the stereovision3 benchmark netlist in BLIF format and the FPGA architecture description:
Fig. 9.22: Timing simulation waveform for stereovision3
$ cp $VTR_ROOT/vtr_flow/benchmarks/vtr_benchmarks_blif/stereovision3.blif.
$ cp $VTR_ROOT/vtr_flow/arch/timing/k6_N10_40nm.xml.

Note: Replace $VTR_ROOT with the root directory of the VTR source tree.

Now we can run VPR to implement the circuit onto the k6_N10_40nm architecture. We also need to provide the vpr --gen_post_synthesis_netlist option to generate the post-implementation netlist and dump the timing information in Standard Delay Format (SDF):

$ vpr k6_N10_40nm.xml stereovision3.blif --gen_post_synthesis_netlist on

Once VPR has completed we should see the generated verilog netlist and SDF:

$ ls *.v *.sdf
sv_chip3_hierarchy_no_mem_post_synthesis.sdf sv_chip3_hierarchy_no_mem_post_synthesis.v

9.4.2 Inspecting the Post-Implementation Netlist

Lets take a quick look at the generated files.

First is a snippet of the verilog netlist:

Listing 9.1: Verilog netlist snippet

```verilog
fpga_interconnect/\routing_segment_lut_n616_output_0_0_to_lut_n497_input_0_4 (  
 .datain(lut_n616_output_0_0 ),  
 .dataout(lut_n497_input_0_4 )
);

//Cell instances
LUT_K (#(  
 .K(6),  
 .LUT_MASK(64'b0000000000000000000000000000000000100001001000100000000100000010)  
 ) )
 lut_n452 (  
 .in({  
 1'b0,  
 lut_n452_input_0_4 ,  
 lut_n452_input_0_3 ,  
 lut_n452_input_0_2 ,  
 1'b0,  
 lut_n452_input_0_0 }),  
 .out(lut_n452_output_0_0 )
);

DFF #(  
 .INITIAL_VALUE(1'b0)  
 )
 latch_top^FF_NODE~387 (  
 .D(\latch_top^FF_NODE~387_input_0_0 ),  
 .Q(\latch_top^FF_NODE~387_output_0_0 ),  
)
```

(continues on next page)

9.4. Post-Implementation Timing Simulation
Here we see three primitives instantiated:

- **fpga_interconnect** represent connections between netlist primitives
- **LUT_K** represent look-up tables (LUTs) (corresponding to .names in the BLIF netlist). Two parameters define the LUTs functionality:
  - K the number of inputs, and
  - LUT_MASK which defines the logic function.
- **DFF** represents a D-Flip-Flop (corresponding to .latch in the BLIF netlist).
  - The INITIAL_VALUE parameter defines the Flip-Flop’s initial state.

Different circuits may produce other types of netlist primitives corresponding to hardened primitive blocks in the FPGA such as adders, multipliers and single or dual port RAM blocks.

**Note:** The different primitives produced by VPR are defined in $VTR_ROOT/vtr_flow/primitives.v

Let's now take a look at the Standard Delay Format (SDF) file:

```plaintext
Listing 9.2: SDF snippet

(CELL "fpga_interconnect")
(INSTANCE routing_segment_lut_n616_output_0_0_to_lut_n497_input_0_4)
(Delay
  (ABSOLUTE
  )
)

(CELL "LUT_K")
(INSTANCE lut_n452)
(Delay
  (ABSOLUTE
    (IOPATH in[0] out (261:261:261) (261:261:261))
  )
)

(CELL "DFF")
(INSTANCE latch_top\^FF_NODE~387)
(Delay
  (ABSOLUTE
```
The SDF defines all the delays in the circuit using the delays calculated by VPR's STA engine from the architecture file we provided.

Here we see the timing description of the cells in Listing 9.1.

In this case the routing segment routing_segment_lut_n616_output_0_0_to_lut_n497_input_0_4 has a delay of 312.648 ps, while the LUT lut_n452 has a delay of 261 ps from each input to the output. The DFF latch_top\^FF_NODE\~387 has a clock-to-q delay of 124 ps and a setup time of 66ps.

9.4.3 Creating a Test Bench

In order to simulate a benchmark we need a test bench which will stimulate our circuit (the Device-Under-Test or DUT).

An example test bench which will randomly perturb the inputs is shown below:

Listing 9.3: The test bench tb.sv.

```verilog

`timescale 1ps/1ps
module tb();

localparam CLOCK_PERIOD = 8000;
localparam CLOCK_DELAY = CLOCK_PERIOD / 2;

//Simulation clock
logic sim_clk;

//DUT inputs
logic \top^tm3_clk_v0 ;
logic \top^tm3_clk_v2 ;
logic \top^tm3_vdin_l1c ;
logic \top^tm3_vdin_vs ;
logic \top^tm3_vdin_href ;
logic \top^tm3_vdin_cref ;
logic \top^tm3_vdin_rts0 ;
logic \top^tm3_vdin_vpo~0 ;
logic \top^tm3_vdin_vpo~1 ;
logic \top^tm3_vdin_vpo~2 ;
logic \top^tm3_vdin_vpo~3 ;
logic \top^tm3_vdin_vpo~4 ;
logic \top^tm3_vdin_vpo~5 ;
logic \top^tm3_vdin_vpo~6 ;
logic \top^tm3_vdin_vpo~7 ;
logic \top^tm3_vdin_vpo~8 ;
logic \top^tm3_vdin_vpo~9 ;
logic \top^tm3_vdin_vpo~10 ;
```
logic \top^tm3_vidin_vpo~11 ;
logic \top^tm3_vidin_vpo~12 ;
logic \top^tm3_vidin_vpo~13 ;
logic \top^tm3_vidin_vpo~14 ;
logic \top^tm3_vidin_vpo~15 ;

//DUT outputs
logic \top^tm3_vidin_sda ;
logic \top^tm3_vidin_scl ;
logic \top^vidin_new_data ;
logic \top^vidin_rgb_reg~0 ;
logic \top^vidin_rgb_reg~1 ;
logic \top^vidin_rgb_reg~2 ;
logic \top^vidin_rgb_reg~3 ;
logic \top^vidin_rgb_reg~4 ;
logic \top^vidin_rgb_reg~5 ;
logic \top^vidin_rgb_reg~6 ;
logic \top^vidin_rgb_reg~7 ;
logic \top^vidin_addr_reg~0 ;
logic \top^vidin_addr_reg~1 ;
logic \top^vidin_addr_reg~2 ;
logic \top^vidin_addr_reg~3 ;
logic \top^vidin_addr_reg~4 ;
logic \top^vidin_addr_reg~5 ;
logic \top^vidin_addr_reg~6 ;
logic \top^vidin_addr_reg~7 ;
logic \top^vidin_addr_reg~8 ;
logic \top^vidin_addr_reg~9 ;
logic \top^vidin_addr_reg~10 ;
logic \top^vidin_addr_reg~11 ;
logic \top^vidin_addr_reg~12 ;
logic \top^vidin_addr_reg~13 ;
logic \top^vidin_addr_reg~14 ;
logic \top^vidin_addr_reg~15 ;
logic \top^vidin_addr_reg~16 ;
logic \top^vidin_addr_reg~17 ;
logic \top^vidin_addr_reg~18 ;

//Instantiate the dut
sv_chip3_hierarchy_no_mem dut ( .* );

//Load the SDF
initial $sdf_annotate("sv_chip3_hierarchy_no_mem_post_synthesis.sdf", dut);

//The simulation clock
initial sim_clk = '1;
always #CLOCK_DELAY sim_clk = ~sim_clk;

//The circuit clocks
assign \top^tm3_clk_v0 = sim_clk;
assign \top^tm3_clk_v2 = sim_clk;
//Randomized input
always@ (posedge sim_clk) begin
    \top^tm3_vidin_llc <= $urandom_range(1,0);
    \top^tm3_vidin_vs <= $urandom_range(1,0);
    \top^tm3_vidin_href <= $urandom_range(1,0);
    \top^tm3_vidin_cref <= $urandom_range(1,0);
    \top^tm3_vidin_rts0 <= $urandom_range(1,0);
    \top^tm3_vidin_vpo~0 <= $urandom_range(1,0);
    \top^tm3_vidin_vpo~1 <= $urandom_range(1,0);
    \top^tm3_vidin_vpo~2 <= $urandom_range(1,0);
    \top^tm3_vidin_vpo~3 <= $urandom_range(1,0);
    \top^tm3_vidin_vpo~4 <= $urandom_range(1,0);
    \top^tm3_vidin_vpo~5 <= $urandom_range(1,0);
    \top^tm3_vidin_vpo~6 <= $urandom_range(1,0);
    \top^tm3_vidin_vpo~7 <= $urandom_range(1,0);
    \top^tm3_vidin_vpo~8 <= $urandom_range(1,0);
    \top^tm3_vidin_vpo~9 <= $urandom_range(1,0);
    \top^tm3_vidin_vpo~10 <= $urandom_range(1,0);
    \top^tm3_vidin_vpo~11 <= $urandom_range(1,0);
    \top^tm3_vidin_vpo~12 <= $urandom_range(1,0);
    \top^tm3_vidin_vpo~13 <= $urandom_range(1,0);
    \top^tm3_vidin_vpo~14 <= $urandom_range(1,0);
    \top^tm3_vidin_vpo~15 <= $urandom_range(1,0);
end
endmodule

The testbench instantiates our circuit as `dut` at line 69. To load the SDF we use the `$sdf_annotate()` system task (line 72) passing the SDF filename and target instance. The clock is defined on lines 75-76 and the random circuit inputs are generated at the rising edge of the clock on lines 84-104.

### 9.4.4 Performing Timing Simulation in Modelsim

To perform the timing simulation we will use *Modelsim*, an HDL simulator from Mentor Graphics.

**Note:** Other simulators may use different commands, but the general approach will be similar.

It is easiest to write a `tb.do` file to setup and configure the simulation:

Listing 9.4: Modelsim do file `tb.do`. Note that `$VTR_ROOT` should be replaced with the relevant path.

```plaintext
#Enable command logging
transcript on

#Setup working directories
if {{file exists gate_work}} {
    vdel -lib gate_work -all
}
```

(continues on next page)
We link together the post-implementation netlist, test bench and VTR primitives on lines 12-14. The simulation is then configured on line 17, some of the options are worth discussing in more detail:

- **+bitblast**: Ensures Modelsim interprets the primitives in `primitives.v` correctly for SDF back-annotation.

**Warning:** Failing to provide **+bitblast** can cause errors during SDF back-annotation

- **+sdf_verbose**: Produces more information about SDF back-annotation, useful for verifying that back-annotation succeeded.

Lastly, we tell the simulation to run on line 31.

Now that we have a `.do` file, let's launch the modelsim GUI:

```
$ vsim
```

and then run our `.do` file from the internal console:

```
ModelSim> do tb.do
```

Once the simulation completes we can view the results in the waveform view as shown in *at the top of the page*, or process the generated VCD file `sim.vcd`. 
10.1 FPGA Assembly (FASM) Output Support

After VPR has generated a placed and routed design, the *genfasm* utility can emit a FASM file to represent the design at a level detailed enough to allow generation of a bitstream to program a device. This FASM output file is enabled by FASM metadata encoded in the VPR architecture definition and routing graph. The output FASM file can be converted into a bitstream format suitable to program the target architecture via architecture specific tooling. Current devices that can be programmed using the vpr + fasm flow include Lattice iCE40 and Xilinx Artix-7 devices, with work on more devices underway. More information on supported devices is available from the Symbiflow website and an overview of the flow for Artix-7 devices is described in IEEE Micro [MurrayAnsellRothman+20].

10.1.1 FASM metadata

The *genfasm* utility uses metadata blocks (see Architecture metadata) attached to the architecture definition and routing graph to emit FASM features. By adding FASM specific metadata to both the architecture definition and the routing graph, a FASM file that represents the place and routed design can be generated.

All metadata tags are ignored when packing, placing and routing. After VPR has been completed placement, *genfasm* utility loads the VPR output files (.net, .place, .route) and then uses the FASM metadata to emit a FASM file. The following metadata “keys” are recognized by *genfasm*:

- “fasm_prefix”
- “fasm_features”
- “fasm_type” and “fasm_lut”
- “fasm_mux”
- “fasm_params”

10.1.2 Invoking genfasm

*genfasm* expects that place and route on the design is completed (e.g. .net, .place, .route files are present), so ensure that routing is complete before executing *genfasm*. *genfasm* should be invoked in the same subdirectory as the routing output. The output FASM file will be written to <blif_root>.fasm.
10.1.3 FASM prefixing

FASM feature names have structure through their prefixes. In general the first part of the FASM feature is the location of the feature, such as the name of the tile the feature is located in, e.g. INT_L_X5Y6 or CLBLL_L_X10Y12. The next part is typically an identifier within the tile. For example a CLBLL tile has two slices, so the next part of the FASM feature name is the slice identifier, e.g. SLICE_X0 or SLICE_X1.

Now consider the CLBLL_L pb_type. This pb_type is repeated in the grid for each tile of that type. To allow one pb_type definition to be defined, the “fasm_prefix” metadata tag is allowed to be attached at the layout level on the <single> tag. This enables the same pb_type to be used for all CLBLL_L tiles, and the “fasm_prefix” is prepended to all FASM metadata within that pb_type. For example:

```
<single priority="1" type="BLK_TI-CLBLL_L" x="35" y="51">
  <metadata>
    <meta name="fasm_prefix">CLBLL_L_X12Y100</meta>
  </metadata>
</single>
<single priority="1" type="BLK_TI-CLBLL_L" x="35" y="50">
  <metadata>
    <meta name="fasm_prefix">CLBLL_L_X12Y101</meta>
  </metadata>
</single>
```

“fasm_prefix” tags can also be used within a pb_type to handle repeated features. For example in the CLB, there are 4 LUTs that can be described by a common pb_type, except that the prefix changes for each. For example, consider the FF's within a CLB. There are 8 FF’s that share a common structure, except for a prefix change. “fasm_prefix” can be a space separated list to assign prefixes to the index of the pb_type, rather than needing to emit N copies of the pb_type with varying prefixes.

```
<pb_type name="BEL_FF-FDSE_or_FDRE" num_pb="8">
  <input name="D" num_pins="1"/>
  <input name="CE" num_pins="1"/>
  <clock name="C" num_pins="1"/>
  <input name="SR" num_pins="1"/>
  <output name="Q" num_pins="1"/>
  <metadata>
    <meta name="fasm_prefix">AFF BFF CFF DFF A5FF B5FF C5FF D5FF</meta>
  </metadata>
</pb_type>
```
Construction of the prefix

“fasm_prefix” is accumulated throughout the structure of the architecture definition. Each “fasm_prefix” is joined together with a period (‘.’), and then a period is added after the prefix before the FASM feature name.

10.1.4 Simple FASM feature emissions

In cases where a FASM feature needs to be emitted simply via use of a pb_type, the “fasm_features” tag can be used. If the pb_type (or mode) is selected, then all “fasm_features” in the metadata will be emitted. Multiple features can be listed, whitespace separated. Example:

```
<metadata>
  <meta name="fasm_features">ZRST</meta>
</metadata>
```

The other place that “fasm_features” is used heavily is on <edge> tags in the routing graph. If an edge is used in the final routed design, “genfasm” will emit features attached to the edge. Example:

```
<edge sink_node="431195" src_node="418849" switch_id="0">
  <metadata>
    <meta name="fasm_features">HCLK_R_X58Y130.HCLK_LEAF_CLK_B_TOP4.HCLK_CK_BUFHCLK7 HCLK_R_X58Y130.ENABLE_BUFFER.HCLK_CK_BUFHCLK7</meta>
  </metadata>
</edge>
```

In this example, when the routing graph connects node 418849 to 431195, two FASM features will be emitted:

- HCLK_R_X58Y130.HCLK_LEAF_CLK_B_TOP4.HCLK_CK_BUFHCLK7
- HCLK_R_X58Y130.ENABLE_BUFFER.HCLK_CK_BUFHCLK7

10.1.5 Emitting LUTs

LUTs are a structure that is explicitly understood by VPR. In order to emit LUTs, two metadata keys must be used, “fasm_type” and “fasm_lut”. “fasm_type” must be either “LUT” or “SPLIT_LUT”. The “fasm_type” modifies how the “fasm_lut” key is interpreted. If the pb_type that the metadata is attached to has no “num_pb” or “num_pb” equals 1, then “fasm_type” can be “LUT”. “fasm_lut” is then the feature that represents the LUT table storage features, example:

```
<metadata>
  <meta name="fasm_type">LUT</meta>
  <meta name="fasm_lut">ALUT.INIT[63:0]</meta>
</metadata>
```

FASM LUT metadata must be attached to the <pb_type> at or within the <mode> tag directly above the <pb_type> with blif_model=".names“. Do note that there is an implicit <mode> tag within intermediate <pb_type> when no explicit <mode> tag is present. The FASM LUT metadata tags will not be recognized attached inside of <pb_type>‘s higher above the leaf type.

When specifying a FASM features with more than one bit, explicitly specify the bit range being set. This is required because “genfasm” does not have access to the actual bit database, and would otherwise not have the width of the feature.
When “fasm_type” is “SPLIT_LUT”, “fasm_lut” must specify both the feature that represents the LUT table storage features and the pb_type path to the LUT being specified. Example:

```xml
<metadata>
  <meta name="fasm_type">SPLIT_LUT</meta>
  <meta name="fasm_lut">
    ALUT.INIT[31:0] = BEL_LT-A5LUT[0]
  </meta>
</metadata>
```

In this case, the LUT in pb_type BEL_LT-A5LUT[0] will use INIT[31:0], and the LUT in pb_type BEL_LT-A5LUT[1] will use INIT[63:32].

### 10.1.6 Within tile interconnect features

When a tile has interconnect feature, e.g. output muxes, the “fasm_mux” tag should be attached to the interconnect tag, likely the `<direct>` or `<mux>` tags. From the perspective of genfasm, the `<direct>` and `<mux>` tags are equivalent. The syntax for the “fasm_mux” newline separated relationship between mux input wire names and FASM features. Example:

```xml
<mux name="D5FFMUX" input="BLK_IG-COMMON_SLICE.DX BLK_IG-COMMON_SLICE.DO5" output="BLK_BB-SLICE_FF.D5[3]">
  <metadata>
    <meta name="fasm_mux">
      BLK_IG-COMMON_SLICE.DO5 : D5FFMUX.IN_A
      BLK_IG-COMMON_SLICE.DX : D5FFMUX.IN_B
    </meta>
  </metadata>
</mux>
```

The above mux connects input BLK_IG-COMMON_SLICE.DX or BLK_IG-COMMON_SLICE.DO5 to BLK_BB-SLICE_FF.D5[3]. When VPR selects BLK_IG-COMMON_SLICE.DO5 for the mux, “genfasm” will emit D5FFMUX.IN_A, etc.

There is not a requirement that all inputs result in a feature being set. In cases where some mux selections result in no feature being set, use “NULL” as the feature name. Example:

```xml
<mux name="CARRY_DI3" input="BLK_IG-COMMON_SLICE.DO5 BLK_IG-COMMON_SLICE.DX" output="BEL_BB-CARRY[2].DI">
  <metadata>
    <meta name="fasm_mux">
      BLK_IG-COMMON_SLICE.DO5 : CARRY4.DCY0
      BLK_IG-COMMON_SLICE.DX : NULL
    </meta>
  </metadata>
</mux>
```

The above examples all used the `<mux>` tag. The “fasm_mux” metadata key can also be used with the `<direct>` tag in the same way, example:

```xml
<direct name="WA7" input="BLK_IG-SLICEM.CX" output="BLK_IG-SLICEM_MODES.WA7">
  <metadata>
    <meta name="fasm_mux">
    </meta>
  </metadata>
</direct>
```
If multiple FASM features are required for a mux, they can be specified using commas as a separator. Example:

```xml
<mux name="D5FFMUX" input="BLK_IG-COMMON_SLICE.DX BLK_IG-COMMON_SLICE.DO5" output="BLK_BB-SLICE_FF.D5[3]">
  <metadata>
    <meta name="fasm_mux">
      BLK_IG-COMMON_SLICE.DO5 : D5FFMUX.IN_A
      BLK_IG-COMMON_SLICE.DX : D5FFMUX.IN_B, D5FF.OTHER_FEATURE
    </meta>
  </metadata>
</mux>
```

### 10.1.7 Passing parameters through to the FASM Output

In many cases there are parameters that need to be passed directly from the input `Extended BLIF (.eblif)` to the FASM file. These can be passed into a FASM feature via the “fasm_params” key. Note that care must be taken to have the “fasm_params” metadata be attached to pb_type that the packer uses, the pb_type with the blif_model= “.subckt”.

The “fasm_params” value is a newline separated list of FASM features to eblif parameters. Example:

```xml
<metadata>
  <meta name="fasm_params">
    INIT[31:0] = INIT_00
    INIT[63:32] = INIT_01
  </meta>
</metadata>
```

The FASM feature is on the left hand side of the equals. When setting a parameter with multiple bits, the bit range must be specified. If the parameter is a single bit, the bit range is not required, but can be supplied for clarity. The right hand side is the parameter name from eblif. If the parameter name is not found in the eblif, that FASM feature will not be emitted.

No errors or warnings will be generated for unused parameters from eblif or unused mappings between eblif parameters and FASM parameters to allow for flexibility in the synthesis output. This does mean it is important to check spelling of the metadata, and create tests that the mapping is working as expected.

Also note that “genfasm” will not accept “x” (unknown/don’t care) or “z” (high impedance) values in parameters. Prior to emitting the eblif for place and route, ensure that all parameters that will be mapped to FASM have a valid “1” or “0”.

---

10.1. FPGA Assembly (FASM) Output Support

327
10.2 Router Diagnosis Tool

The Router Diagnosis tool (route_diag) is an utility that helps developers understand the issues related to the routing phase of VPR. Instead of running the whole routing step, route_diag performs one step of routing, aimed at analyzing specific connections between a SINK/SOURCE nodes pair. Moreover, it is able also to profile all the possible paths of a given SOURCE node.

To correctly run the utility tool, the user needs to compile VTR with the VTR_ENABLE_DEBUG_LOGGING set to ON and found in the CMakeLists.txt configuration file.

The tool is compiled with all the other targets when running the full build of VtR. It is also possible, though, to build the route_diag utility standalone, by running the following command:

```
make route_diag
```

To use the Route Diagnosis tool, the users has different parameters at disposal:

```
--sink_rr_node <int>
  Specifies the SINK RR NODE part of the pair that needs to be analyzed

--source_rr_node <int>
  Specifies the SOURCE RR NODE part of the pair that needs to be analyzed

--router_debug_sink_rr <int>
  Controls when router debugging is enabled for the specified sink RR.
    • For values >= 0, the value is taken as the sink RR Node ID for which to enable router debug output.
    • For values < 0, sink-based router debug output is disabled.
```

The Router Diagnosis tool must be provided at least with the RR GRAPH and the architecture description file to correctly function.
11.1 Contribution Guidelines

Thanks for considering contributing to VTR! Here are some helpful guidelines to follow.

11.1.1 Common Scenarios

I have a question

If you have questions about VTR take a look at our Support Resources.

If the answer to your question wasn’t in the documentation (and you think it should have been), consider enhancing the documentation. That way someone (perhaps your future self!) will be able to quickly find the answer in the future.

I found a bug!

While we strive to make VTR reliable and robust, bugs are inevitable in large-scale software projects.

Please file a detailed bug report. This ensures we know about the problem and can work towards fixing it.

It would be great if VTR supported …

VTR has many features and is highly flexible. Make sure you’ve checkout out all our Support Resources to see if VTR already supports what you want.

If VTR does not support your use case, consider filling an enhancement.

I have a bug-fix/feature I’d like to include in VTR

Great! Submitting bug-fixes and features is a great way to improve VTR. See the guidelines for submitting code.
11.1.2 The Details

Enhancing Documentation

Enhancing documentation is a great way to start contributing to VTR.
You can edit the documentation directly by clicking the Edit on GitHub link of the relevant page, or by editing the re-structured text (.rst) files under doc/src.
Generally it is best to make small incremental changes. If you are considering larger changes its best to discuss them first (e.g. file a bug or enhancement).
Once you’ve made your enhancements open a pull request to get your changes considered for inclusion in the documentation.

Filling Bug Reports

First, search for existing issues to see if the bug has already been reported.
If no bug exists you will need to collect key pieces of information. This information helps us to quickly reproduce (and hopefully fix) the issue:

• What behaviour you expect
  How you think VTR should be working.
• What behaviour you are seeing
  What VTR actually does on your system.
• Detailed steps to re-produce the bug
  This is key to getting your bug fixed.
  Provided detailed steps to reproduce the bug, including the exact commands to reproduce the bug. Attach all relevant files (e.g. FPGA architecture files, benchmark circuits, log files).
  If we can’t re-produce the issue it is very difficult to fix.
• Context about what you are trying to achieve
  Sometimes VTR does things in a different way than you expect. Telling us what you are trying to accomplish helps us to come up with better real-world solutions.
• Details about your environment
  Tell us what version of VTR you are using (e.g. the output of vpr --version), which Operating System and compiler you are using, or any other relevant information about where or how you are building/running VTR.
Once you’ve gathered all the information open an Issue on our issue tracker.
If you know how to fix the issue, or already have it coded-up, please also consider submitting the fix. This is likely the fastest way to get bugs fixed!
Filling Enhancement Requests

First, search existing issues to see if your enhancement request overlaps with an existing Issue. If not feature request exists you will need to describe your enhancement:

- New behaviour
  How your proposed enhancement will work (from a user’s perspective).
- Contrast with current behaviour
  How will your enhancement differ from the current behaviour (from a user’s perspective).
- Potential Implementation
  Describe (if you have some idea) how the proposed enhancement would be implemented.
- Context
  What is the broader goal you are trying to accomplish? How does this enhancement help? This allows us to understand why this enhancement is beneficial, and come up with the best real-world solution.

**VTR developers have limited time and resources, and will not be able to address all feature requests.** Typically, simple enhancements, and those which are broadly useful to a wide group of users get higher priority.

Features which are not generally useful, or useful to only a small group of users will tend to get lower priority. (Of course coding the enhancement yourself is an easy way to bypass this challenge).

Once you’ve gathered all the information open an Issue on our issue tracker.

Submitting Code to VTR

VTR welcomes external contributions.

In general changes that are narrowly focused (e.g. small bug fixes) are easier to review and include in the code base.

Large changes, such as substantial new features or significant code-refactoring are more challenging to review. It is probably best to file an enhancement first to discuss your approach.

Additionally, new features which are generally useful are much easier to justify adding to the code base, whereas features useful in only a few specialized cases are more difficult to justify.

Once your fix/enhancement is ready to go, start a pull request.

Making Pull Requests

It is assumed that by opening a pull request to VTR you have permission to do so, and the changes are under the relevant License. VTR does not require a Contributor License Agreement (CLA) or formal Developer Certificate of Origin (DCO) for contributions.

Each pull request should describe it’s motivation and context (linking to a relevant Issue for non-trivial changes).

Code-changes should also describe:

- The type of change (e.g. bug-fix, feature)
- How it has been tested
- What tests have been added

All new features must have tests added which exercise the new features. This ensures any future changes which break your feature will be detected. It is also best to add tests when fixing bugs, for the same reason.
See *Adding Tests* for details on how to create new regression tests. If you aren’t sure what tests are needed, ask a maintainer.

- **How the feature has been documented**

  Any new user-facing features should be documented in the public documentation, which is in `.rst` format under `doc/src`, and served at https://docs.verilogtorouting.org

Once everything is ready create a pull request.

**Tips for Pull Requests** The following are general tips for making your pull requests easy to review (and hence more likely to be merged):

- **Keep changes small**

  Large change sets are difficult and time-consuming to review. If a change set is becoming too large, consider splitting it into smaller pieces; you’ll probably want to *file an issue* to discuss things first.

- **Do one thing only**

  All the changes and commits in your pull request should be relevant to the bug/feature it addresses. There should be no unrelated changes (e.g. adding IDE files, re-formatting unchanged code).

  Unrelated changes make it difficult to accept a pull request, since it does more than what the pull request described.

- **Match existing code style** When modifying existing code, try match the existing coding style. This helps to keep the code consistent and reduces noise in the pull request (e.g. by avoiding re-formatting changes), which makes it easier to review and more likely to be merged.

### 11.2 Commit Procedures

For general guidance on contributing to VTR see *Submitting Code to VTR*.

The actual machanics of submitting code are outlined below.

However they differ slightly depending on whether you are:

- an **internal developer** (i.e. you have commit access to the main VTR repository at `github.com/verilog-to-routing/vtr-verilog-to-routing`) or,
- an **external developer** (i.e. no commit access).

The overall approach is similar, but we call out the differences below.

1. Setup a local repository on your development machine.

   a. **External Developers**

      - Create a ‘fork’ of the VTR repository.

        Usually this is done on GitHub, giving you a copy of the VTR repository (i.e. `github.com/<username>/vtr-verilog-to-routing`) to which you have commit rights. See *About forks* in the GitHub documentation.

      - Clone your ‘fork’ onto your local machine.

        For example, `git clone git@github.com:<username>/vtr-verilog-to-routing.git`, where `<username>` is your GitHub username.

   b. **Internal Developers**
• Clone the main VTR repository onto your local machine.
  
  For example, `git clone git@github.com:verilog-to-routing/vtr-verilog-to-routing.git`.

2. Move into the cloned repository.
  
  For example, `cd vtr-verilog-to-routing`.

3. Create a `branch`, based off of `master` to work on.
  
  For example, `git checkout -b my_awesome_branch master`, where `my_awesome_branch` is some helpful (and descriptive) name you give your branch. *Please try to pick descriptive branch names!*

4. Make your changes to the VTR code base.

5. Test your changes to ensure they work as intended and have not broken other features.
  
  At the bare minimum it is recommended to run:

  ```
  make
  ./run_reg_test.py vtr_reg_basic vtr_reg_strong  
  #Rebuild the code
  #Run tests
  ```

  See *Running Tests* for more details.

  Also note that additional *code formatting* checks, and tests will be run when you open a Pull Request.

6. Commit your changes (i.e. `git add` followed by `git commit`).

  Please try to use good commit messages!

  See *Commit Messages and Structure* for details.

7. Push the changes to GitHub.

  For example, `git push origin my_awesome_branch`.

a. **External Developers**
   
  Your code changes will now exist in your branch (e.g. `my_awesome_branch`) within your fork (e.g. `github.com/<username>/vtr-verilog-to-routing/tree/my_awesome_branch`, where `<username>` is your GitHub username)

b. **Internal Developers**
   
  Your code changes will now exist in your branch (e.g. `my_awesome_branch`) within the main VTR repository (i.e. `github.com/verilog-to-routing/vtr-verilog-to-routing/tree/my_awesome_branch`)

8. Create a Pull Request (PR) to request your changes be merged into VTR.

   • Navigate to your branch on GitHub
   
     a. **External Developers**
     
        Navigate to your branch within your fork on GitHub (e.g. `https://github.com/<username>/vtr-verilog-to-routing/tree/my_awesome_branch`, where `<username>` is your GitHub username, and `my_awesome_branch` is your branch name).

     b. **Internal Developers**
     
        Navigate to your branch on GitHub (e.g. `https://github.com/verilog-to-routing/vtr-verilog-to-routing/tree/my_awesome_branch`, where `my_awesome_branch` is your branch name).

   • Select the New pull request button.

     a. **External Developers**
If prompted, select `verilog-to-routing/vtr-verilog-to-routing` as the base repository.

11.3 Commit Messages and Structure

11.3.1 Commit Messages

Commit messages are an important part of understanding the code base and its history. It is therefore extremely important to provide the following information in the commit message:

- What is being changed?
- Why is this change occurring?

The diff of changes included with the commit provides the details of what is actually changed, so only a high-level description of what is being done is needed. However, a code diff provides no insight into why the change is being made, so this extremely helpful context can only be encoded in the commit message.

The preferred convention in VTR is to structure commit messages as follows:

```
Header line: explain the commit in one line (use the imperative)

More detailed explanatory text. Explain the problem that this commit
is solving. Focus on why you are making this change as opposed to how
(the code explains that). Are there side effects or other unintuitive
consequences of this change? Here's the place to explain them.

If necessary. Wrap lines at some reasonable point (e.g. 74 characters,
or so) In some contexts, the header line is treated as the subject
of the commit and the rest of the text as the body. The blank line
separating the summary from the body is critical (unless you omit
the body entirely); various tools like `log`, `shortlog` and `rebase`
can get confused if you run the two together.

Further paragraphs come after blank lines.

- Bullet points are okay, too

- Typically a hyphen or asterisk is used for the bullet, preceded
  by a single space, with blank lines in between, but conventions
  vary here

You can also put issue tracker references at the bottom like this:

Fixes: #123
See also: #456, #789
```

(based off of here, and here).

Commit messages do not always need to be long, so use your judgement. More complex or involved changes with wider ranging implications likely deserve longer commit messages than fixing a simple typo.

It is often helpful to phrase the first line of a commit as an imperative/command written as if to tell the repository what to do (e.g. `Update netlist data structure comments. Add tests for feature XYZ. Fix bug which ...`).

To provide quick context, some VTR developers also tag the first line with the main part of the code base effected, some common ones include:
11.3.2 Commit Structure

Generally, you should strive to keep commits atomic (i.e. they do one logical change to the code). This often means keeping commits small and focused in what they change. Of course, a large number of miniscule commits is also unhelpful (overwhelming and difficult to see the structure), and sometimes things can only be done in large changes – so use your judgement. A reasonable rule of thumb is to try and ensure VTR will still compile after each commit.

For those familiar with history re-writing features in git (e.g. rebase) you can sometimes use these to clean-up your commit history after the fact. However these should only be done on private branches, and never directly on master.

11.4 Code Formatting

Some parts of the VTR code base (e.g. VPR, libarchfpga, libvtrutil) have C/C++ code formatting requirements which are checked automatically by regression tests. If your code changes are not compliant with the formatting, you can run:

```bash
make format
```

from the root of the VTR source tree. This will automatically reformat your code to be compliant with formatting requirements (this requires the `clang-format` tool to be available on your system).

Python code must also be compliant with the formatting. To format Python code, you can run:

```bash
make format-py
```

from the root of the VTR source tree (this requires the `black` tool to be available on your system).

11.4.1 Large Scale Reformatting

For large scale reformatting (should only be performed by VTR maintainers) the script `dev/autoformat.py` can be used to reformat the C/C++ code and commit it as ‘VTR Robot’, which keeps the revision history clearer and records metadata about reformatting commits (which allows `git hyper-blame` to skip such commits). The `--python` option can be used for large scale formatting of Python code.
11.4.2 Python Linting

Python files are automatically checked using `pylint` to ensure they follow established Python conventions. You can run `pylint` on the entire repository by running `./dev/pylint_check.py`. Certain files which were created before we adopted Python lint checking are grandfathered and are not checked. To check all files, provide the `--check_grandfathered` argument. You can also manually check individual files using `./dev/pylint_check.py <path_to_file1> <path_to_file2> ....`

11.5 Running Tests

VTR has a variety of tests which are used to check for correctness, performance and Quality of Result (QoR).

There are 4 main regression testing suites:

- **vtr_reg_basic**: ~1 minute serial
  - **Goal**: Fast functionality check
  - **Feature Coverage**: Low
  - **Benchmarks**: A few small and simple circuits
  - **Architectures**: A few simple architectures
  
  This regression test is *not* suitable for evaluating QoR or performance. It’s primary purpose is to make sure the various tools do not crash/fail in the basic VTR flow.

  QoR checks in this regression test are primarily ‘canary’ checks to catch gross degradations in QoR. Occasionally, code changes can cause QoR failures (e.g. due to CAD noise – particularly on small benchmarks); usually such failures are not a concern if the QoR differences are small.

- **vtr_reg_strong**: ~20 minutes serial, ~15 minutes with `-j4`
  - **Goal**: Broad functionality check
  - **Feature Coverage**: High
  - **Benchmarks**: A few small circuits, with some special benchmarks to exercise specific features
  - **Architectures**: A variety of architectures, including special architectures to exercise specific features
  
  This regression test is *not* suitable for evaluating QoR or performance. It’s primary purpose is try and achieve high functionality coverage.

  QoR checks in this regression test are primarily ‘canary’ checks to catch gross degradations in QoR. Occasionally, changes can cause QoR failures (e.g. due to CAD noise – particularly on small benchmarks); usually such failures are not a concern if the QoR differences are small.

- **vtr_reg_nightly_test#**, #:1-3:
  - **Goal**: Basic QoR and Performance evaluation
  - **Feature Coverage**: Medium
  - **Architectures**: A wider variety of architectures
  - **Benchmarks**: Small-medium size, diverse. All include:
  
  - VTR benchmarks
  - Additional benchmarks for each suite.
QoR checks in these regression suites are aimed at evaluating quality and run-time of the VTR flow. As a result any QoR failures are a concern and should be investigated and understood.

Note:
These suites comprise a single large suite, vtr_reg_nightly and should be run together to test nightly level regression. They are mostly similar in benchmark coverage in terms of size and diversity however each suite tests some unique benchmarks in addition to the VTR benchmarks.

<table>
<thead>
<tr>
<th>suite</th>
<th>wall-clock time</th>
<th>Additional benchmarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>vtr_reg_nightly_test1</td>
<td>~4.5 hours</td>
<td>ISPD and MCNC20</td>
</tr>
<tr>
<td>vtr_reg_nightly_test2</td>
<td>~6 hours</td>
<td>Titan23 and Titan other</td>
</tr>
<tr>
<td>vtr_reg_nightly_test3</td>
<td>~5.5 hours</td>
<td>none</td>
</tr>
</tbody>
</table>

• vtr_reg_weekly: ~42 hours with -j4

Goal: Full QoR and Performance evaluation.

Feature Coverage: Medium

Benchmarks: Medium-Large size, diverse. Includes:
- VTR benchmarks
- Titan23 benchmarks

Architectures: A wide variety of architectures

QoR checks in this regression are aimed at evaluating quality and run-time of the VTR flow. As a result any QoR failures are a concern and should be investigated and understood.

These can be run with run_reg_test.py:

```bash
#From the VTR root directory
$ ./run_reg_test.py vtr_reg_basic
$ ./run_reg_test.py vtr_reg_strong
```

The nightly and weekly regressions require the Titan and ISPD benchmarks which can be integrated into your VTR tree with:

```
make get_titan_benchmarks
make get_ispd_benchmarks
```

They can then be run using run_reg_test.py:

```
$ ./run_reg_test.py vtr_reg_nightly_test1
$ ./run_reg_test.py vtr_reg_nightly_test2
$ ./run_reg_test.py vtr_reg_nightly_test3
$ ./run_reg_test.py vtr_reg_weekly
```

To speed-up things up, individual sub-tests can be run in parallel using the -j option:

```
#Run up to 4 tests in parallel
$ ./run_reg_test.py vtr_reg_strong -j4
```

You can also run multiple regression tests together:

```
#Run both the basic and strong regression, with up to 4 tests in parallel
$ ./run_reg_test.py vtr_reg_basic vtr_reg_strong -j4
```

11.5. Running Tests
11.5.1 Running in a large cluster using SLURM

For the very large runs, you can submit your runs on a large cluster. A template of submission script to a Slurm-managed cluster can be found under vtr_flow/tasks/slurm/

11.5.2 Odin Functionality Tests

Odin has its own set of tests to verify the correctness of its synthesis results:

- odin_reg_basic: ~2 minutes serial
- odin_reg_strong: ~6 minutes serial

These can be run with:

```
#From the VTR root directory
$ ./run_reg_test.py odin_reg_basic
$ ./run_reg_test.py odin_reg_strong
```

and should be used when making changes to Odin.

11.5.3 Unit Tests

VTR also has a limited set of unit tests, which can be run with:

```
#From the VTR root directory
$ make && make test
```

11.5.4 Running tests on Pull Requests (PRs) via Kokoro

Because of the long runtime for nightly and weekly tests, a Kokoro job can be used to run these tests once a Pull Request (PR) has been made at https://github.com/verilog-to-routing/vtr-verilog-to-routing.

Any pull request made by a contributor of the verilog-to-routing GitHub project on https://github.com/verilog-to-routing/ will get a set of jobs immediately. Non-contributors can request a contributor on the project add a label “kokoro:force-run” to the PR. Kokoro will then detect the tag, remove the tag, and then and issue jobs for that PR. If the tag remains after being added, there may not be an available Kokoro runner, so wait.

Re-running tests on Kokoro

If a job fails due to an intermittent failure or a re-run is desired, a contributor can add the label “kokoro:force-run” to re-issue jobs for that PR.
Checking results from Kokoro tests

Currently there is not a way for an in-flight job to be monitored.

Once a job has been completed, you can follow the “Details” link that appears on the PR status. The Kokoro page will show the job’s stdout in the ‘Target Log’ tab (once the job has completed). The full log can be downloading by clicking the ‘Download Full Log’ button, or from the ‘Artifacts’ tab.

Downloading logs from Google Cloud Storage (GCS)

After a Kokoro run is complete a number of useful log files (e.g. for each VPR invocation) are stored to Google Cloud Storage (GCS).

The top level directory containing all VTR Kokoro runs is:

```
https://console.cloud.google.com/storage/browser/vtr-verilog-to-routing/artifacts/prod/
  →foss-fpga-tools/verilog-to-routing/upstream/
```

PR jobs are under the `presubmit` directory, and continuous jobs (which run on the master branch) are under the `continuous` directory.

Each Kokoro run has a unique build number, which can be found in the log file (available via the Kokoro run webpage). For example, if the log file contains:

```
export KOKORO_BUILD_NUMBER="450"
```

then the Kokoro build number is 450.

If build 450 corresponded to a PR (presubmit) build of the nightly regression tests, the resulting output files would be available at:

```
https://console.cloud.google.com/storage/browser/vtr-verilog-to-routing/artifacts/prod/
  →foss-fpga-tools/verilog-to-routing/upstream/presubmit/nightly/450/
```

where `presubmit/nightly/450/` (the type, test name and build number) have been appended to the base URL. Navigating to that URL will allow you to browse and download the collected log files.

To download all the files from that Kokoro run, replace `https://console.cloud.google.com/storage/browser/` in the URL with `gs://`/ and invoke the gsutil command (and it’s `cp` `-R` sub-command), like so:

```
gsutil -m cp -R gs://vtr-verilog-to-routing/artifacts/prod/foss-fpga-tools/verilog-to-
  →routing/upstream/presubmit/nightly/450 .
```

This will download all of the logs to the current directory for inspection.

Kokoro runner details

Kokoro runners are a standard `n1-highmem-16` VM with a 4 TB pd-standard disk used to perform the build of VPR and run the tests.
What to do if Kokoro jobs are not starting?

There are several reasons Kokoro jobs might not be starting. Try adding the “kokoro:force-run” label if it is not already added, or remove and add it if it already was added.

If adding the label has no affect, check GCS status, as a GCS disruption will also disrupt Kokoro.

Another reason jobs may not start is if there is a large backlog of jobs running, there may be no runners left to start. In this case, someone with Kokoro management rights may need to terminate stale jobs, or wait for job timeouts.

11.6 Debugging Failed Tests

If a test fails you probably want to look at the log files to determine the cause.

Let’s assume we have a failure in vtr_reg_basic:

```
# In the VTR root directory
$ ./run_reg_test.py vtr_reg_strong
# Output trimmed...
regression_tests/vtr_reg_basic/basic_no_timing
-----------------------------
k4_N10_memSize16384_memData64/ch_intrinsics/common failed: vpr
k4_N10_memSize16384_memData64/diffeq1/common failed: vpr
# Output trimmed...
regression_tests/vtr_reg_basic/basic_no_timing...[Fail]
k4_N10_memSize16384_memData64.xml/ch_intrinsics.v vpr_status: golden = success result = exited
# Output trimmed...
Error: 10 tests failed!
```

Here we can see that vpr failed, which caused subsequent QoR failures ([Fail]), and resulted in 10 total errors.

To see the log files we need to find the run directory. We can see from the output that the specific test which failed was regression_tests/vtr_reg_basic/basic_no_timing. All the regression tests take place under vtr_flow/tasks, so the test directory is vtr_flow/tasks/regression_tests/vtr_reg_basic/basic_no_timing. Let’s move to that directory:

```
# From the VTR root directory
$ cd vtr_flow/tasks/regression_tests/vtr_reg_basic/basic_no_timing
$ ls
config run001 run003
latest run002 run004 run005
```

There we see there is a config directory (which defines the test), and a set of run-directories. Each time a test is run it creates a new runXXX directory (where XXX is an incrementing number). From the above we can tell that our last run was run005 (the symbolic link latest also points to the most recent run directory). From the output of run_reg_test.py we know that one of the failing architecture/circuit/parameters combinations was k4_N10_memSize16384_memData64/ch_intrinsics/common. Each architecture/circuit/parameter combination is run in its own sub-folder. Let’s move to that directory:

```
$ cd run005/k4_N10_memSize16384_memData64/ch_intrinsics/common
$ ls
abc.out k4_N10_memSize16384_memData64.xml qor_results.txt
ch_intrinsics.net odin.out thread_1.out
```

(continues on next page)
Here we can see the individual log files produced by each tool (e.g. vpr.out), which we can use to guide our debugging. We could also manually re-run the tools (e.g. with a debugger) using files in this directory.

### 11.7 Evaluating Quality of Result (QoR) Changes

VTR uses highly tuned and optimized algorithms and data structures. Changes which effect these can have significant impacts on the quality of VTR’s design implementations (timing, area etc.) and VTR’s run-time/memory usage. Such changes need to be evaluated carefully before they are pushed/merged to ensure no quality degradation occurs.

If you are unsure of what level of QoR evaluation is necessary for your changes, please ask a VTR developer for guidance.

#### 11.7.1 General QoR Evaluation Principles

The goal of performing a QoR evaluation is to measure precisely the impact of a set of code/architecture/benchmark changes on both the quality of VTR’s design implementation (i.e. the result of VTR’s optimizations), and on tool run-time and memory usage.

This process is made more challenging by the fact that many of VTR’s optimization algorithms are based on heuristics (some of which depend on randomization). This means that VTR’s implementation results are dependent upon:

- The initial conditions (e.g. input architecture & netlist, random number generator seed), and
- The precise optimization algorithms used.

The result is that a minor change to either of these can can make the measured QoR change. This effect can be viewed as an intrinsic ‘noise’ or ‘variance’ to any QoR measurement for a particular architecture/benchmark/algorithm combination.

There are typically two key methods used to measure the ‘true’ QoR:

1. Averaging metrics across multiple architectures and benchmark circuits.
2. Averaging metrics multiple runs of the same architecture and benchmark, but using different random number generator seeds

   This is a further variance reduction technique, although it can be very CPU-time intensive. A typical example would be to sweep an entire benchmark set across 3 or 5 different seeds.

In practice any algorithm changes will likely cause improvements on some architecture/benchmark combinations, and degradations on others. As a result we primarily focus on the average behaviour of a change to evaluate its impact. However extreme outlier behaviour on particular circuits is also important, since it may indicate bugs or other unexpected behaviour.
Key QoR Metrics

The following are key QoR metrics which should be used to evaluate the impact of changes in VTR.

Implementation Quality Metrics:

<table>
<thead>
<tr>
<th>Metric</th>
<th>Meaning</th>
<th>Sensitivity</th>
</tr>
</thead>
<tbody>
<tr>
<td>num_pre_packed_blocks</td>
<td>Number of primitive netlist blocks (after tech. mapping, before packing)</td>
<td>Low</td>
</tr>
<tr>
<td>num_post_packed_blocks</td>
<td>Number of Clustered Blocks (after packing)</td>
<td>Medium</td>
</tr>
<tr>
<td>device_grid_tiles</td>
<td>FPGA size in grid tiles</td>
<td>Low-Medium</td>
</tr>
<tr>
<td>min_chan_width</td>
<td>The minimum routable channel width</td>
<td>Medium*</td>
</tr>
<tr>
<td>crit_path_routed_wirelength</td>
<td>The routed wirelength at the relaxed channel width</td>
<td>Medium</td>
</tr>
<tr>
<td>critical_path_delay</td>
<td>The critical path delay at the relaxed channel width</td>
<td>Medium-High</td>
</tr>
</tbody>
</table>

* By default, VPR attempts to find the minimum routable channel width; it then performs routing at a relaxed (e.g. 1.3x minimum) channel width. At minimum channel width routing congestion can distort the true timing/wirelength characteristics. Combined with the fact that most FPGA architectures are built with an abundance of routing, post-routing metrics are usually only evaluated at the relaxed channel width.

Run-time/Memory Usage Metrics:

<table>
<thead>
<tr>
<th>Metric</th>
<th>Meaning</th>
<th>Sensitivity</th>
</tr>
</thead>
<tbody>
<tr>
<td>vtr_flow_elapsed_time</td>
<td>Wall-clock time to complete the VTR flow</td>
<td>Low</td>
</tr>
<tr>
<td>pack_time</td>
<td>Wall-clock time VPR spent during packing</td>
<td>Low</td>
</tr>
<tr>
<td>place_time</td>
<td>Wall-clock time VPR spent during placement</td>
<td>Low</td>
</tr>
<tr>
<td>min_chan_width_route_time</td>
<td>Wall-clock time VPR spent during routing at the minimum routable channel width</td>
<td>High*</td>
</tr>
<tr>
<td>crit_path_route_time</td>
<td>Wall-clock time VPR spent during routing at the relaxed channel width</td>
<td>Low</td>
</tr>
<tr>
<td>max_vpr_mem</td>
<td>Maximum memory used by VPR (in kilobytes)</td>
<td>Low</td>
</tr>
</tbody>
</table>

* Note that the minimum channel width route time is chaotic and can be highly variable (e.g. 10x variation is not unusual). Minimum channel width routing performs a binary search to find the minimum channel width. Since route time is highly dependent on congestion, run-time is highly dependent on the precise channel widths searched (which may change due to perturbations).

In practice you will likely want to consider additional and more detailed metrics, particularly those directly related to the changes you are making. For example, if your change related to hold-time optimization you would want to include hold-time related metrics such as hold_TNS (hold total negative slack) and hold_WNS (hold worst negative slack). If your change related to packing, you would want to report additional packing-related metrics, such as the number of clusters formed by each block type (e.g. numbers of CLBs, RAMs, DSPs, IOs).
**Benchmark Selection**

An important factor in performing any QoR evaluation is the benchmark set selected. In order to draw reasonably general conclusions about the impact of a change we desire two characteristics of the benchmark set:

1. It includes a large number of benchmarks which are representative of the application domains of interest.
   
   This ensures we don’t over-tune to a specific benchmark or application domain.

2. It should include benchmarks of large sizes.

   This ensures we can optimize and scale to large problem spaces.

In practice (for various reasons) satisfying both of these goals simultaneously is challenging. The key goal here is to ensure the benchmark set is not unreasonably biased in some manner (e.g. benchmarks which are too small, benchmarks too skewed to a particular application domain).

**Fairly measuring tool run-time**

Accurately and fairly measuring the run-time of computer programs is challenging in practice. A variety of factors effect run-time including:

- Operating System
- System load (e.g. other programs running)
- Variance in hardware performance (e.g. different CPUs on different machines, CPU frequency scaling)

To make reasonably ‘fair’ run-time comparisons it is important to isolate the change as much as possible from other factors. This involves keeping as much of the experimental environment identical as possible including:

1. Target benchmarks
2. Target architecture
3. Code base (e.g. VTR revision)
4. CAD parameters
5. Computer system (e.g. CPU model, CPU frequency/power scaling, OS version)
6. Compiler version

**11.7.2 Collecting QoR Measurements**

The first step is to collect QoR metrics on your selected benchmark set.

You need at least two sets of QoR measurements:

1. The baseline QoR (i.e. unmodified VTR).
2. The modified QoR (i.e. VTR with your changes).

Note that it is important to generate both sets of QoR measurements on the same computing infrastructure to ensure a fair run-time comparison.

The following examples show how a single set of QoR measurements can be produced using the VTR flow infrastruc-
Example: VTR Benchmarks QoR Measurement

The VTR benchmarks are a group of benchmark circuits distributed with the VTR project. They are provided as syne-
thizable verilog and can be re-mapped to VTR supported architectures. They consist mostly of small to medium sized
circuits from a mix of application domains. They are used primarily to evaluate the VTR’s optimization quality in an
architecture exploration/evaluation setting (e.g. determining minimum channel widths).

A typical approach to evaluating an algorithm change would be to run vtr_reg_qor_chain task from the nightly regression test:

```bash
#From the VTR root
$ cd vtr_flow/tasks

#Run the VTR benchmarks
$ ../scripts/run_vtr_task.py regression_tests/vtr_reg_nightly_test3/vtr_reg_qor_chain

#Several hours later... they complete

#Parse the results
$ ../scripts/python_libs/vtr/parse_vtr_task.py regression_tests/vtr_reg_nightly_test3/

#The run directory should now contain a summary parse_results.txt file
$ head -5 vtr_reg_nightly_test3/vtr_reg_qor_chain/latest/parse_results.txt
```

(continues on next page)
### 11.7. Evaluating Quality of Result (QoR) Changes

<table>
<thead>
<tr>
<th>File Path</th>
<th>Blob Merge V</th>
<th>Common</th>
</tr>
</thead>
<tbody>
<tr>
<td>k6_frac_N10_frac_chain_mem32K_40nm.xml</td>
<td>blob_merge.v</td>
<td>common</td>
</tr>
<tr>
<td>k6_frac_N10_frac_chain_mem32K_40nm.xml</td>
<td>boundtop.v</td>
<td>common</td>
</tr>
<tr>
<td>k6_frac_N10_frac_chain_mem32K_40nm.xml</td>
<td>ch_intrinsics.v</td>
<td>common</td>
</tr>
</tbody>
</table>
Example: Titan Benchmarks QoR Measurements

The Titan benchmarks are a group of large benchmark circuits from a wide range of applications, which are compatible with the VTR project. They are typically used as post-technology mapped netlists which have been pre-synthesized with Quartus. They are substantially larger and more realistic than the VTR benchmarks, but can only target specifically compatible architectures. They are used primarily to evaluate the optimization quality and scalability of VTR’s CAD algorithms while targeting a fixed architecture (e.g. at a fixed channel width).

A typical approach to evaluating an algorithm change would be to run `titan_quick_qor` task from the nightly regression test:

Running and Integrating the Titan Benchmarks with VTR

```
#From the VTR root

#Download and integrate the Titan benchmarks into the VTR source tree
$ make get_titan_benchmarks

#Move to the task directory
$ cd vtr_flow/tasks

#Run the VTR benchmarks
$ ../scripts/run_vtr_task.py regression_tests/vtr_reg_nightly_test2/titan_quick_qor

#Several days later... they complete

#Parse the results
$ ../scripts/python_libs/vtr/parse_vtr_task.py regression_tests/vtr_reg_nightly_test2/titan_quick_qor

#The run directory should now contain a summary parse_results.txt file
$ head -5 vtr_reg_nightly_test2/titan_quick_qor/latest/parse_results.txt
```

```
arch  circuit  vpr_revision  vpr_status  error  num_pre_packed_nets  num_pre_ packed_blocks  num_post_packed_nets  num_post_packed_blocks  device_ width  device_height  num_clb  num_outputs  num_memories  num_mult  placed_wirelength_est  placed_CPD_est  placed_setup_TNS_est  placed_setup_WNS_est  routed_wirelength  crit_path  route_success_iteration  logic_block_area_total  logic_block_area_used  routing_area_total  routing_area_per_tile  critical_path_delay  setup_TNS  setup_WNS  hold_TNS  hold_WNS  pack_time  place_time  crit_path_route_time  max_vpr_mem  max_odin_mem  max_abc_mem
stratixiv_arch.timing.xml  neuron_stratixiv_arch_timing.blif  0208312
  →  success  119888  86875
  →  51408  3370  128
  →  95  -1  42  35  -1
  →  -8.70971  1086419  20
  →  -1  0  0  2
  → -66512e+08  21917.1  9.64877
  → -262034 -9.64877  0  0  127.92  218.48
  →  259.96  5133800  -1  -1
(continues on next page)
```
Example: Koios Benchmarks QoR Measurement

The Koios benchmarks are a group of Deep Learning benchmark circuits distributed with the VTR project. They are provided as synthesizable verilog and can be re-mapped to VTR supported architectures. They consist mostly of medium to large sized circuits from Deep Learning (DL). They can be used for FPGA architecture exploration for DL and also for tuning CAD tools.

A typical approach to evaluating an algorithm change would be to run koios (or koios_no_complex_dsp) task from the nightly regression test (vtr_reg_nightly_test4) and the koios (or koios_no_complex_dsp) task from the weekly regression test (vtr_reg_weekly). The nightly test contains smaller benchmarks, whereas the large designs are in the weekly regression test. To measure QoR for the entire benchmark suite, both nightly and weekly tests should be run and the results should be concatenated.

The koios regression task runs these benchmarks with complex_dsp functionality enabled, whereas koios_no_complex_dsp regression task runs these benchmarks without complex_dsp functionality. Normally, only the koios tasks should be enough for QoR.

The following steps show a sequence of commands to run the koios tasks on the Koios benchmarks from both nightly and weekly regressions:
# From the VTR root
$ cd vtr_flow/tasks

# Run the VTR benchmarks
$ ../scripts/run_vtr_task.py regression_tests/vtr_reg_nightly_test4/koios &
$ ../scripts/run_vtr_task.py regression_tests/vtr_reg_weekly/koios &

# Several hours later... they complete

# Parse the results
$ ../scripts/python_libs/vtr/parse_vtr_task.py regression_tests/vtr_reg_nightly_test4/koios
$ ../scripts/python_libs/vtr/parse_vtr_task.py regression_tests/vtr_reg_weekly/koios

# The run directory should now contain a summary parse_results.txt file
$ head -5 vtr_reg_nightly_test4/koios/<latest_run_dir>/parse_results.txt

<table>
<thead>
<tr>
<th>arch</th>
<th>circuit</th>
<th>script_params</th>
<th>vtr_flow_elapsed_time</th>
<th>error</th>
<th>odin_synth_time</th>
<th>max_odin_mem</th>
<th>abc_depth</th>
<th>abc_mem</th>
<th>abc_synthesis_time</th>
<th>max_abc_mem</th>
<th>abc_memories</th>
<th>num_clb</th>
<th>num_io</th>
<th>num_abc_synthesis_time</th>
<th>max_num_abc_mem</th>
<th>ace_time</th>
<th>num_ace_mem</th>
<th>num_ace_mem</th>
<th>num_ace_mem</th>
<th>num_ace_mem</th>
<th>num_ace_mem</th>
<th>num_ace_mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

k6FracN10LB_mem20K_complexDSP_customSB_22nm.xml  tpu_like.small.v  common 2871.10 9.36 235096 5

(continues on next page)
<table>
<thead>
<tr>
<th>k6FracN10LB_mem20K_complexDSP_customSB_22nm.xml</th>
<th>dla_like.small.v</th>
</tr>
</thead>
<tbody>
<tr>
<td>common</td>
<td></td>
</tr>
<tr>
<td>7527.41</td>
<td>42.24</td>
</tr>
<tr>
<td>729876</td>
<td>5</td>
</tr>
<tr>
<td>-3941.31</td>
<td>-1</td>
</tr>
<tr>
<td>-630244</td>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>5545</td>
<td>194</td>
</tr>
<tr>
<td>828</td>
<td>-1</td>
</tr>
<tr>
<td>success</td>
<td>v8.0.0-4161-</td>
</tr>
<tr>
<td>g8f4b3e9ca</td>
<td>release IPO VTR_ASSERT_LEVEL=2</td>
</tr>
<tr>
<td>GNU 7.5.0 on Linux-4.15.0-</td>
<td></td>
</tr>
<tr>
<td>-124-generic x86_64</td>
<td>2021-05-28T23:09:34</td>
</tr>
<tr>
<td>jupiter0</td>
<td>/export/aman/vtr_verilog-to-routing/vtr_flow/tasks</td>
</tr>
<tr>
<td>4409476</td>
<td></td>
</tr>
<tr>
<td>194</td>
<td>13</td>
</tr>
<tr>
<td>217044</td>
<td>174718</td>
</tr>
<tr>
<td>1</td>
<td>91037</td>
</tr>
<tr>
<td>6708</td>
<td>164</td>
</tr>
<tr>
<td>164</td>
<td>26896</td>
</tr>
<tr>
<td>memory auto</td>
<td></td>
</tr>
<tr>
<td>1604.22</td>
<td>969627</td>
</tr>
<tr>
<td>663.41</td>
<td>2.84</td>
</tr>
<tr>
<td>5.61569</td>
<td></td>
</tr>
<tr>
<td>-424718</td>
<td>-5.61569</td>
</tr>
<tr>
<td>5.61569</td>
<td>21.49</td>
</tr>
<tr>
<td>0.584073</td>
<td>0.</td>
</tr>
<tr>
<td>385993</td>
<td>104.796</td>
</tr>
<tr>
<td>73.1698</td>
<td>1450542</td>
</tr>
<tr>
<td>14</td>
<td>8.6211e+08</td>
</tr>
<tr>
<td>3.01197e+08</td>
<td>5.93540e+08</td>
</tr>
<tr>
<td>22068.</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>53.97</td>
</tr>
<tr>
<td>132.203</td>
<td>96.049</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>14</td>
<td></td>
</tr>
<tr>
<td>k6FracN10LB_mem20K_complexDSP_customSB_22nm.xml</td>
<td>bnn.v</td>
</tr>
<tr>
<td>common</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2028.52</td>
</tr>
<tr>
<td>40.37</td>
<td>577472</td>
</tr>
<tr>
<td>3</td>
<td>240.94</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>513656</td>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<td>5695</td>
</tr>
<tr>
<td>260</td>
<td>0</td>
</tr>
<tr>
<td>-1</td>
<td>success v8.0.0-4161-g8f4b3e9ca</td>
</tr>
<tr>
<td>release IPO VTR_ASSERT_LEVEL=2</td>
<td></td>
</tr>
<tr>
<td>GNU 7.5.0 on Linux-4.15.0-</td>
<td></td>
</tr>
<tr>
<td>-124-generic x86_64</td>
<td>2021-05-28T23:09:34</td>
</tr>
<tr>
<td>jupiter0</td>
<td>/export/aman/vtr_verilog-to-routing/vtr_flow/tasks</td>
</tr>
<tr>
<td>2195980</td>
<td>260</td>
</tr>
<tr>
<td>122</td>
<td></td>
</tr>
<tr>
<td>-231647</td>
<td>179602</td>
</tr>
<tr>
<td>1</td>
<td>86181</td>
</tr>
<tr>
<td>6140</td>
<td>83</td>
</tr>
<tr>
<td>83</td>
<td>6889</td>
</tr>
<tr>
<td>clb</td>
<td>auto</td>
</tr>
<tr>
<td>613.32</td>
<td>940951</td>
</tr>
<tr>
<td>-503.35</td>
<td>2.87</td>
</tr>
<tr>
<td>6.4402</td>
<td>-131403</td>
</tr>
<tr>
<td>-6.4402</td>
<td>6.</td>
</tr>
<tr>
<td>4402</td>
<td>5.41</td>
</tr>
<tr>
<td>0.753268</td>
<td>0.564332</td>
</tr>
<tr>
<td>85.331</td>
<td>60</td>
</tr>
<tr>
<td>8639</td>
<td>-1</td>
</tr>
<tr>
<td>1224690</td>
<td>16</td>
</tr>
<tr>
<td>2.13666e+08</td>
<td>1.</td>
</tr>
<tr>
<td>74902e+08</td>
<td>1.51359e+08</td>
</tr>
<tr>
<td>21971.1</td>
<td>50.49</td>
</tr>
<tr>
<td>114</td>
<td></td>
</tr>
<tr>
<td>382</td>
<td>84.8538</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>k6FracN10LB_mem20K_complexDSP_customSB_22nm.xml</td>
<td>attention_layer.v</td>
</tr>
<tr>
<td>common</td>
<td></td>
</tr>
<tr>
<td>1330.99</td>
<td>11.83</td>
</tr>
<tr>
<td>1095592</td>
<td>7</td>
</tr>
<tr>
<td>-59.16</td>
<td>-1</td>
</tr>
<tr>
<td>-506012</td>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>-1248</td>
<td>1058</td>
</tr>
<tr>
<td>161</td>
<td>-1</td>
</tr>
<tr>
<td>success</td>
<td>v8.0.0-4161-</td>
</tr>
<tr>
<td>g8f4b3e9ca</td>
<td>release IPO VTR_ASSERT_LEVEL=2</td>
</tr>
<tr>
<td>GNU 7.5.0 on Linux-4.15.0-</td>
<td></td>
</tr>
<tr>
<td>-124-generic x86_64</td>
<td>2021-05-28T23:09:34</td>
</tr>
<tr>
<td>jupiter0</td>
<td>/export/aman/vtr_verilog-to-routing/vtr_flow/tasks</td>
</tr>
<tr>
<td>1180420</td>
<td></td>
</tr>
<tr>
<td>-1058</td>
<td>16</td>
</tr>
<tr>
<td>47407</td>
<td>39134</td>
</tr>
<tr>
<td>1</td>
<td>26605</td>
</tr>
<tr>
<td>2588</td>
<td>86</td>
</tr>
<tr>
<td>86</td>
<td>7396</td>
</tr>
<tr>
<td>dsp_top</td>
<td>auto</td>
</tr>
<tr>
<td>728.</td>
<td></td>
</tr>
<tr>
<td>70</td>
<td>234151</td>
</tr>
<tr>
<td>118.11</td>
<td>0.71</td>
</tr>
<tr>
<td>5.89837</td>
<td>78343.</td>
</tr>
<tr>
<td>6</td>
<td>-5.89837</td>
</tr>
<tr>
<td>5.89837</td>
<td>6.64</td>
</tr>
<tr>
<td>0.181478</td>
<td>0.</td>
</tr>
<tr>
<td>146942</td>
<td>31.9659</td>
</tr>
<tr>
<td>24.5807</td>
<td>366899</td>
</tr>
<tr>
<td>-17</td>
<td>2.32446e+08</td>
</tr>
<tr>
<td>8.36361e+07</td>
<td>1.62201e+08</td>
</tr>
<tr>
<td>21930.</td>
<td></td>
</tr>
<tr>
<td>-9</td>
<td>16.25</td>
</tr>
<tr>
<td>40.6352</td>
<td>32.1556</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

$ head -5 vtr_reg_weekly/koios/<latest_run_dir>/parse_results.txt
<table>
<thead>
<tr>
<th>k6FracN10LB_mem20K_complexDSP_customSB_22nm.xml clstm_like.small.v</th>
<th>common 19316.21</th>
</tr>
</thead>
<tbody>
<tr>
<td>162.86 2395612 3</td>
<td>15651.23 -1 -1 1337084 -1</td>
</tr>
<tr>
<td>-1 9309 293 407 -1</td>
<td>success v8.0.0-4470-ge625fdfe9 release IPO</td>
</tr>
<tr>
<td>VTR_ASSERT_LEVEL=2 GNU 7.5.0 on Linux-4.15.0-124-generic x86_64 2021-06-18T14:02:07 jupiter0 /export/aman/vtr_aman/vtr-verilog-to-routing/vtr_flow/</td>
<td></td>
</tr>
<tr>
<td>tasks 4793656 293 33664 1</td>
<td>99618 10661 152 152</td>
</tr>
<tr>
<td>-1 23104 dsp_top auto</td>
<td>780.11 1482046 931.17 6.18 6.93257-474705 -6.93257 6.93257 27.96 1.30491 1.10886 193.621</td>
</tr>
<tr>
<td>-1 23104 dsp_top auto</td>
<td>780.11 1482046 931.17 6.18 6.93257-474705 -6.93257 6.93257 27.96 1.30491 1.10886 193.621</td>
</tr>
<tr>
<td>-1 23104 dsp_top auto</td>
<td>780.11 1482046 931.17 6.18 6.93257-474705 -6.93257 6.93257 27.96 1.30491 1.10886 193.621</td>
</tr>
<tr>
<td>-1 23104 dsp_top auto</td>
<td>780.11 1482046 931.17 6.18 6.93257-474705 -6.93257 6.93257 27.96 1.30491 1.10886 193.621</td>
</tr>
<tr>
<td>-1 23104 dsp_top auto</td>
<td>780.11 1482046 931.17 6.18 6.93257-474705 -6.93257 6.93257 27.96 1.30491 1.10886 193.621</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>k6FracN10LB_mem20K_complexDSP_customSB_22nm.xml clstm_like.medium.v</th>
<th>common 66156.51</th>
</tr>
</thead>
<tbody>
<tr>
<td>483.84 4490632 3</td>
<td>60586.75 -1 -1 2480684 -1</td>
</tr>
<tr>
<td>-1 17641 293 784 -1</td>
<td>success v8.0.0-4470-ge625fdfe9 release IPO</td>
</tr>
<tr>
<td>VTR_ASSERT_LEVEL=2 GNU 7.5.0 on Linux-4.15.0-124-generic x86_64 2021-06-18T14:02:07 jupiter0 /export/aman/vtr_aman/vtr-verilog-to-routing/vtr_flow/</td>
<td></td>
</tr>
<tr>
<td>tasks 8819072 293 578 696105 624609 1</td>
<td>181497 19958 206 206</td>
</tr>
<tr>
<td>-1 42436 dsp_top auto</td>
<td>1057.61 3181171 1581.42 7.89 8.18417-1.24303e+06 -8.18417 8.18417 38.05 1.40496 0.993779 233.67 213.509</td>
</tr>
<tr>
<td>-1 42436 dsp_top auto</td>
<td>1057.61 3181171 1581.42 7.89 8.18417-1.24303e+06 -8.18417 8.18417 38.05 1.40496 0.993779 233.67 213.509</td>
</tr>
<tr>
<td>-1 42436 dsp_top auto</td>
<td>1057.61 3181171 1581.42 7.89 8.18417-1.24303e+06 -8.18417 8.18417 38.05 1.40496 0.993779 233.67 213.509</td>
</tr>
<tr>
<td>-1 42436 dsp_top auto</td>
<td>1057.61 3181171 1581.42 7.89 8.18417-1.24303e+06 -8.18417 8.18417 38.05 1.40496 0.993779 233.67 213.509</td>
</tr>
<tr>
<td>-1 42436 dsp_top auto</td>
<td>1057.61 3181171 1581.42 7.89 8.18417-1.24303e+06 -8.18417 8.18417 38.05 1.40496 0.993779 233.67 213.509</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>k6FracN10LB_mem20K_complexDSP_customSB_22nm.xml clstm_like.large.v</th>
<th>common 101695.32</th>
</tr>
</thead>
<tbody>
<tr>
<td>717.44 654768 3</td>
<td>94597.42 -1 -1 3590044 -1</td>
</tr>
<tr>
<td>-1 25995 293 1161 -1</td>
<td>success v8.0.0-4470-ge625fdfe9 release IPO</td>
</tr>
<tr>
<td>VTR_ASSERT_LEVEL=2 GNU 7.5.0 on Linux-4.15.0-124-generic x86_64 2021-06-18T14:02:07 jupiter0 /export/aman/vtr_aman/vtr-verilog-to-routing/vtr_flow/</td>
<td></td>
</tr>
<tr>
<td>tasks 13040176 293 866 1018158 915547 1</td>
<td>263328 29277 248 248</td>
</tr>
<tr>
<td>-1 61504 dsp_top auto</td>
<td>1503.62 5185716 1951.96 10.18 8.86387-1.97516e+06 -8.86387 1.36407e+09 7.68183e+08 9.34572e+08</td>
</tr>
<tr>
<td>-1 61504 dsp_top auto</td>
<td>1503.62 5185716 1951.96 10.18 8.86387-1.97516e+06 -8.86387 1.36407e+09 7.68183e+08 9.34572e+08</td>
</tr>
<tr>
<td>-1 61504 dsp_top auto</td>
<td>1503.62 5185716 1951.96 10.18 8.86387-1.97516e+06 -8.86387 1.36407e+09 7.68183e+08 9.34572e+08</td>
</tr>
</tbody>
</table>

Chapter 11. Developer Guide
11.7.3 Comparing QoR Measurements

Once you have two (or more) sets of QoR measurements they now need to be compared.

A general method is as follows:

1. Normalize all metrics to the values in the baseline measurements (this makes the relative changes easy to evaluate)
2. Produce tables for each set of QoR measurements showing the per-benchmark relative values for each metric
3. Calculate the GEOMEAN over all benchmarks for each normalized metric
4. Produce a summary table showing the Metric Geomeans for each set of QoR measurements

QoR Comparison Gotchas

There are a variety of ‘gotchas’ you need to avoid to ensure fair comparisons:

- GEOMEAN’s must be over the same set of benchmarks. A common issue is that a benchmark failed to complete for some reason, and it’s metric values are missing
- Run-times need to be collected on the same compute infrastructure at the same system load (ideally unloaded).

Example QoR Comparison

Suppose we’ve make a change to VTR, and we now want to evaluate the change. As described above we produce QoR measurements for both the VTR baseline, and our modified version.

We then have the following (hypothetical) QoR Metrics.

Baseline QoR Metrics:
<table>
<thead>
<tr>
<th>arch</th>
<th>circuit</th>
<th>num_pre_packed_blocks</th>
<th>num_post_packed_blocks</th>
<th>device_grid_tiles</th>
<th>min_chan_width</th>
<th>crit_path_routed_wirelength</th>
<th>crit_path_delay</th>
<th>vtr_flow_elapsed_time</th>
<th>pack_time</th>
<th>place_time</th>
<th>min_chan_width_route_time</th>
<th>crit_path_route_time</th>
<th>max_vpr_mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>k6_frac_N10</td>
<td>top.v</td>
<td>11407</td>
<td>389</td>
<td>900</td>
<td>64</td>
<td>75615</td>
<td>141.53</td>
<td>11009.08</td>
<td>45.47</td>
<td>1410872</td>
<td>5.99</td>
<td>19.94</td>
<td>84</td>
</tr>
<tr>
<td>k6_frac_N10</td>
<td>blob_merge.v</td>
<td>1141</td>
<td>389</td>
<td>169</td>
<td>34</td>
<td>3767</td>
<td>2.55</td>
<td>12514.85</td>
<td>19.94</td>
<td>85760</td>
<td>5.99</td>
<td>19.94</td>
<td>84</td>
</tr>
<tr>
<td>k6_frac_N10</td>
<td>bound_top.v</td>
<td>1141</td>
<td>389</td>
<td>169</td>
<td>34</td>
<td>3767</td>
<td>2.55</td>
<td>12514.85</td>
<td>19.94</td>
<td>85760</td>
<td>5.99</td>
<td>19.94</td>
<td>84</td>
</tr>
<tr>
<td>k6_frac_N10</td>
<td>ch_intrinsics.v</td>
<td>493</td>
<td>247</td>
<td>100</td>
<td>46</td>
<td>1438</td>
<td>2.45</td>
<td>12514.85</td>
<td>19.94</td>
<td>85760</td>
<td>5.99</td>
<td>19.94</td>
<td>84</td>
</tr>
<tr>
<td>k6_frac_N10</td>
<td>diffeq1.v</td>
<td>886</td>
<td>313</td>
<td>256</td>
<td>60</td>
<td>9624</td>
<td>17.96</td>
<td>12514.85</td>
<td>19.94</td>
<td>85760</td>
<td>5.99</td>
<td>19.94</td>
<td>84</td>
</tr>
<tr>
<td>k6_frac_N10</td>
<td>diffeq2.v</td>
<td>599</td>
<td>201</td>
<td>256</td>
<td>52</td>
<td>8928</td>
<td>13.70</td>
<td>12514.85</td>
<td>19.94</td>
<td>85760</td>
<td>5.99</td>
<td>19.94</td>
<td>84</td>
</tr>
<tr>
<td>k6_frac_N10</td>
<td>LU8PEEng.v</td>
<td>31396</td>
<td>2286</td>
<td>2916</td>
<td>100</td>
<td>348085</td>
<td>79.45</td>
<td>12514.85</td>
<td>19.94</td>
<td>85760</td>
<td>5.99</td>
<td>19.94</td>
<td>84</td>
</tr>
<tr>
<td>k6_frac_N10</td>
<td>LU32PEEng.v</td>
<td>101542</td>
<td>7251</td>
<td>9216</td>
<td>158</td>
<td>1554942</td>
<td>80.06</td>
<td>12514.85</td>
<td>19.94</td>
<td>85760</td>
<td>5.99</td>
<td>19.94</td>
<td>84</td>
</tr>
<tr>
<td>k6_frac_N10</td>
<td>mcml.v</td>
<td>165809</td>
<td>6767</td>
<td>8649</td>
<td>128</td>
<td>1311825</td>
<td>51.19</td>
<td>12514.85</td>
<td>19.94</td>
<td>85760</td>
<td>5.99</td>
<td>19.94</td>
<td>84</td>
</tr>
<tr>
<td>k6_frac_N10</td>
<td>mkDelayWorker32B.v</td>
<td>4145</td>
<td>1327</td>
<td>2500</td>
<td>38</td>
<td>30086</td>
<td>8.39</td>
<td>12514.85</td>
<td>19.94</td>
<td>85760</td>
<td>5.99</td>
<td>19.94</td>
<td>84</td>
</tr>
<tr>
<td>k6_frac_N10</td>
<td>mkPktMerge.v</td>
<td>1160</td>
<td>516</td>
<td>784</td>
<td>44</td>
<td>13370</td>
<td>4.44</td>
<td>12514.85</td>
<td>19.94</td>
<td>85760</td>
<td>5.99</td>
<td>19.94</td>
<td>84</td>
</tr>
<tr>
<td>k6_frac_N10</td>
<td>mkSMAdapter4B.v</td>
<td>2852</td>
<td>548</td>
<td>400</td>
<td>48</td>
<td>19274</td>
<td>5.27</td>
<td>12514.85</td>
<td>19.94</td>
<td>85760</td>
<td>5.99</td>
<td>19.94</td>
<td>84</td>
</tr>
<tr>
<td>k6_frac_N10</td>
<td>or1200.v</td>
<td>4530</td>
<td>1321</td>
<td>729</td>
<td>62</td>
<td>51633</td>
<td>9.67</td>
<td>12514.85</td>
<td>19.94</td>
<td>85760</td>
<td>5.99</td>
<td>19.94</td>
<td>84</td>
</tr>
<tr>
<td>k6_frac_N10</td>
<td>raygen_top.v</td>
<td>2934</td>
<td>710</td>
<td>361</td>
<td>58</td>
<td>22045</td>
<td>5.15</td>
<td>12514.85</td>
<td>19.94</td>
<td>85760</td>
<td>5.99</td>
<td>19.94</td>
<td>84</td>
</tr>
<tr>
<td>k6_frac_N10</td>
<td>sha.v</td>
<td>3024</td>
<td>236</td>
<td>289</td>
<td>62</td>
<td>16653</td>
<td>10.01</td>
<td>12514.85</td>
<td>19.94</td>
<td>85760</td>
<td>5.99</td>
<td>19.94</td>
<td>84</td>
</tr>
<tr>
<td>k6_frac_N10</td>
<td>stereovision0.v</td>
<td>21801</td>
<td>1122</td>
<td>1156</td>
<td>58</td>
<td>64935</td>
<td>3.63</td>
<td>12514.85</td>
<td>19.94</td>
<td>85760</td>
<td>5.99</td>
<td>19.94</td>
<td>84</td>
</tr>
<tr>
<td>k6_frac_N10</td>
<td>stereovision1.v</td>
<td>19538</td>
<td>1096</td>
<td>1600</td>
<td>100</td>
<td>143517</td>
<td>5.62</td>
<td>12514.85</td>
<td>19.94</td>
<td>85760</td>
<td>5.99</td>
<td>19.94</td>
<td>84</td>
</tr>
<tr>
<td>k6_frac_N10</td>
<td>stereovision2.v</td>
<td>42078</td>
<td>2534</td>
<td>7396</td>
<td>134</td>
<td>650583</td>
<td>15.32</td>
<td>12514.85</td>
<td>19.94</td>
<td>85760</td>
<td>5.99</td>
<td>19.94</td>
<td>84</td>
</tr>
<tr>
<td>k6_frac_N10</td>
<td>stereovision3.v</td>
<td>324</td>
<td>55</td>
<td>49</td>
<td>30</td>
<td>768</td>
<td>2.66</td>
<td>12514.85</td>
<td>19.94</td>
<td>85760</td>
<td>5.99</td>
<td>19.94</td>
<td>84</td>
</tr>
</tbody>
</table>

Modified QoR Metrics:
Based on these metrics we then calculate the following ratios and summary.

**QoR Metric Ratio** (Modified QoR / Baseline QoR):
<table>
<thead>
<tr>
<th>arch</th>
<th>circuit</th>
<th>num_pre_packed_blocks</th>
<th>num_post_packed_blocks</th>
<th>num_device_grid_tiles</th>
<th>max_vpr_mem</th>
<th>k6_frac_N10_frac_chain_mem32K_40nm.xml</th>
<th>bgm.v</th>
<th>1.00</th>
<th>0.97</th>
<th>1.00</th>
<th>0.98</th>
<th>1.05</th>
<th>1.39</th>
<th>1.04</th>
<th>0.92</th>
</tr>
</thead>
<tbody>
<tr>
<td>k6_frac_N10</td>
<td>objmac</td>
<td>chab0_men32K_40kshil</td>
<td>0.98</td>
<td>1.02</td>
<td>1.00</td>
<td>0.98</td>
<td>0.50</td>
<td>1.05</td>
<td>1.39</td>
<td>1.04</td>
<td>0.92</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>k6_frac_N10</td>
<td>objmac</td>
<td>chab0_men32K_40kshil</td>
<td>1.02</td>
<td>0.96</td>
<td>0.90</td>
<td>0.51</td>
<td>1.18</td>
<td>1.22</td>
<td>0.96</td>
<td>0.91</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>k6_frac_N10</td>
<td>objmac</td>
<td>chab0_men32K_40kshil</td>
<td>0.92</td>
<td>0.89</td>
<td>0.62</td>
<td>0.44</td>
<td>0.85</td>
<td>0.43</td>
<td>1.13</td>
<td>0.95</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>k6_frac_N10</td>
<td>objmac</td>
<td>chab0_men32K_40kshil</td>
<td>0.99</td>
<td>1.02</td>
<td>0.68</td>
<td>0.41</td>
<td>0.87</td>
<td>0.46</td>
<td>1.00</td>
<td>0.97</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>k6_frac_N10</td>
<td>objmac</td>
<td>chab0_men32K_40kshil</td>
<td>1.04</td>
<td>0.96</td>
<td>0.98</td>
<td>0.28</td>
<td>0.71</td>
<td>1.13</td>
<td>1.55</td>
<td>0.83</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>k6_frac_N10</td>
<td>objmac</td>
<td>chab0_men32K_40kshil</td>
<td>0.85</td>
<td>0.96</td>
<td>1.08</td>
<td>0.45</td>
<td>1.20</td>
<td>1.20</td>
<td>0.83</td>
<td>0.80</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>k6_frac_N10</td>
<td>objmac</td>
<td>chab0_men32K_40kshil</td>
<td>1.00</td>
<td>0.98</td>
<td>0.84</td>
<td>0.50</td>
<td>1.00</td>
<td>0.84</td>
<td>1.08</td>
<td>0.94</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>k6_frac_N10</td>
<td>objmac</td>
<td>chab0_men32K_40kshil</td>
<td>1.09</td>
<td>1.00</td>
<td>1.01</td>
<td>0.49</td>
<td>0.96</td>
<td>1.02</td>
<td>1.11</td>
<td>0.91</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>k6_frac_N10</td>
<td>objmac</td>
<td>chab0_men32K_40kshil</td>
<td>0.95</td>
<td>0.89</td>
<td>1.03</td>
<td>0.57</td>
<td>0.92</td>
<td>1.20</td>
<td>0.78</td>
<td>0.87</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>k6_frac_N10</td>
<td>objmac</td>
<td>chab0_men32K_40kshil</td>
<td>1.11</td>
<td>1.00</td>
<td>0.82</td>
<td>0.50</td>
<td>0.96</td>
<td>0.75</td>
<td>0.91</td>
<td>0.98</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>k6_frac_N10</td>
<td>objmac</td>
<td>chab0_men32K_40kshil</td>
<td>0.82</td>
<td>1.04</td>
<td>1.03</td>
<td>0.95</td>
<td>0.33</td>
<td>0.92</td>
<td>1.08</td>
<td>0.96</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>k6_frac_N10</td>
<td>objmac</td>
<td>chab0_men32K_40kshil</td>
<td>1.03</td>
<td>0.99</td>
<td>0.58</td>
<td>0.31</td>
<td>0.64</td>
<td>0.73</td>
<td>0.97</td>
<td>0.89</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>k6_frac_N10</td>
<td>objmac</td>
<td>chab0_men32K_40kshil</td>
<td>0.93</td>
<td>0.90</td>
<td>1.91</td>
<td>0.30</td>
<td>0.64</td>
<td>3.83</td>
<td>0.86</td>
<td>0.81</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>k6_frac_N10</td>
<td>objmac</td>
<td>chab0_men32K_40kshil</td>
<td>0.94</td>
<td>0.98</td>
<td>0.57</td>
<td>0.29</td>
<td>0.60</td>
<td>0.65</td>
<td>0.70</td>
<td>0.86</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>k6_frac_N10</td>
<td>objmac</td>
<td>chab0_men32K_40kshil</td>
<td>0.96</td>
<td>1.05</td>
<td>0.86</td>
<td>0.46</td>
<td>0.83</td>
<td>0.73</td>
<td>0.92</td>
<td>0.90</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>k6_frac_N10</td>
<td>objmac</td>
<td>chab0_men32K_40kshil</td>
<td>1.08</td>
<td>1.00</td>
<td>1.05</td>
<td>0.46</td>
<td>0.97</td>
<td>1.71</td>
<td>1.00</td>
<td>0.91</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>k6_frac_N10</td>
<td>objmac</td>
<td>chab0_men32K_40kshil</td>
<td>1.00</td>
<td>1.07</td>
<td>1.26</td>
<td>0.40</td>
<td>0.89</td>
<td>1.66</td>
<td>0.75</td>
<td>0.71</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>k6_frac_N10</td>
<td>objmac</td>
<td>chab0_men32K_40kshil</td>
<td>0.99</td>
<td>0.96</td>
<td>1.53</td>
<td>0.52</td>
<td>0.90</td>
<td>1.59</td>
<td>0.99</td>
<td>0.86</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>k6_frac_N10</td>
<td>objmac</td>
<td>chab0_men32K_40kshil</td>
<td>1.20</td>
<td>0.95</td>
<td>0.69</td>
<td>0.41</td>
<td>0.70</td>
<td>0.75</td>
<td>1.00</td>
<td>1.04</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GE-MEAN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>QoR Summary:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

QoR Summary:
From the results we can see that our change, on average, achieved a small reduction in the number of logic blocks (0.95) in return for a 2% increase in minimum channel width and 1% increase in routed wirelength. From a run-time perspective the packer is substantially faster (0.42).

### Automated QoR Comparison Script

To automate some of the QoR comparison VTR includes a script to compare `parse_results.txt` files and generate a spreadsheet including the ratio and summary tables.

For example:

```
# From the VTR Root
$ ./vtr_flow/scripts/qor_compare.py parse_results1.txt parse_results2.txt parse_results3.txt -o comparison.xlsx
```

will produce ratio tables and a summary table for the files `parse_results1.txt`, `parse_results2.txt` and `parse_results3.txt`, where the first file (parse_results1.txt) is assumed to be the baseline used to produce normalized ratios.

### Generating New QoR Golden Result

There may be times when a regression test fails its QoR test because its golden_result needs to be changed due to known changes in code behaviour. In this case, a new golden result needs to be generated so that the test can be passed. To generate a new golden result, follow the steps outlined below.

1. Move to the `vtr_flow/tasks` directory from the VTR root, and run the failing test. For example, if a test called `vtr_ex_test` in `vtr_reg_nightly_test3` was failing:

```
# From the VTR root
$ cd vtr_flow/tasks
$ ../scripts/run_vtr_task.py regression_tests/vtr_reg_nightly_test3/vtr_ex_test
```

2. Next, generate new golden reference results using `parse_vtr_task.py` and the `-create_golden` option.

```
$ ../scripts/python_libs/vtr/parse_vtr_task.py regression_tests/vtr_reg_nightly_test3/vtr_ex_test -create_golden
```

3. Lastly, check that the results match with the `-check_golden` option.
Once the `-check_golden` command passes, the changes to the golden result can be committed so that the reg test will pass in future runs of `vtr_reg_nightly_test3`.

### 11.8 Adding Tests

Any time you add a feature to VTR you **must** add a test which exercises the feature. This ensures that regression tests will detect if the feature breaks in the future.

Consider which regression test suite your test should be added to (see *Running Tests* descriptions).

Typically, test which exercise new features should be added to `vtr_reg_strong`. These tests should use small benchmarks to ensure they:

- run quickly (so they get run often!), and
- are easier to debug. If your test will take more than ~1 minute it should probably go in a longer running regression test (but see first if you can create a smaller testcase first).

#### 11.8.1 Adding a test to `vtr_reg_strong`

This describes adding a test to `vtr_reg_strong`, but the process is similar for the other regression tests.

1. Create a configuration file

   First move to the `vtr_reg_strong` directory:

   ```bash
   # From the VTR root directory
   $ cd vtr_flow/tasks/regression_tests/vtr_reg_strong
   $ ls
   qor_geomean.txt  strong_flyover_wires  strong_pack_and_place
   strong_analysis_only  strong_fpu_hard_block_arch  strong_power
   strong_bounding_box  strong_frupturable_luts  strong_route_only
   strong_breadth_first  strong_func_formal_flow  strong_scale_delay_budgets
   strong_constant_outputs  strong_func_formal_vpr  strong_sweep_constant_outputs
   strong_custom_grid  strong_global_routing  strong_timing
   strong_custom_pin_locs  strong_manual_anneing  strong_titan
   strong_custom_switch_block  strong_mnc  strong_valgrind
   strong_echo_files  strong_minimax_budgets  strong_verify_rr_graph
   strong_fc_abs  strong_multiclock  task_list.txt
   strong_fix_pins_pad_file  strong_no_timing  task_summary
   strong_fix_pins_random  strong_pack
   ```

   Each folder (prefixed with `strong_` in this case) defines a task (sub-test).

   Let's make a new task named `strong_mytest`. An easy way is to copy an existing configuration file such as `strong_timing/config/config.txt`

   ```bash
   $ mkdir -p strong_mytest/config
   $ cp strong_timing/config/config.txt strong_mytest/config/.
   ```

   You can now edit `strong_mytest/config/config.txt` to customize your test.
2. Generate golden reference results

Now we need to test our new test and generate ‘golden’ reference results. These will be used to compare future runs of our test to detect any changes in behaviour (e.g. bugs).

From the VTR root, we move to the `vtr_flow/tasks` directory, and then run our new test:

```bash
# From the VTR root directory
$ cd vtr_flow/tasks
$ ../scripts/run_vtr_task.py regression_tests/vtr_reg_strong/strong_mytest
```

```
regression_tests/vtr_reg_strong/strong_mytest
-----------------------------------------
Current time: Jan-25 06:51 PM. Expected runtime of next benchmark: Unknown
k6_frac_N10_mem32K_40nm/ch_intrinsics...OK
```

Next we can generate the golden reference results using `parse_vtr_task.py` with the `-create_golden` option:

```bash
$ ../scripts/python_libs/vtr/parse_vtr_task.py regression_tests/vtr_reg_strong/
  → strong_mytest -create_golden
```

And check that everything matches with `-check_golden`:

```bash
$ ../scripts/python_libs/vtr/parse_vtr_task.py regression_tests/vtr_reg_strong/
  → strong_mytest -check_golden
regression_tests/vtr_reg_strong/strong_mytest...[Pass]
```

3. Add it to the task list

We now need to add our new `strong_mytest` task to the task list, so it is run whenever `vtr_reg_strong` is run. We do this by adding the line `regression_tests/vtr_reg_strong/strong_mytest` to the end of `vtr_reg_strong`'s `task_list.txt`:

```bash
# From the VTR root directory
$ vim vtr_flow/tasks/regression_tests/vtr_reg_strong/task_list.txt
# Add a new line 'regression_tests/vtr_reg_strong/strong_mytest' to the end of the file
```

Now, when we run `vtr_reg_strong`:

```bash
# From the VTR root directory
$ ./run_reg_test.py vtr_reg_strong
# Output trimmed...
regression_tests/vtr_reg_strong/strong_mytest
-----------------------------------------
# Output trimmed...
```

we see our test is run.

4. Commit the new test

Finally you need to commit your test:

```bash
# Add the config.txt and golden_results.txt for the test
$ git add vtr_flow/tasks/regression_tests/vtr_reg_strong/strong_mytest/
# Add the change to the task_list.txt
$ git add vtr_flow/tasks/regression_tests/vtr_reg_strong/task_list.txt
```

(continues on next page)
11.9 Debugging Aids

VTR has support for several additional tools/features to aid debugging.

11.9.1 Sanitizers

VTR can be compiled using sanitizers which will detect invalid memory accesses, memory leaks and undefined behaviour (supported by both GCC and LLVM):

```
# From the VTR root directory
$ cmake -D VTR_ENABLE_SANITIZE=ON build
$ make
```

You can suppress reporting of known memory leaks in libraries used by vpr by setting the environment variable below:

```
LSAN_OPTIONS=suppressions=$VTR_ROOT/vpr/lsan.supp
```

where $VTR_ROOT is the root directory of your vtr source code tree.

Note that some of the continuous integration (CI) regtests (run automatically on pull requests) turn on sanitizers (currently S: Basic and R: Odin-II Basic Tests)

11.9.2 Valgrind

An alternative way to run vtr programs to check for invalid memory accesses and memory leaks is to use the valgrind tool. valgrind can be run on any build except the sanitized build, without recompilation. For example, to run on vpr use

```
# From the VTR root directory
valgrind --leak-check=full --suppressions=./vpr/valgrind.supp ./vpr/vpr [... usual vpr... --options here ...]
```

The suppression file included in the command above will suppress reporting of known memory leaks in libraries included by vpr.

Note that valgrind is run on some flows by the continuous integration (CI) tests.
11.9.3 Assertion Levels

VTR supports configurable assertion levels.

The default level (2) which turns on most assertions which don’t cause significant run-time penalties.

This level can be increased:

```bash
# From the VTR root directory
$ cmake -D VTR_ASSERT_LEVEL=3 build
$ make
```

this turns on more extensive assertion checking and re-builds VTR.

11.9.4 GDB Pretty Printers

To make it easier to debug some of VTR’s data structures with GDB.

STL Pretty Printers

It is helpful to enable STL pretty printers, which make it much easier to debug data structures using STL.

For example printing a `std::vector<int>` by default prints:

```bash
(gdb) p/r x_locs
$2 = {
  _M_impl = {
    _M_finish = 0x555556f063dc, _M_end_of_storage = 0x555556f064b0}, <No data fields>
}
```

which is not very helpful.

But with STL pretty printers it prints:

```bash
(gdb) p x_locs
$2 = std::vector of length 11, capacity 64 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

which is much more helpful for debugging!

If STL pretty printers aren’t already enabled on your system, add the following to your `.gdbinit` file:

```python
import sys
sys.path.insert(0, "$STL_PRINTER_ROOT")
from libstdcxx.v6.printers import register_libstdcxx_printers
register_libstdcxx_printers(None)
end
```

where `$STL_PRINTER_ROOT` should be replaced with the appropriate path to the STL pretty printers. For example recent versions of GCC include these under `/usr/share/gcc-**/python` (e.g. `/usr/share/gcc-9/python`)
**VTR Pretty Printers**

VTR includes some pretty printers for some VPR/VTR specific types.

For example, without the pretty printers you would see the following when printing a VPR `AtomBlockId`:

```
(gdb) p blk_id
$1 = {
   id_  = 71
}
```

But with the VTR pretty printers enabled you would see:

```
(gdb) p blk_id
$1 = AtomBlockId(71)
```

To enable the VTR pretty printers in GDB add the following to your `.gdbinit` file:

```
python
import sys
sys.path.insert(0, "$VTR_ROOT/dev")
import vtr_gdb_pretty_printers
vtr_gdb_pretty_printers.vtr_type_lookup
end
```

where `$VTR_ROOT` should be replaced with the root of the VTR source tree on your system.

**11.9.5 RR (Record Replay) Debugger**

RR extends GDB with the ability to record a run of a tool and then re-run it to reproduce any observed issues. RR also enables efficient reverse execution (!) which can be extremely helpful when tracking down the source of a bug.

**11.10 Speeding up the edit-compile-test cycle**

Rapid iteration through the edit-compile-test/debug cycle is very helpful when making code changes to VTR. The following is some guidance on techniques to reduce the time required.

**11.11 Speeding Compilation**

1. Parallel compilation

   For instance when building VTR using make, you can specify the `-j N` option to compile the code base with `N` parallel jobs:

   ```
   $ make -j N
   ```

   A reasonable value for `N` is equal to the number of threads your system can run. For instance, if your system has 4 cores with HyperThreading (i.e. 2-way SMT) you could run:
$ make -j8

2. Building only a subset of VTR

If you know your changes only effect a specific tool in VTR, you can request that only that tool is rebuilt. For instance, if you only wanted to re-compile VPR you could run:

$ make vpr

which would avoid re-building other tools (e.g. ODIN, ABC).

3. Use ccache

ccache is a program which caches previous compilation results. This can save significant time, for instance, when switching back and forth between release and debug builds.

VTR’s cmake configuration should automatically detect and make use of ccache once it is installed.

For instance on Ubuntu/Debian systems you can install ccache with:

$ sudo apt install ccache

This only needs to be done once on your development system.

4. Disable Interprocedural Optimizations (IPO)

IPO re-optimizes an entire executable at link time, and is automatically enabled by VTR if a supporting compiler is found. This can notably improve performance (e.g. ~10-20% faster), but can significantly increase compilation time (e.g. >2x in some cases). When frequently re-compiling and debugging the extra execution speed may not be worth the longer compilation times. In such cases you can manually disable IPO by setting the cmake parameter VTR_IPO_BUILD=off.

For instance using the wrapper Makefile:

$ make CMAKE_PARAMS="-DVTR_IPO_BUILD=off"

Note that this option is sticky, so subsequent calls to make don’t need to keep specifying VTR_IPO_BUILD, until you want to re-enable it.

This setting can also be changed with the ccmake tool (i.e. ccmake build).

All of these option can be used in combination. For example, the following will re-build only VPR using 8 parallel jobs with IPO disabled:

make CMAKE_PARAMS="-DVTR_IPO_BUILD=off" -j8 vpr

11.12 Profiling VTR

1. Install gprof, gprof2dot, and xdot. Specifically, the previous two packages require python3, and you should install the last one with sudo apt install for all the dependencies you will need for visualizing your profile results.

pip3 install gprof
pip3 install gprof2dot
sudo apt install xdot

Contact your administrator if you do not have the sudo rights.
2. Use the CMake option below to enable VPR profiler build.

```
make CMAKE_PARAMS="-DVTR_ENABLE_PROFILING=ON" vpr
```

3. With the profiler build, each time you run the VTR flow script, it will produce an extra file `gmon.out` that contains the raw profile information. Run `gprof` to parse this file. You will need to specify the path to the VPR executable.

```
gprof $VTR_ROOT/vpr/vpr gmon.out > gprof.txt
```

4. Next, use `gprof2dot` to transform the parsed results to a `.dot` file, which describes the graph of your final profile results. If you encounter long function names, specify the `-s` option for a cleaner graph.

```
gprof2dot -s gprof.txt > vpr.dot
```

5. You can chain the above commands to directly produce the `.dot` file:

```
gprof $VTR_ROOT/vpr/vpr gmon.out | gprof2dot -s > vpr.dot
```

6. Use `xdot` to view your results:

```
xdot vpr.dot
```

7. To save your results as a `png` file:

```
dot -Tpng -Gdpi=300 vpr.dot > vpr.png
```

Note that you can use the `-Gdpi` option to make your picture clearer if you find the default dpi settings not clear enough.

### 11.13 External Subtrees

VTR includes some code which is developed in external repositories, and is integrated into the VTR source tree using `git` subtrees.

To simplify the process of working with subtrees we use the `dev/external_subtrees.py` script.

For instance, running `./dev/external_subtrees.py --list` from the VTR root it shows the subtrees:

```
Component: abc Path: abc URL: https://github.com/berkeley-abc/abc.git URL_Ref: master
Component: libargparse Path: libs/EXTERNAL/libargparse URL: https://github.com/kmurray/libargparse.git URL_Ref: master
Component: libblifparse Path: libs/EXTERNAL/libblifparse URL: https://github.com/kmurray/libblifparse.git URL_Ref: master
Component: libsdcparse Path: libs/EXTERNAL/libsdcparse URL: https://github.com/kmurray/libsdcparse.git URL_Ref: master
Component: libtatum Path: libs/EXTERNAL/libtatum URL: https://github.com/kmurray/tatum.git URL_Ref: master
```

Code included in VTR by subtrees should not be modified within the VTR source tree. Instead changes should be made in the relevant up-stream repository, and then synced into the VTR tree.
11.13.1 Updating an existing Subtree

1. From the VTR root run: `./dev/external_subtrees.py $SUBTREE_NAME`, where $SUBTREE_NAME is the name of an existing subtree.

   For example to update the libtatum subtree:

   ```
   ./dev/external_subtrees.py --update libtatum
   ```

11.13.2 Adding a new Subtree

To add a new external subtree to VTR do the following:

1. Add the subtree specification to `dev/subtree_config.xml`.

   For example to add a subtree name `libfoo` from the master branch of `https://github.com/kmurray/libfoo.git` to `libs/EXTERNAL/libfoo` you would add:

   ```xml
   <subtree
     name="libfoo"
     internal_path="libs/EXTERNAL/libfoo"
     external_url="https://github.com/kmurray/libfoo.git"
     default_external_ref="master"/>
   ```

   within the existing `<subtrees>` tag.

   Note that the internal_path directory should not already exist.

   You can confirm it works by running: `dev/external_subtrees.py --list`:

   | Component: abc | Path: abc | URL: https://github. |
   | --com/berkeley-abc/abc.git | URL_Ref: master |
   | Component: libargparse | Path: libs/EXTERNAL/libargparse | URL: https://github. |
   | --com/kmurray/libargparse.git | URL_Ref: master |
   | Component: libblifparse | Path: libs/EXTERNAL/libblifparse | URL: https://github. |
   | --com/kmurray/libblifparse.git | URL_Ref: master |
   | Component: libsdcparse | Path: libs/EXTERNAL/libsdcparse | URL: https://github. |
   | --com/kmurray/libsdcparse.git | URL_Ref: master |
   | Component: libtatum | Path: libs/EXTERNAL/libtatum | URL: https://github. |
   | --com/kmurray/tatum.git | URL_Ref: master |
   | Component: libfoo | Path: libs/EXTERNAL/libfoo | URL: https://github. |
   | --com/kmurray/libfoo.git | URL_Ref: master |

   which shows libfoo is now recognized.

2. Run `./dev/external_subtrees.py --update $SUBTREE_NAME` to add the subtree.

   For the `libfoo` example above this would be:

   ```
   ./dev/external_subtrees.py --update libfoo
   ```

   This will create two commits to the repository. The first will squash all the upstream changes, the second will merge those changes into the current branch.
11.13.3 Subtree Rational

VTR uses subtrees to allow easy tracking of upstream dependencies.

Their main advantages included:

- Works out-of-the-box: no actions needed post checkout to pull in dependencies (e.g. no `git submodule update --init --recursive`)
- Simplified upstream version tracking
- Potential for local changes (although in VTR we do not use this to make keeping in sync easier)

See here for a more detailed discussion.

11.14 Finding Bugs with Coverity

Coverity Scan is a static code analysis service which can be used to detect bugs.

11.14.1 Browsing Defects

To view defects detected do the following:

1. Get a coverity scan account
   - Contact a project maintainer for an invitation.
2. Browse the existing defects through the coverity web interface

11.14.2 Submitting a build

To submit a build to coverity do the following:

1. Download the coverity build tool
2. Configure VTR to perform a `debug` build. This ensures that all assertions are enabled, without assertions coverity may report bugs that are guarded against by assertions. We also set VTR asserts to the highest level.

   ```
   #From the VTR root
   mkdir -p build
   cd build
   CC=gcc CXX=g++ cmake -DCMAKE_BUILD_TYPE=debug -DVTR_ASSERT_LEVEL=3 ..
   ```

   Note that we explicitly asked for gcc and g++, the coverity build tool defaults to these compilers, and may not like the default ‘cc’ or ‘c++’ (even if they are linked to gcc/g++).

3. Run the coverity build tool

   ```
   #From the build directory where we ran cmake
   cov-build --dir cov-int make -j8
   ```

4. Archive the output directory

   ```
   tar -czvf vtr_coverity.tar.gz cov-int
   ```

5. Submit the archive through the coverity web interface
Once the build has been analyzed you can browse the latest results through the coverity web interface

### 11.14.3 No files emitted

If you get the following warning from cov-build:

```shell
[WARNING] No files were emitted.
```

You may need to configure coverity to ‘know’ about your compiler. For example:

```shell
cov-configure --compiler `which gcc-7`
```

On unix-like systems run `scan-build make` from the root VTR directory. To output the html analysis to a specific folder, run `scan-build make -o /some/folder`

### 11.15 Release Procedures

#### 11.15.1 General Principles

We periodically make ‘official’ VTR releases. While we aim to keep the VTR master branch stable through-out development some users prefer to work of off an official release. Historically this has coincided with the publishing of a paper detailing and carefully evaluating the changes from the previous VTR release. This is particularly helpful for giving academics a named baseline version of VTR to which they can compare which has a known quality.

In preparation for a release it may make sense to produce ‘release candidates’ which when fully tested and evaluated (and after any bug fixes) become the official release.

#### 11.15.2 Checklist

The following outlines the procedure to following when making an official VTR release:

- Check the code compiles on the list of supported compilers
- Check that all regression tests pass functionality
- Update regression test golden results to match the released version
- Check that all regression tests pass QoR
- Create a new entry in the CHANGELOG.md for the release, summarizing at a high-level user-facing changes
- Increment the version number (set in root CMakeLists.txt)
- Create a git annotated tag (e.g. v8.0.0) and push it to github
- GitHub will automatically create a release based on the tag
- Add the new change log entry to the GitHub release description
- Update the ReadTheDocs configuration to build and serve documentation for the relevant tag (e.g. v8.0.0)
- Send a release announcement email to the vtr-announce mailing list (make sure to thank all contributors!)
11.16 Sphinx API Documentation for C/C++ Projects

The Sphinx API documentation for VTR C/C++ projects is created using Doxygen and Breathe plugin. Doxygen is a standard tool for generating documentation from annotated code. It is used to generate XML output that can then be parsed by the Breathe plugin, which provides the RST directives used to embed the code comments into the Sphinx documentation.

Every VPR C/C++ project requires a few steps that have to be completed, to generate the Sphinx documentation:

- Create doxyfile for the project
- Update the Breathe config
- Create RST files with the API description using Breathe directives
- Generate the project documentation

11.16.1 Create Doxyfile

A doxyfile a Doxygen configuration file that provides all the necessary information about the documented project. It is used to generate Doxygen output in the chosen format.

The configuration includes the specification of input files, output directory, generated documentation formats, and much more. The config for a particular VPR project should be saved in the `<vtr-verilog-to-routing>/doc/_doxygen` directory. The doxyfile should be named as `<key>.dox`, where `<key>` is a `breathe_projects` dictionary key associated with the VPR project.

The minimal doxyfile should contain only the configuration values that are not set by default. As mentioned before, the Breathe plugin expects the XML input. Therefore the `GENERATE_XML` option should be turned on. Below there is a content of `vpr.dox` file content, which contains the VPR Doxygen configuration:

```doxygen
PROJECT_NAME = "Verilog to Routing - VPR"
OUTPUT_DIRECTORY = ../_build/doxygen/vpr
FULL_PATH_NAMES = NO
OPTIMIZE_OUTPUT_FOR_C = YES
EXTRACT_ALL = YES
EXTRACT_PRIVATE = YES
EXTRACT_STATIC = YES
WARN_IF_UNDOCUMENTED = NO
INPUT = ../../vpr
RECURSIVE = YES
GENERATE_HTML = NO
GENERATE_LATEX = NO
GENERATE_XML = YES
```

The general Doxyfile template can be generated using:

```bash
doxygen -g template.dox
```
11.16.2 Breathe Configuration

Breathe plugin is responsible for parsing the XML file generated by the Doxygen. It provides the convenient RST directives that allow to embed the read documentation into the Sphinx documentation.

To add the new project to the Sphinx API generation mechanism, you have to update the `breathe_projects` dictionary in the Sphinx `conf.py` file. The dictionary consists of key-value pairs which describe the project. The key is related to the project name that will be used in the Breathe plugin directives. The value associated with the key points to the directory where the XML Doxygen output is generated.

Example of this configuration structure is presented below:

```
breathe_projects = {
    "vpr" : "../_build/doxygen/vpr/xml",
    "abc" : "../_build/doxygen/abc/xml",
    "ace2" : "../_build/doxygen/ace2/xml",
    "ODIN_II" : "../_build/doxygen/ODIN_II/xml",
    "blifexplorer" : "../_build/doxygen/blifexplorer/xml",
}
```

More information about the Breathe plugin can be found in the Breathe Documentation.

11.16.3 Create RST with API Documentation

To generate the Sphinx API documentation, you should use the directives provided by the Breathe plugin. A complete list of Breathe directives can be found in the Directives & Config Variables section of the Breathe Documentation.

Example of `doxygenclass` directive used for the VPR project is presented below:

```
.. doxygenclass:: VprContext
   :project: vpr
   :members:
```

11.16.4 Generate the Documentation

Currently, the Doxygen is set up to run automatically whenever the documentation is regenerated. The Doxygen XML generation is skipped when the Doxygen is not installed on your machine or when the `SKIP_DOXYGEN=True` environment variable is set.

The Doxygen is being run for every project described in the `breathe_projects` dictionary. Therefore it is essential to keep the same name of the project name key and the doxyfile name.

11.17 Documenting VTR Code with Doxygen

VTR uses Doxygen and Sphinx for generating code documentation. Doxygen is used to parse a codebase, extract code comments, and save them into an XML file. The XML is then read by the Sphinx Breathe plugin, which converts it to an HTML available publicly in the project documentation. The documentation generated with Sphinx can be found in the API Reference section.

This note presents how to document source code in the VTR project and check whether Doxygen can parse the created description. Code conventions presented below were chosen arbitrarily for the project, from many more options available in Doxygen. To read more about the tool, visit the official Doxygen documentation.
11.17.1 Documenting Code

There are three basic types of Doxygen code comments used in the VTR documentation:

- block comments
- one-line comments before a code element
- one-line comments after an element member

In most cases, a piece of documentation should be placed before a code element. Comments after an element should be used only for documenting members of enumeration types, structures, and unions.

**Block Comments**

You should use Doxygen block comments with both brief and detailed descriptions to document code by default. As the name suggests, a brief description should be a one-liner with general information about the code element. A detailed description provides more specific information about the element, its usage, or implementation details. In the case of functions and methods, information about parameters and returned value comes after the detailed description. Note that brief and detailed descriptions have to be separated with at least one empty line.

Here is an example of a Doxygen block comment:

```c
/**
 * @brief This is a brief function description
 * This is a detailed description. It should be separated from
 * the brief description with one blank line.
 * @param a A description of a
 * @param b A description of b
 * @return A return value description
 */
int my_func(int a, int b) {
    return a + b;
}
```

General guidelines for using Doxygen block comments:

1. A block-comment block **has to** start with the /**, otherwise Doxygen will not recognize it. All the comment lines **have to** be preceded by an asterisk. All the asterisks **have to** create a straight vertical line.

2. Brief and detailed descriptions **have to** be separated with one empty line.

3. A detailed description and a parameter list **should be** separated with one empty line.

4. A parameter list **should be** indented one level. All the parameter descriptions should be aligned together.

5. A returned value description **should be** separated with one empty line from either a detailed or a parameter description.
One-line Comments Before an Element

One-line comments can be used instead of the block comments described above, only if a brief description is sufficient for documenting the particular code element. Usually, this is the case with global variables and defines.

Here is an example of a one-line Doxygen comment (before a code element):

```cpp
/// @brief This is one-line documentation comment
int var = 0;
```

General guidelines for using Doxygen one-line comments (before a code element):

1. A one-line comment before an element **has to** start with ///, otherwise Doxygen will not recognize it.
2. Since this style of code comments **should be** used only for brief descriptions, it **should** contain a @brief tag.
3. One-line comments **should not** be overused. They are acceptable for single variables and defines, but more complicated elements like classes and structures should be documented more carefully with Doxygen block comments.

One-line Comments After an Element Member

There is another type of one-line code comments used to document members of enumeration types, structures, and unions. In those situations, the whole element should be documented in a standard way using a Doxygen block comment. However, the particular element members should be described after they appear in the code with the one-line comments.

Here is an example of a one-line Doxygen comment (after an element member):

```cpp
/**
 * @brief This is a brief enum description
 *
 * This is a detailed description. It should be separated from
 * the brief description with one blank line
 */
enum seasons {
    spring = 3, ///< Describes spring enum value
    summer,   ///< Describes summer enum value
    autumn = 7, ///< Describes autumn enum value
    winter    ///< Describes winter enum value
};
```

General guidelines for using Doxygen one-line comments (after an element member):

1. One-line code comment after an element member **has to** start with ///<, otherwise Doxygen will not recognize it.
2. This comment style **should be** used together with a Doxygen block comment for describing the whole element, before the members’ description.
11.17.2 Documenting Files

All files that contain the source code should be documented with a Doxygen-style header. The file description in Doxygen is similar to code element description, and should be placed at the beginning of the file. The comment should contain information about an author, date of the document creation, and a description of functionalities introduced in the file.

Here is an example of file documentation:

```c
/**
 * @file
 * @author John Doe
 * @date 2020-09-03
 * @brief This is a brief document description
 *
 * This is a detailed description. It should be separated from
 * the brief description with one blank line
 */
```

General suggestions about a Doxygen file comments:

1. A file comment has to start with the @file tag, otherwise it will not be recognized by Doxygen.

2. The @file, @author, @date, and @brief tags should form a single group of elements. A detailed description (if available) has to be placed one empty line after the brief description.

3. A file comment should consist of at least the @file and @brief tags.

11.17.3 Validation of Doxygen Comments (Updating API Reference)

Validation of Doxygen code comments might be time-consuming since it requires setting the whole Doxygen project using Doxygen configuration files (doxyfiles). One solution to that problem is to use the configuration created for generating the official VTR documentation. The following steps will show you how to add new code comments to the Sphinx API Reference, available in the VTR documentation:

1. Ensure that the documented project has a doxyfile, and it is added to breathe configuration. All the doxyfiles used by the Sphinx documentation are placed in `<vtr_root>/doc/_doxygen` (For details, check Sphinx API Documentation for C/C++ Projects) This will ensure that Doxygen XMLs will be created for that project during the Sphinx documentation building process.

2. Check that the `<vtr_root>/doc/src/api/<project_name>` directory with a index.rst file exists. If not, create both the directory and the index file. Here is an example of the index.rst file for the VPR project.

```rst
VPR API
=======

.. toctree::
   :maxdepth: 1

   contexts
   netlist

Note: Do not forget about adding the index file title. The ==== marks should be of the same length as the title.
```
3. Create a RST file, which will contain the references to the Doxygen code comments. Sphinx uses the Breathe plugin for extracting Doxygen comments from the generated XML files. The simplest check can be done by dumping all the Doxygen comments from the single file with a ..doxygenfile :: directive.

Assuming that your RST file name is myrst.rst, and you created it to check the Doxygen comments in the mycode.cpp file within the vpr project, the contents of the file might be the following:

```
=====
MyRST
=====

.. doxygenfile:: mycode.cpp
  :project: vpr
```

**Note:** A complete list of Breathe directives can be found in the [Breathe documentation](#).

4. Add the newly created RST file to the index.rst. In this example, that will lead to the following change in the index.rst:

```
VPR API
=======
```

```
.. toctree::
  :maxdepth: 1

  contexts
  netlist
  myrst
```

5. Generate the Sphinx documentation by using `make html` command inside the `<vtr_root>/doc/` directory.

6. The new section should be available in the API Reference. To verify that open the `<vtr_root>/doc/_build/html/index.html` with your browser and check the API Reference section. If the introduced code comments are unavailable, you can analyze the Sphinx build log.

### 11.17.4 Additional Resources

- Doxygen documentation
- Breathe documentation

### 11.18 Developer Tutorials

#### 11.18.1 New Developer Tutorial

**Overview**

Welcome to the Verilog-to-Routing (VTR) Project. This project is an open-source, international, collaboration towards a comprehensive FPGA architecture exploration system that includes CAD tools, benchmarks, transistor-optimized architecture files, and documentation, along with support to make this all fit together and work. The purpose of this
tutorial is to equip you, the new developer, with the tools and understanding that you need to begin making a useful contribution to this project.

While you are going through this tutorial, please record down things that should be changed. Whether it is the tutorial itself, documentation, or other parts of the VTR project. Your thoughts are valuable and welcome because fresh eyes help evaluate whether or not our work is clearly presented.

Environment Setup

Log into your workstation/personal computer. Check your account for general features such as internet, printing, git, etc. If there are problems at this stage, talk to your advisor to get this setup.

If you are not familiar with development on Linux, this is the time to get up to speed. Look up online tutorials on general commands, basic development using Makefiles, etc.

Background Reading

Read the first two chapters of “Architecture and CAD for deep-submicron FPGAs” by Vaughn Betz, et al. This is a great introduction to the topic of FPGA CAD and architecture. Note though that this book is old so it only covers a small core of what the VTR project is currently capable of.

Read chapters 1 to 5 of “FPGA Architecture: Survey and Challenges” by Ian Kuon et al.

Review material learned with fellow colleagues.

Setup VTR

Use git to clone a copy of VTR from the GitHub repository:

https://github.com/verilog-to-routing/vtr-verilog-to-routing

Build the project by running the make command

Run ./run_quick_test.pl to check that the build worked

Follow the Quick Start Guide, and Basic Design Flow Tutorial found in the VTR Documentation (docs.verilogtorouting.org). These tutorials will allow you to run a circuit through the entire flow and read the statistics gathered from that run.

Use VTR

Create your own custom Verilog file. Create your own custom architecture file using one of the existing architecture files as a template. Use VTR to map that circuit that you created to that architecture that you created. The VTR documentation, to be found at the https://docs.verilogtorouting.org will prove useful. You may also wish to look at the following links for descriptions of the language used inside the architecture files:

- Classical Soft Logic Block Example: http://www.eecg.utoronto.ca/vpr/utfal_ex1.html

Perform a simple architecture experiment. Run an experiment that varies Fc_in from 0.01 to 1.00 on the benchmarks ch_intrinsics, or1200, and sha. Use tasks/timing as your template. Graph the geometric average of minimum channel width and critical path delay for these three benchmarks across your different values of Fc_in. Review the results with your colleagues and/or advisor.
Open the Black Box

At this stage, you have gotten a taste of how an FPGA architect would go about using VTR. As a developer though, you need a much deeper understanding of how this tool works. The purpose of this section is to have you learn the details of the VTR CAD flow by having you manually do what the scripts do.

Using the custom Verilog circuit and architecture created in the previous step, directly run Odin II on it to generate a blif netlist. You may need to skim the ODIN_II/README.rst and the vtr_flow/scripts/run_vtr_flow.py.

Using the output netlist of Odin II, run ABC to generate a technology-mapped blif file. You may need to skim the ABC homepage (http://www.eecs.berkeley.edu/~alanmi/abc/).

```
# Run the ABC program from regular terminal (bash shell)
$VTR_ROOT/abc abc

# Using the ABC shell to read and write blif file
abc 01> read_blif Odin_II_output.blif
abc 01> write_blif abc_output.blif
```

Using the output of ABC and your architecture file, run VPR to complete the mapping of a user circuit to a target architecture. You may need to consult the VPR User Manual.

```
# Run the VPR program
$VTR_ROOT/vpr vpr architecture.xml abc_output.blif
```

Read the VPR section of the online documentation.

Submitting Changes and Regression Testing

Read README.developers.md in the base directory of VTR. Code changes rapidly so please help keep this up to date if you see something that is out of date.

Make your first change to git by modifying README.md and pushing it. I recommend adding your name to the list of contributors. If you have nothing to modify, just add/remove a line of whitespace at the bottom of the file.

Now that you have completed the tutorial, you should have a general sense of what the VTR project is about and how the different parts work together. It’s time to talk to your advisor to get your first assignment.

11.18.2 Timing Graph Debugging Tutorial

When developing VPR or creating/calibrating the timing characteristics of a new architectural model it can be helpful to look ‘inside’ at VPR’s timing graph and analysis results.

**Warning:** This is a very low-level tutorial suitable for power-users and VTR developers
Generating a GraphViz DOT file of the Entire Timing Graph

One approach is to have VPR generate a GraphViz DOT file, which visualizes the structure of the timing graph, and the analysis results. This is enabled by running VPR with `vpr --echo_file` set to on. This will generate a set of .dot files in the current directory representing the timing graph, delays, and results of Static Timing Analysis (STA).

```bash
$ vpr $VTR_ROOT/vtr_flow/arch/timing/EAarch.xml $VTR_ROOT/vtr_flow/benchmarks/blif/multiclock/multiclock.blif --echo_file on

$ ls *.dot
timing_graph.place_final.echo.dot timing_graph.place_initial.echo.dot timing_graph.pre_pack.echo.dot
```

The .dot files can then be visualized using a tool like xdot which draws an interactive version of the timing graph.

```bash
$ xdot timing_graph.place_final.echo.dot
```

**Warning:** On all but the smallest designs the full timing graph .dot file is too large to visualize with xdot. See the next section for how to show only a subset of the timing graph.

Which will bring up an interactive visualization of the graph: Where each node in the timing graph is labeled

```
Node(X) (TYPE)
```

Where Node(X) (e.g. Node(3)) represents the ID of the timing graph node, and (TYPE) (e.g. OPIN) is the type of node in the graph.

Each node is also annotated with timing information (produced by STA) like

```
DATA_ARRIVAL
Domain(1) to * from Node(16)
time: 5.2143e-10
```

```
DATA_ARRIVAL
Domain(2) to * from Node(20)
time: 6.9184e-10
```

```
DATA_REQUIRED
Domain(1) to Domain(1) for Node(24)
time: -2.357e-10
```

```
SLACK
Domain(1) to Domain(1) for Node(24)
time: -5.45e-10
```

where the first line of each entry is the type of timing information (e.g. data arrival time, data required time, slack), the second line indicates the related launching and capture clocks (with * acting as a wildcard) and the relevant timing graph node which originated the value, and the third line is the actual time value (in seconds).

The edges in the timing graph are also annotated with their Edge IDs and delays. Special edges related to setup/hold (tsu, thld) and clock-to-q delays (tcq) of sequential elements (e.g. Flip-Flops) are also labeled and drawn with different colors.
Fig. 11.1: Full timing graph visualized with xdot on a very small multi-clock circuit.
Generating a GraphViz DOT file of a subset of the Timing Graph

For most non-trivial designs the entire timing graph is too large and difficult to visualize.

To assist this you can generate a DOT file for a subset of the timing graph with \texttt{vpr --echo_dot_timing_graph_node}

$ \texttt{vpr} \$\texttt{VTR_ROOT/vtr_flow/arch/timing/EAarch.xml} \$\texttt{VTR_ROOT/vtr_flow/benchmarks/blif/_multiclock/multiclock.blif} --echo_file on --echo_dot_timing_graph_node 23

Running \texttt{xdot timing_graph.place_final.echo.dot} now shows the only the subset of the timing graph which fans-in or fans-out of the specified node (in this case node 23).

Cross-referencing Node IDs with VPR Timing Reports

The DOT files only label timing graph nodes with their node IDs. When debugging it is often helpful to correlate these with what are seen in timing reports.

To do this, we need to have VPR generate more detailed timing reports which have additional debug information. This can be done with \texttt{vpr --timing_report_detail} set to debug:

$ \texttt{vpr} \$\texttt{VTR_ROOT/vtr_flow/arch/timing/EAarch.xml} \$\texttt{VTR_ROOT/vtr_flow/benchmarks/blif/_multiclock/multiclock.blif} --timing_report_detail debug

$ \texttt{ls report_timing*}
report_timing.hold.rpt report_timing.setup.rpt

Viewing report_timing.setup.rpt:

#Path 6
Startpoint: FFB.Q[0] (.latch at (1,1) tnode(15) clocked by clk2)
Endpoint : FFD.D[0] (.latch at (1,1) tnode(25) clocked by clk2)
Path Type : setup

<table>
<thead>
<tr>
<th>Point</th>
<th>Incr</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>clock clk2 (rise edge)</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>clock source latency</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>clk2.inpad[0] (.input at (3,2) tnode(4))</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>(intra 'io' routing)</td>
<td>0.042</td>
</tr>
<tr>
<td></td>
<td>(inter-block routing:global net)</td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>(intra 'clb' routing)</td>
<td>0.000</td>
</tr>
<tr>
<td>FFB.clk[0] (.latch at (1,1) tnode(9))</td>
<td>0.000</td>
<td>0.042</td>
</tr>
<tr>
<td></td>
<td>(primitive '.latch' Tcq_max)</td>
<td>0.124</td>
</tr>
<tr>
<td>FFB.Q[0] (.latch at (1,1) tnode(15)) [clock-to-output]</td>
<td>0.000</td>
<td>0.166</td>
</tr>
<tr>
<td></td>
<td>(intra 'clb' routing)</td>
<td>0.120</td>
</tr>
<tr>
<td>to_FFD.in[1] (.names at (1,1) tnode(21))</td>
<td>0.000</td>
<td>0.286</td>
</tr>
<tr>
<td></td>
<td>(primitive '.names' combinational delay)</td>
<td>0.235</td>
</tr>
<tr>
<td>to_FFD.out[0] (.names at (1,1) tnode(23))</td>
<td>0.000</td>
<td>0.521</td>
</tr>
<tr>
<td></td>
<td>(intra 'clb' routing)</td>
<td>0.000</td>
</tr>
<tr>
<td>FFD.D[0] (.latch at (1,1) tnode(25))</td>
<td>0.000</td>
<td>0.521</td>
</tr>
<tr>
<td>data arrival time</td>
<td>0.000</td>
<td>0.521</td>
</tr>
<tr>
<td>clock clk2 (rise edge)</td>
<td>0.000</td>
<td>0.000</td>
</tr>
</tbody>
</table>

(continues on next page)
Fig. 11.2: Subset of the timing graph which fans in and out of node 23.
We can see that the elements corresponding to specific timing graph nodes are labeled with \textit{tnode(X)}. For instance:

\begin{verbatim}
| to_FFD.out[0] (.names at (1,1) tnode(23)) 0.000 0.521 |
\end{verbatim}

shows the netlist pin named \texttt{to\_FFD\_out[0]} is \textit{tnode(23)}, which corresponds to Node(23) in the DOT file.

### 11.19 VTR Support Resources

For support using VPR please use these resources:

1. Check the VTR Documentation: https://docs.verilogtorouting.org
   
   The VTR documentation includes:
   
   - Overviews of what VTR is, and how the flow fits together
   - Tutorials on using VTR
   - Detailed descriptions of tools and their command-line options
   - Descriptions of the file-formats used by VTR

2. Contact the VTR users mailing list: vtr-users@googlegroups.com

   The mailing list includes developers and users of VTR. If you have a specific usage case not covered by the documentation, someone on the mailing list may be able to help.

3. If you’ve found a bug or have an idea for an enhancement consider filing an issue. See here for more details.

### 11.20 VTR License

The software package “VTR” includes the software tools ODIN II, ABC, and VPR as well as additional benchmarks, documentation, libraries and scripts. The authors of the various components of VTR retain their ownership of their tools.

- Unless otherwise noted (in particular ABC, the benchmark circuits and some libraries), all software, documents, and scripts in VTR, follows the standard MIT license described here copied below for your convenience:
The MIT License (MIT)

Copyright 2012 VTR Developers

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

• Terms and conditions for ABC is found here copied below for your convenience:

Copyright (c) The Regents of the University of California. All rights reserved.

Permission is hereby granted, without written agreement and without license or royalty fees, to use, copy, modify, and distribute this software and its documentation for any purpose, provided that the above copyright notice and the following two paragraphs appear in all copies of this software.

IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF THE UNIVERSITY OF CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE SOFTWARE PROVIDED HEREUNDER IS ON AN “AS IS” BASIS, AND THE UNIVERSITY OF CALIFORNIA HAS NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

The benchmark circuits are all open source but each have their own individual terms and conditions which are listed in the source code of each benchmark.

Subject to these conditions, the software is provided free of charge to all interested parties.

If you do decide to use this tool, please reference our work as references are important in academia.

Donations in the form of research grants to promote further research and development on the tools will be gladly accepted, either anonymously or with attribution on our future publications.
Note that changes from release candidates (e.g. v8.0.0-rc1, v8.0.0-rc2) are included/repeated in the final release (e.g. v8.0.0) change log.

12.1 Unreleased

The following are changes which have been implemented in the VTR master branch but have not yet been included in an official release.

12.1.1 Added

12.1.2 Changed

12.1.3 Fixed

12.1.4 Deprecated

12.1.5 Removed

12.2 v8.0.0 - 2020-03-24

12.2.1 Added

- Support for arbitrary FPGA device grids/floorplans
- Support for clustered blocks with width > 1
- Customizable connection-block and switch-blocks patterns (controlled from FPGA architecture file)
- Fan-out dependent routing mux delays
- VPR can generate/load a routing architecture (routing resource graph) in XML format
- VPR can load routing from a .route file
- VPR can performing analysis (STA/Power/Area) independently from optimization (via vpr --analysis)
- VPR supports netlist primitives with multiple clocks
- VPR can perform hold-time (minimum delay) timing analysis
- Minimum delays can be annotated in the FPGA architecture file
• Flow supports formal verification of circuit implementation against input netlist
• Support for generating FASM to drive bitstream generators
• Routing predictor which predicts and aborts impossible routings early (saves significant run-time during minimum channel width search)
• Support for minimum routable channel width ‘hints’ (reduces minimum channel width search run-time if accurate)
• Improved VPR debugging/verbosity controls
• VPR can perform basic netlist cleaning (e.g. sweeping dangling logic)
• VPR graphics visualizations:
  – Critical path during placement/routing
  – Cluster pin utilization heatmap
  – Routing utilization heatmap
  – Routing resource cost heatmaps
  – Placement macros
• VPR can route constant nets
• VPR can route clock nets
• VPR can load netlists in extended BLIF (eBLIF) format
• Support for generating post-placement timing reports
• Improved router ‘map’ lookahead which adapts to routing architecture structure
• Script to upgrade legacy architecture files (vtr_flow/scripts/upgrade_arch.py)
• Support for Fc overrides which depend on both pin and target wire segment type
• Support for non-configurable switches (shorts, inline-buffers) used to model structures like clock-trees and non-linear wires (e.g. ‘L’ or ‘T’ shapes)
• Various other features since VTR 7

12.2.2 Changed

• VPR will exit with code 1 on errors (something went wrong), and code 2 when unable to implement a circuit (e.g. unroutable)
• VPR now gives more complete help about command-line options (vpr -h)
• Improved a wide variety of error messages
• Improved STA timing reports (more details, clearer format)
• VPR now uses Tatum as its STA engine
• VPR now detects mismatched architecture (.xml) and implementation (.net/place/route) files more robustly
• Improved router run-time and quality through incremental re-routing and improved handling of high-fanout nets
• The timing edges within each netlist primitive must now be specified in the section of the architecture file
• All interconnect tags must have unique names in the architecture file
• Connection block input pin switch must now be specified in section of the architecture file
• Renamed switch types buffered/pass_trans to more descriptive tristate/pass_gate in architecture file
• Require longline segment types to have no switchblock/connectionblock specification
• Improve naming (true/false -> none/full/instance) and give more control over block pin equivalnace specifications
• VPR will produce a .route file even if the routing is illegal (aids debugging), however analysis results will not be produced unless vpr --analysis is specified
• VPR long arguments are now always prefixed by two dashes (e.g. --route) while short single-letter arguments are prefixed by a single dash (e.g. -h)
• Improved logic optimization through using a recent 2018 version of ABC and new synthesis script
• Significantly improved implementation quality (~14% smaller minimum routable channel widths, 32-42% reduced wirelength, 7-10% lower critical path delay)
• Significantly reduced run-time (~5.5-6.3x faster) and memory usage (~3.3-5x lower)
• Support for non-contiguous track numbers in externally loaded RR graphs
• Improved placer quality (reduced cost round-off)
• Various other changes since VTR 7

12.2.3 Fixed

• FPGA Architecture file tags can be in arbitrary orders
• SDC command arguments can be in arbitrary orders
• Numerous other fixes since VTR 7

12.2.4 Removed

• Classic VPR timing analyzer
• IO channel distribution section of architecture file

12.2.5 Deprecated

• VPR’s breadth-first router (use the timing-driven router, which provides supperior QoR and Run-time)

12.3 v8.0.0-rc2 - 2019-08-01

12.3.1 Changed

• Support for non-contiguous track numbers in externally loaded RR graphs
• Improved placer quality (reduced cost round-off)
12.4 v8.0.0-rc1 - 2019-06-13

12.4.1 Added

- Support for arbitrary FPGA device grids/floorplans
- Support for clustered blocks with width > 1
- Customizable connection-block and switch-blocks patterns (controlled from FPGA architecture file)
- Fan-out dependent routing mux delays
- VPR can generate/load a routing architecture (routing resource graph) in XML format
- VPR can load routing from a .route file
- VPR can performing analysis (STA/Power/Area) independently from optimization (via vpr --analysis)
- VPR supports netlist primitives with multiple clocks
- VPR can perform hold-time (minimum delay) timing analysis
- Minimum delays can be annotated in the FPGA architecture file
- Flow supports formal verification of circuit implementation against input netlist
- Support for generating FASM to drive bitstream generators
- Routing predictor which predicts and aborts impossible routings early (saves significant run-time during minimum channel width search)
- Support for minimum routable channel width ‘hints’ (reduces minimum channel width search run-time if accurate)
- Improved VPR debugging/verbosity controls
- VPR can perform basic netlist cleaning (e.g. sweeping dangling logic)
- VPR graphics visualizations:
  - Critical path during placement/routing
  - Cluster pin utilization heatmap
  - Routing utilization heatmap
  - Routing resource cost heatmaps
  - Placement macros
- VPR can route constant nets
- VPR can route clock nets
- VPR can load netlists in extended BLIF (eBLIF) format
- Support for generating post-placement timing reports
- Improved router ‘map’ lookahead which adapts to routing architecture structure
- Script to upgrade legacy architecture files (vtr_flow/scripts/upgrade_arch.py)
- Support for Fc overrides which depend on both pin and target wire segment type
- Support for non-configurable switches (shorts, inline-buffers) used to model structures like clock-trees and non-linear wires (e.g. ‘L’ or ‘T’ shapes)
- Various other features since VTR 7
12.4.2 Changed

- VPR will exit with code 1 on errors (something went wrong), and code 2 when unable to implement a circuit (e.g. unroutable)
- VPR now gives more complete help about command-line options (vpr -h)
- Improved a wide variety of error messages
- Improved STA timing reports (more details, clearer format)
- VPR now uses Tatum as its STA engine
- VPR now detects mismatched architecture (.xml) and implementation (.net/.place/.route) files more robustly
- Improved router run-time and quality through incremental re-routing and improved handling of high-fanout nets
- The timing edges within each netlist primitive must now be specified in the section of the architecture file
- All interconnect tags must have unique names in the architecture file
- Connection block input pin switch must now be specified in section of the architecture file
- Renamed switch types buffered/pass_trans to more descriptive tristate/pass_gate in architecture file
- Require longline segment types to have no switchblock/connectionblock specification
- Improve naming (true/false -> none/full/instance) and give more control over block pin equivalence specifications
- VPR will produce a .route file even if the routing is illegal (aids debugging), however analysis results will not be produced unless vpr --analsysis is specified
- VPR long arguments are now always prefixed by two dashes (e.g. --route) while short single-letter arguments are prefixed by a single dash (e.g. -h)
- Improved logic optimization through using a recent 2018 version of ABC and new synthesis script
- Significantly improved implementation quality (~14% smaller minimum routable channel widths, 32-42% reduced wirelength, 7-10% lower critical path delay)
- Significantly reduced run-time (~5.5-6.3x faster) and memory usage (~3.3-5x lower)
- Various other changes since VTR 7

12.4.3 Fixed

- FPGA Architecture file tags can be in arbitrary orders
- SDC command arguments can be in arbitrary orders
- Numerous other fixes since VTR 7

12.4.4 Deprecated

12.4.5 Removed

- Classic VPR timing analyzer
- IO channel distribution section of architecture file
13.1 Mailing Lists

VTR maintains several mailing lists. Most users will be interested in VTR Users and VTR Announce.

- **VTR Announce**
  - VTR release announcements (low traffic)
- **VTR Users**: vtr-users@googlegroups.com
  - Discussions about using the VTR project.
- **VTR Devel**: vtr-devel@googlegroups.com
  - Discussions about VTR development.
- **VTR Commits**:
  - Revision Control Commits to the VTR project.

13.2 Issue Tracker

Please file bugs on our issue tracker.

Pull Requests are welcome!
$VTR_ROOT$ The directory containing the root of the VTR source tree.

For instance, if you extracted/cloned the VTR source into `/home/myusername/vtr`, your $VTR_ROOT$ would be `/home/myusername/vtr`.

**MWTA** Minimum Width Transistor Area (MWTA) is a simple process technology independent unit for measuring circuit area. It corresponds to the size the smallest (minimum width) transistor area.

For example, a $1\times$ (unit-sized) CMOS inverter consists of two minimum width transistors (a PMOS pull-up, and NMOS pull-down).

For more details see [BRM99] (the original presentation of the MWTA model), and [CB13] (an updated MWTA model).
CHAPTER
FIFTEEN

PUBLICATIONS & REFERENCES
16.1 Contexts

16.1.1 Classes

class VprContext : public Context
This object encapsulates VPR’s state.

There is typically a single instance which is accessed via the global variable g_vpr_ctx (see globals.h/.cpp).

It is divided up into separate sub-contexts of logically related data structures.

Each sub-context can be accessed via member functions which return a reference to the sub-context:

- The default the member function (e.g. device()) return an const (immutable) reference providing read-only access to the context. This should be the preferred form, as the compiler will detect unintentional state changes.

- The ‘mutable’ member function (e.g. mutable_device()) will return a non-const (mutable) reference allowing modification of the context. This should only be used on an as-needed basis.

Typical usage in VPR would be to call the appropriate accessor to get a reference to the context of interest, and then operate on it.

For example if we were performing an action which required access to the current placement, we would do:

```cpp
void my_analysis_algorithm() {
    //Get read-only access to the placement
    auto& place_ctx = g_vpr_ctx.placement();

    //Do something that depends on (but does not change)
    //the current placement...
}
```

If we needed to modify the placement (e.g. we were implementing another placement algorithm) we would do:

```cpp
void my_placement_algorithm() {
    //Get read-write access to the placement
    auto& place_ctx = g_vpr_ctx.mutable_placement();

    //Do something that modifies the placement
```
Note: The returned contexts are not copyable, so they must be taken by reference.

16.1.2 Structures

struct AtomContext : public Context
State relating to the atom-level netlist.
This should contain only data structures related to user specified netlist being implemented by VPR onto the target device.

Public Members

AtomNetlist nlist
Atom netlist.

AtomLookup lookup
Mappings to/from the Atom Netlist to physically described .blif models.

struct ClusteringContext : public Context
State relating to clustering.
This should contain only data structures that describe the current clustering/packing, or related clusterer/packer algorithmic state.

Public Members

ClusteredNetlist clb_nlist
New netlist class derived from Netlist.

struct Context
A Context is collection of state relating to a particular part of VPR.
This is a base class who’s only purpose is to disable copying of contexts. This ensures that attempting to use a context by value (instead of by reference) will result in a compilation error.
No data or member functions should be defined in this class!


struct DeviceContext : public Context
State relating the device.
This should contain only data structures describing the targeted device.
Public Members

DeviceGrid **grid**
FPGA complex block grid \([0 .. \text{grid.width()-1}][0 .. \text{grid.height()-1}]\).

bool **has_multiple_equivalent_tiles**
Boolean that indicates whether the architecture implements an N:M physical tiles to logical blocks mapping.

\(t\text{\_chan\_width} \text{\ chan\_width}\)
chan_width is for x|y-directed channels; i.e. between rows

\(std::vector\langle t\_rr\_rc\_data\rangle \text{\ rr\_rc\_data}\)
Fly-weighted Resistance/Capacitance data for RR Nodes.

\(std::vector<\text{std::vector<int>}> \text{\ rr\_non\_config\_node\_sets}\)
Sets of non-configurably connected nodes.

\(std::unordered\_map<int, int> \text{\ rr\_node\_to\_non\_config\_node\_set}\)
Reverse look-up from RR node to non-configurably connected node set (index into rr_nonconf_node_sets)

int **virtual\_clock\_network\_root\_idx**
rr_node_idx that connects to the input of all clock network wires
Useful for two stage clock routing XXX: currently only one place to source the clock networks so only storing a single value

\(std::vector<\text{std::map<int, int>}> \text{\ switch\_fanin\_remap}\)
switch_fanin_remap is only used for printing out switch fanin stats (the -switch_stats option)
array index: \([0..(\text{num\_arch\_switches}-1)]\); map key: num of all possible fanin of that type of switch on chip
map value: remapped switch index (index in rr_switch_inf)

\(std::string \text{\ read\_rr\_graph\_filename}\)
Name of rrgraph file read (if any).
Used to determine when reading rrgraph if file is already loaded.

struct PlacementContext : public Context
State relating to placement.
This should contain only data structures that describe the current placement, or related placer algorithm state.
Public Members

\[
\text{vtr::vector\_map}<\text{ClusterBlockId, t\_block\_loc}> \text{ block\_locs}
\]
Clustered block placement locations.

\[
\text{vtr::vector\_map}<\text{ClusterPinId, int}> \text{ physical\_pins}
\]
Clustered pin placement mapping with physical pin.

\[
\text{vtr::Matrix}<\text{t\_grid\_blocks}> \text{ grid\_blocks}
\]
Clustered block associated with each grid location (i.e. inverse of block\_locs)

\[
\text{std::vector<pl\_macro}> \text{ pl\_macros}
\]
The pl\_macros array stores all the placement macros (usually carry chains).

\[
\text{t\_compressed\_block\_grids} \text{ compressed\_block\_grids}
\]
Compressed grid space for each block type.
Used to efficiently find logically ‘adjacent’ blocks of the same block type even though the may be physically far apart

\[
\text{std::string} \text{ placement\_id}
\]
SHA256 digest of the .place file.
Used for unique identification and consistency checking


class PowerContext : public Context
State relating to power analysis.
This should contain only data structures related to power analysis, or related power analysis algorithmic state.

Public Members

\[
\text{std::unordered\_map<AtomNetId, t\_net\_power}> \text{ atom\_net\_power}
\]
Atom net power info.

class RoutingContext : public Context
State relating to routing.
This should contain only data structures that describe the current routing implementation, or related router algorithmic state.

Public Members

\[
\text{vtr::dynamic\_bitset} \text{ non\_configurable\_bitset}
\]
Information about whether a node is part of a non-configurable set.
(i.e. connected to others with non-configurable edges like metal shorts that can’t be disabled) Stored in a single bit per rr\_node for efficiency bit value 0: node is not part of a non-configurable set bit value 1: node is part of a non-configurable set Initialized once when RoutingContext is initialized, static throughout invocation of router
Information about current routing status of each net.

Limits area within which each net must be routed.

SHA256 digest of the .route file.

Used for unique identification and consistency checking

Cache of router lookahead object.

Cache key: (lookahead type, read lookahead (if any), segment definitions).

State relating to timing.

This should contain only data structures related to timing analysis, or related timing analysis algorithmic state.

This represents the timing dependencies between pins of the atom netlist

The current timing constraints, as loaded from an SDC file (or set by default).

These specify how timing analysis is performed (e.g. target clock periods)

The netlist logically consists of several different components: Blocks, Ports, Pins and Nets Each component in the netlist has a unique template identifier (BlockId, PortId, PinId, NetId) used to retrieve information about it. In this implementation these ID’s are unique throughout the netlist (i.e. every port in the netlist has a unique ID, even if the ports share a common type).

A Block is the primitive netlist element (a node in the netlist hyper-graph). Blocks have various attributes (a name, a type etc.) and are associated with sets of input/output/clock ports.

Block related information can be retrieved using the block_*() member functions.

Pins
Pins define single-bit connections between a block and a net.
Pin related information can be retrieved using the pin_*() member functions.

Nets
Nets represent the connections between blocks (the edges of the netlist hyper-graph). Each net has a single driver pin, and a set of sink pins.
Net related information can be retrieved using the net_*() member functions.

Ports
A Port is a (potentially multi-bit) group of pins.
For example, the two operands and output of an N-bit adder would logically be grouped as three ports. Ports have a specified bit-width which defines how many pins form the port.
Port related information can be retrieved using the port_*() member functions.

Usage
The following provides usage examples for common use-cases.

Walking the netlist
To iterate over the whole netlist use the blocks() and/or nets() member functions:

```cpp
Netlist netlist;
    // ... initialize the netlist

    // Iterate over all the blocks
    for(BlockId blk_id : netlist.blocks()) {
        // Do something with each block
    }

    // Iterate over all the nets
    for(NetId net_id : netlist.nets()) {
        // Do something with each net
    }
```

To retrieve information about a netlist component call one of the associated member functions:

```cpp
    // Print out each block's name
    for(BlockId blk_id : netlist.blocks()) {
        // Get the block name
        const std::string& block_name = netlist.block_name(blk_id);

        // Print it
        printf("Block: %s\n", block_name.c_str());
    }
```

Note that the member functions are associated with the type of component (e.g. block_name() yields the name of a block, net_name() yields the name of a net).

Tracing cross-references
It is common to need to trace the netlist connectivity. The Netlist allows this to be done efficiently by maintaining cross-references between the various netlist components.

The following diagram shows the main methods and relationships between netlist components:

```
+---------+ pin_block()
| |<--------------------------+
| Block | |
| |-----------------------+ |
+---------+ block_pins() |
    ^ v
| ^ | +---------+ net_pins() +---------+
| | | |<-------------| |
block_ports() | | port_block() | Pin | | Net |
| | | |------------->| |
| | ^ | +---------+ pin_net() +---------+
| | v ^
+---------+ port_pins() |
    ^ v
| +---------+ pin_port()
```

![Netlist Diagram](image)

Note that methods which are plurals (e.g. net_pins()) return multiple components.

As an example consider the case where we wish to find all the blocks associated with a particular net:

```cpp
NetId net_id;

//... Initialize net_id with the net of interest

//Iterate through each pin on the net to get the associated port
for(PinId pin_id : netlist.net_pins(net_id)) {
    //Get the port associated with the pin
    PortId port_id = netlist.pin_port(pin_id);

    //Get the block associated with the port
    BlockId blk_id = netlist.port_block(port_id);

    //Print out the block name
    const std::string& block_name = netlist.block_name(blk_id);
    printf("Associated block: %s\n", block_name.c_str());
}
```

Netlist also defines some convenience functions for common operations to avoid tracking the intermediate IDs if they are not needed. The following produces the same result as above:

```cpp
NetId net_id;

//... Initialize net_id with the net of interest

//Iterate through each pin on the net to get the associated port
for(PinId pin_id : netlist.net_pins(net_id)) {

(continues on next page)
As another example, consider the inverse problem of identifying the nets connected as inputs to a particular block:

```cpp
BlkId blk_id;

//... Initialize blk_id with the block of interest

//Iterate through the ports
for(PortId port_id : netlist.block_input_ports(blk_id)) {
    //Iterate through the pins
    for(PinId pin_id : netlist.port_pins(port_id)) {
        //Retrieve the net
        NetId net_id = netlist.pin_net(pin_id);

        //Get its name
        const std::string& net_name = netlist.net_name(net_id);
        printf("Associated net: %s\n", net_name.c_str());
    }
}
```

Here we used the `block_input_ports()` method which returned an iterable range of all the input ports associated with `blk_id`. We then used the `port_pins()` method to get iterable ranges of all the pins associated with each port, from which we can find the associated net.

Often port information is not relevant so this can be further simplified by iterating over a block’s pins directly (e.g. by calling one of the `block_*_pins()` functions):

```cpp
BlkId blk_id;

//... Initialize blk_id with the block of interest

//Iterate over the blocks ports directly
for(PinId pin_id : netlist.block_input_pins(blk_id)) {
    //Retrieve the net
    NetId net_id = netlist.pin_net(pin_id);

    //Get its name
    const std::string& net_name = netlist.net_name(net_id);
    printf("Associated net: %s\n", net_name.c_str());
}
```

Note the use of range-based-for loops in the above examples; it could also have written (more verbosely) using a conventional for loop and explicit iterators as follows:
BlkId blk_id;

///... Initialize blk_id with the block of interest

//Iterate over the blocks ports directly
auto pins = netlist.block_input_pins(blk_id);
for(auto pin_iter = pins.begin(); pin_iter != pins.end(); ++pin_iter) {

    //Retrieve the net
    NetId net_id = netlist.pin_net(*pin_iter);
    //Get its name
    const std::string& net_name = netlist.net_name(net_id);
    printf("Associated net: %s\n", net_name.c_str());
}

Creating the netlist

The netlist can be created by using the create_*() member functions to create individual Blocks/Ports/Pins/Nets.

For instance to create the following netlist (where each block is the same type, and has an input port ‘A’ and output port ‘B’):

```
----------- net1 -----------
| block_1 |--------------------| block_2 |
----------- |  -----------
|          |
|          |
|          |
---------->| block_3 |
```

We could do the following:

```
const t_model* blk_model = .... //Initialize the block model appropriately

Netlist netlist("my_netlist"); //Initialize the netlist with name 'my_netlist'

//Create the first block
BlockId blk1 = netlist.create_block("block_1", blk_model);

//Create the first block's output port
//Note that the input/output/clock type of the port is determined automatically from the block model
PortId blk1_out = netlist.create_port(blk1, "B");

//Create the net
NetId net1 = netlist.create_net("net1");

//Associate the net with blk1
netlist.create_pin(blk1_out, 0, net1, PinType::DRIVER);

//Create block 2 and hook it up to net1
BlockId blk2 = netlist.create_block("block_2", blk_model);
PortId blk2_in = netlist.create_port(blk2, "A");
```

(continues on next page)
netlist.create_pin(blk2_in, 0, net1, PinType::SINK);

//Create block 3 and hook it up to net1
BlockId blk3 = netlist.create_block("block_3", blk_model);
PortId blk3_in = netlist.create_port(blk3, "A");
netlist.create_pin(blk3_in, 0, net1, PinType::SINK);

Modifying the netlist

The netlist can also be modified by using the remove_*() member functions. If we wanted to remove block_3 from the netlist creation example above we could do the following:

//Mark blk3 and any references to it invalid
netlist.remove_block(blk3);

//Compress the netlist to actually remove the data associated with blk3
// NOTE: This will invalidate all client held IDs (e.g. blk1, blk1_out, net1, blk2,...blk2_in)
netlist.compress();

The resulting netlist connectivity now looks like:

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>block_1</td>
<td>block_2</td>
</tr>
</tbody>
</table>

Note that until compress() is called any ‘removed’ elements will have invalid IDs (e.g. BlockId::INVALID()). As a result after calling remove_block() (which invalidates blk3) we then called compress() to remove the invalid IDs.

Also note that compress() is relatively slow. As a result avoid calling compress() after every call to a remove_*() function, and instead batch up calls to remove_*() and call compress() only after a set of modifications have been applied.

Verifying the netlist

Particularly after construction and/or modification it is a good idea to check that the netlist is in a valid and consistent state. This can be done with the verify() member function:

netlist.verify()

If the netlist is not valid verify() will throw an exception, otherwise it returns true.

Invariants

The Netlist maintains stronger invariants if the netlist is in compressed form.

is compressed (‘not dirty’)

If the netlist is compressed (i.e. !is_dirty(), meaning there have been NO calls to remove_*() since the last call to compress()) the following invariant will hold:

• Any range returned will contain only valid IDs

In practise this means the following conditions hold:

• Blocks will not contain empty ports/pins (e.g. ports with no pin/net connections)
• Ports will not contain pins with no associated net
• Nets will not contain invalid sink pins
This means that no error checking for invalid IDs is needed if simply iterating through netlist (see below for some exceptions).

NOTE: you may still encounter invalid IDs in the following cases:

- net_driver() will return an invalid ID if the net is undriven
- port_pin() / port_net() will return an invalid ID if the bit index corresponds to an unconnected pin

If the netlist is not compressed (i.e. is_dirty()), meaning there have been calls to remove_*() with no subsequent calls to compress()) then the invariant above does not hold.

Any range may return invalid IDs. In practise this means,

- Blocks may contain invalid ports/pins
- Ports may contain invalid pins
- Pins may not have a valid associated net
- Nets may contain invalid sink pins

Implementation Details

The netlist is stored in Struct-of-Arrays format rather than the more conventional Array-of-Structs. This improves cache locality by keeping component attributes of the same type in contiguous memory. This prevents unneeded member data from being pulled into the cache (since most code accesses only a few attributes at a time this tends to be more efficient).

Clients of this class pass nearly-opaque IDs (BlockId, PortId, PinId, NetId, StringId) to retrieve information. The ID is internally converted to an index to retrieve the required value from it’s associated storage.

By using nearly-opaque IDs we can change the underlying data layout as need to optimize performance/memory, without disrupting client code.

**Strings**

To minimize memory usage, we store each unique string only once in the netlist and give it a unique ID (StringId). Any references to this string then make use of the StringId.

In particular this prevents the (potentially large) strings from begin duplicated multiple times in various look-ups, instead the more space efficient StringId is duplicated.

Note that StringId is an internal implementation detail and should not be exposed as part of the public interface. Any public functions should take and return std::string’s instead.

**Block pins/Block ports data layout**

The pins/ports for each block are stored in a similar manner, for brevity we describe only pins here.

The pins for each block (i.e. PinId’s) are stored in a single vector for each block (the block_pins_member). This allows us to iterate over all pins (i.e. block_pins()), or specific subsets of pins (e.g. only inputs with block_input_pins()).

To accomplish this all pins of the same group (input/output/clock) are located next to each other. An example is shown below, where the block has n input pins, m output pins and k clock pins.

---
| ipin_1 | ipin_2 | ... | ipin_n | opin_1 | opin_2 | ... | opin_m | clock_pin_1 | clock_pin_2 | ... | clock_pin_k |
---

(continues on next page)
Provided we know the internal dividing points (i.e. opin_begin and clock_pin_begin) we can easily build the various ranges of interest:

| all pins   | [begin, end)                             |
| input pins | [begin, opin_begin)                       |
| output pins| [opin_begin, clock_pin_begin)             |
| clock pins | [clock_pin_begin, end)                    |

Since any reallocation would invalidate any iterators to these internal dividers, we separately store the number of input/output/clock pins per block (i.e. in block_num_input_pins_, block_num_output_pins_ and block_num_clock_pins_). The internal dividers can then be easily calculated (e.g. see block_output_pins()), even if new pins are inserted (provided the counts are updated).

Adding data to the netlist

The Netlist should contain only information directly related to the netlist state (i.e. netlist connectivity). Various mappings to/from elements (e.g. what CLB contains an atom block), and algorithmic state (e.g. if a net is routed) do NOT constitute netlist state and should NOT be stored here.

Such implementation state should be stored in other data structures (which may reference the Netlist’s IDs).

The netlist state should be immutable (i.e. read-only) for most of the CAD flow.

Interactions with other netlists

Currently, the AtomNetlist and ClusteredNetlist are both derived from Netlist. The AtomNetlist has primitive specific details (t_model, TruthTable), and handles all operations with the atoms. The ClusteredNetlist contains information on the CLB (Clustered Logic Block) level, which includes the physical description of the blocks (t_logical_block_type), as well as the internal hierarchy and wiring (t_pb/t_pb_route).

The calling-conventions of the functions in the AtomNetlist and ClusteredNetlist is as follows:

Functions where the derived class (Atom/Clustered) calls the base class (Netlist) create_*()

Functions where the base class calls the derived class (Non-Virtual Interface idiom as described https://en.wikibooks.org/wiki/More_C%2B%2B_Idioms/Non-Virtual_Interface) remove_*() clean_*() validate_*_sizes() shrink_to_fit()

The derived functions based off of the virtual functions have suffix *_impl()

template<typename BlockId, typename PortId, typename PinId, typename NetId>
class Netlist

404 Chapter 16. VPR API
Public Functions

const std::string &netlist_name() const
Retrieve the name of the netlist.

const std::string &netlist_id() const
Retrieve the unique identifier for this netlist. This is typically a secure digest of the input file.

bool verify() const
Sanity check for internal consistency (throws an exception on failure)

bool is_dirty() const
Returns true if the netlist has invalid entries due to modifications (e.g. from remove_*() calls)

bool is_compressed() const
Returns true if the netlist has no invalid entries due to modifications (e.g. from remove_*() calls)

Note: This is a convenience method which is the logical inverse of is_dirty()

void print_stats() const
Item counts and container info (for debugging)

const std::string &block_name(const BlockId blk_id) const
Returns the name of the specified block.

bool block_is_combinational(const BlockId blk_id) const
Returns true if the block is purely combinational (i.e. no input clocks and not a primary input.

attr_range block_attrs(const BlockId blk_id) const
Returns a range of all attributes associated with the specified block.

param_range block_params(const BlockId blk_id) const
Returns a range of all parameters associated with the specified block.

pin_range block_pins(const BlockId blk_id) const
Returns a range of all pins associated with the specified block.

pin_range block_input_pins(const BlockId blk_id) const
Returns a range of all input pins associated with the specified block.

pin_range block_output_pins(const BlockId blk_id) const
Returns a range of all output pins associated with the specified block.

Note: This is typically only data pins, but some blocks (e.g. PLLs) can produce outputs which are clocks.

pin_range block_clock_pins(const BlockId blk_id) const
Returns a range of all clock pins associated with the specified block.

port_range block_ports(const BlockId blk_id) const
Returns a range of all ports associated with the specified block.

port_range block_input_ports(const BlockId blk_id) const
Returns a range consisting of the input ports associated with the specified block.

port_range block_output_ports(const BlockId blk_id) const
Returns a range consisting of the output ports associated with the specified block.
### Note
This is typically only data ports, but some blocks (e.g. PLLs) can produce outputs which are clocks.

**port_range** `block_clock_ports`(const `BlockId` blk_id) const
Returns a range consisting of the input clock ports associated with the specified block.

**void** remove_block(const `BlockId` blk_id)
Removes a block from the netlist. This will also remove the associated ports and pins.

**Parameters**
- `blk_id` – The block to be removed

**const std::string &** port_name(const `PortId` port_id) const
Returns the name of the specified port.

**BlockId** port_block(const `PortId` port_id) const
Returns the block associated with the specified port.

**pin_range** port_pins(const `PortId` port_id) const
Returns the set of valid pins associated with the port.

**PinId** port_pin(const `PortId` port_id, const `BitIndex` port_bit) const
Returns the pin (potentially invalid) associated with the specified port and port bit index.

**Note:** This function is a synonym for `find_pin()`

**Parameters**
- `port_id` – The ID of the associated port
- `port_bit` – The bit index of the pin in the port

**NetId** port_net(const `PortId` port_id, const `BitIndex` port_bit) const
Returns the net (potentially invalid) associated with the specified port and port bit index.

**Parameters**
- `port_id` – The ID of the associated port
- `port_bit` – The bit index of the pin in the port

**BitIndex** port_width(const `PortId` port_id) const
Returns the width (number of bits) in the specified port.

**PortType** port_type(const `PortId` port_id) const
Returns the type of the specified port.

**void** remove_port(const `PortId` port_id)
Removes a port from the netlist.

The port's pins are also marked invalid and removed from any associated nets

**Parameters**
- `port_id` – The ID of the port to be removed

**std::string** pin_name(const `PinId` pin_id) const
Returns the constructed name (derived from block and port) for the specified pin.

**PinType** pin_type(const `PinId` pin_id) const
Returns the type of the specified pin.

**NetId** pin_net(const `PinId` pin_id) const
Returns the net associated with the specified pin.
int \texttt{pin\_net\_index}(\texttt{const PinId pin\_id}) \texttt{const}
\begin{itemize}
\item Returns the index of the specified pin within its connected net.
\end{itemize}

\texttt{PortId pin\_port}(\texttt{const PinId pin\_id}) \texttt{const}
\begin{itemize}
\item Returns the port associated with the specified pin.
\end{itemize}

\texttt{BitIndex pin\_port\_bit}(\texttt{const PinId pin\_id}) \texttt{const}
\begin{itemize}
\item Returns the port bit index associated with the specified pin.
\end{itemize}

\texttt{BlockId pin\_block}(\texttt{const PinId pin\_id}) \texttt{const}
\begin{itemize}
\item Returns the block associated with the specified pin.
\end{itemize}

\texttt{PortType pin\_port\_type}(\texttt{const PinId pin\_id}) \texttt{const}
\begin{itemize}
\item Returns the port type associated with the specified pin.
\end{itemize}

\texttt{bool pin\_is\_constant}(\texttt{const PinId pin\_id}) \texttt{const}
\begin{itemize}
\item Returns true if the pin is a constant (i.e., its value never changes).
\end{itemize}

\texttt{void remove\_pin}(\texttt{const PinId pin\_id})
\begin{itemize}
\item Removes a pin from the netlist.
\end{itemize}
\begin{itemize}
\item The pin is marked invalid, and removed from any associated nets
\end{itemize}
\begin{itemize}
\item \textbf{Parameters} \texttt{pin\_id} – The pin\_id of the pin to be removed
\end{itemize}

\texttt{const std::string &net\_name}(\texttt{const NetId net\_id}) \texttt{const}
\begin{itemize}
\item Returns the name of the specified net.
\end{itemize}

\texttt{pin\_range net\_pins}(\texttt{const NetId net\_id}) \texttt{const}
\begin{itemize}
\item Returns a range consisting of all the pins in the net (driver and sinks)
\end{itemize}
\begin{itemize}
\item The first element in the range is the driver (and may be invalid) The remaining elements (potentially none) are the sinks
\end{itemize}

\texttt{PinId net\_pin}(\texttt{const NetId net\_id, int net\_pin\_index}) \texttt{const}
\begin{itemize}
\item Returns the net\_pin\_index’th pin of the specified net.
\end{itemize}

\texttt{BlockId net\_pin\_block}(\texttt{const NetId net\_id, int net\_pin\_index}) \texttt{const}
\begin{itemize}
\item Returns the block associated with the net\_pin\_index’th pin of the specified net.
\end{itemize}

\texttt{PinId net\_driver}(\texttt{const NetId net\_id}) \texttt{const}
\begin{itemize}
\item Returns the (potentially invalid) net driver pin.
\end{itemize}

\texttt{BlockId net\_driver\_block}(\texttt{const NetId net\_id}) \texttt{const}
\begin{itemize}
\item Returns the (potentially invalid) net driver block.
\end{itemize}

\texttt{pin\_range net\_sinks}(\texttt{const NetId net\_id}) \texttt{const}
\begin{itemize}
\item Returns a (potentially empty) range consisting of net’s sink pins.
\end{itemize}

\texttt{bool net\_is\_constant}(\texttt{const NetId net\_id}) \texttt{const}
\begin{itemize}
\item Returns true if the net is driven by a constant pin (i.e., its value never changes).
\end{itemize}

\texttt{void remove\_net}(\texttt{const NetId net\_id})
\begin{itemize}
\item Removes a net from the netlist.
\end{itemize}
\begin{itemize}
\item This will mark the net’s pins as having no associated.
\end{itemize}
\begin{itemize}
\item \textbf{Parameters} \texttt{net\_id} – The net to be removed
\end{itemize}

\texttt{void remove\_net\_pin}(\texttt{const NetId net\_id, const PinId pin\_id})
\begin{itemize}
\item Removes a connection between a net and pin.
\end{itemize}
\begin{itemize}
\item The pin is removed from the net and the pin will be marked as having no associated net
\end{itemize}
\begin{itemize}
\item \textbf{Parameters} \texttt{net\_id} – The net to be removed
\end{itemize}
\begin{itemize}
\item \texttt{pin\_id} – The pin\_id of the pin to be removed
\end{itemize}
• **net_id** – The net from which the pin is to be removed

• **pin_id** – The pin to be removed from the net

```cpp
block_range blocks() const
    Returns a range consisting of all blocks in the netlist.
```

```cpp
port_range ports() const
    Returns a range consisting of all ports in the netlist.
```

```cpp
net_range nets() const
    Returns a range consisting of all nets in the netlist.
```

```cpp
pin_range pins() const
    Returns a range consisting of all pins in the netlist.
```

```cpp
BlockId find_block(const std::string &name) const
    Returns the BlockId of the specified block or BlockId::INVALID() if not found.
    
    **Parameters**
    - **name** – The name of the block
```

```cpp
PortId find_port(const BlockId blk_id, const std::string &name) const
    Returns the PortId of the specified port if it exists or PortId::INVALID() if not.
    
    **Parameters**
    - **blk_id** – The ID of the block who’s ports will be checked
    - **name** – The name of the port to look for
```

```cpp
NetId find_net(const std::string &name) const
    Returns the NetId of the specified net or NetId::INVALID() if not found.
    
    **Parameters**
    - **name** – The name of the net
```

```cpp
PinId find_pin(const PortId port_id, BitIndex port_bit) const
    Returns the PinId of the specified pin or PinId::INVALID() if not found.
    
    **Parameters**
    - **port_id** – The ID of the associated port
    - **port_bit** – The bit index of the pin in the port
```

```cpp
PinId find_pin(const std::string name) const
    Returns the PinId of the specified pin or PinId::INVALID() if not found.
    
    **Note:** This method is SLOW, O(num_pins) — avoid if possible
    
    **Parameters**
    - **name** – The name of the pin
```

```cpp
void set_pin_net(const PinId pin, PinType pin_type, const NetId net)
    Add the specified pin to the specified net as pin_type.
    Automatically removes any previous net connection for this pin.
    
    **Parameters**
```
• **pin** – The pin to add

• **pin_type** – The type of the pin (i.e. driver or sink)

• **net** – The net to add the pin to

```c
void set_pin_is_constant(const PinId pin_id, const bool value)
```

Mark a pin as being a constant generator.

There are some cases where a pin can not be identified as a is constant until after the full netlist has been built; so we expose a way to mark existing pins as constants.

**Parameters**

• **pin_id** – The pin to be marked

• **value** – The boolean value to set the pin_is_constant attribute

```c
void set_block_name(const BlockId blk_id, const std::string new_name)
```

Re-name a block.

**Parameters**

• **blk_id** – The block to be renamed

• **new_name** – The new name for the specified block

```c
void set_block_attr(const BlockId blk_id, const std::string &name, const std::string &value)
```

Set a block attribute.

**Parameters**

• **blk_id** – The block to which the attribute is attached

• **name** – The name of the attribute to set

• **value** – The new value for the specified attribute on the specified block

```c
void set_block_param(const BlockId blk_id, const std::string &name, const std::string &value)
```

Set a block parameter.

**Parameters**

• **blk_id** – The block to which the parameter is attached

• **name** – The name of the parameter to set

• **value** – The new value for the specified parameter on the specified block

```c
void merge_nets(const NetId driver_net, const NetId sink_net)
```

Merges sink_net into driver_net.

After merging driver_net will contain all the sinks of sink_net

**Parameters**

• **driver_net** – The net which includes the driver pin

• **sink_net** – The target net to be merged into driver_net (must have no driver pin)

```c
IdRemapper remove_and_compress()
```

Wrapper for `remove_unused() & compress()`

This function should be used in the case where a netlist is fully modified

```c
void remove_unused()
```

This should be called after completing a series of netlist modifications (e.g. removing blocks/ports/pins/nets).
Marks netlist components which have become redundant due to other removals (e.g. ports with only invalid pins) as invalid so they will be destroyed during `compress()`.

IdRemapper `compress()`
Compresses the netlist, removing any invalid and/or unreferenced blocks/ports/pins/nets.

**Note:** this invalidates all existing IDs!

### 16.2.2 Clustered Netlist

**Overview**

The `ClusteredNetlist` is derived from the `Netlist` class, and contains some separate information on Blocks, Pins, and Nets. It does not make use of Ports.

**Blocks**

The pieces of unique block information are: `block_pbs_`: Physical block describing the clustering and internal hierarchy structure of each CLB. `block_types_`: The type of physical block the block is mapped to, e.g. logic block, RAM, DSP (Can be user-defined types). `block_nets_`: Based on the block’s pins (indexed from [0…num_pins - 1]), lists which pins are used/unused with the net using it. `block_pin_nets_`: Returns the index of a pin relative to the net, when given a block and a pin’s index on that block (from the type descriptor). Differs from `block_nets_`.

**Differences between block_nets_ & block_pin_nets_**

| 0--->| |---3   | 0--->| |--3 |
| 1--->| Block |--->4 | 1--->| Block |--->4 |
| 2--->| |--->5 | 2--->| |--->5 |

`block_nets_` tracks all pins on a block, and returns the ClusterNetId to which a pin is connected to. If the pin is unused/open, ClusterNetId::INVALID() is stored.

`block_pin_nets_` tracks whether the nets connected to the block are drivers/receivers of that net. Driver/receiver nets are determined by the pin_class of the block’s pin. A net connected to a driver pin in the block has a 0 is stored. A net connected to a receiver has a counter (from [1…num_sinks - 1]).

The net is connected to multiple blocks. Each `block_pin_nets_` has a unique number in that net.

E.g.

```
+-------------+  +-------------+  +-------------+  +-------------+
  0-->|   |--3 Net A  0-->|   |--3 Net A  0-->|   |--3 Net A  0-->|   |--3 Net A
  1-->| Block 1 |--4------1-->| Block 2 |--4------1-->| Block 3 |--4------1-->| Block 4 |--4
  2-->|   |--5  2-->|   |--5  2-->|   |--5  2-->|   |--5
     +-------------+  +-------------+  +-------------+  +-------------+
               |  +-------------+  +-------------+  +-------------+  +-------------+
               |                    |  +-------------+  +-------------+  +-------------+  +-------------+
               |                    |                    |  +-------------+  +-------------+  +-------------+  +-------------+
               |                    |                    |                    |  +-------------+  +-------------+  +-------------+  +-------------+
               0-->| Block 3 |  0-->| Block 3 |  0-->| Block 3 |  0-->| Block 3 |
               |                    |  |                    |  |                    |  |                    |
               |                    |  |                    |  |                    |  |                    |
               |                    |  |                    |  |                    |  |                    |
               |                    |  |                    |  |                    |  |                    |
```


In the example, Net A is driven by Block 1, and received by Blocks 2 & 3. For Block 1, block_pin_nets_ of pin 4 returns 0, as it is the driver. For Block 2, block_pin_nets_ of pin 1 returns 1 (or 2), non-zero as it is a receiver. For Block 3, block_pin_nets_ of pin 0 returns 2 (or 1), non-zero as it is also a receiver.

The block_pin_nets_ data structure exists for quick indexing, rather than using a linear search with the available functions from the base Netlist, into the net_delay_ structure in the PostClusterDelayCalculator of inter_cluster_delay(). net_delay_ is a 2D array, where the indexing scheme is [net_id] followed by [pin_index on net].

**Pins**

The only piece of unique pin information is: logical_pin_index_

*Example of logical_pin_index_*

Given a ClusterPinId, logical_pin_index_ will return the index of the pin within its block relative to the t_logical_block_type (logical description of the block).

```
+-----------+
0-->|O X|-->3
1-->|O Block O|-->4
2-->|X O|-->5 (e.g. ClusterPinId = 92)
+-----------+
```

The index skips over unused pins, e.g. CLB has 6 pins (3 in, 3 out, numbered [0…5]), where the first two ins, and last two outs are used. Indices [0,1] represent the ins, and [4,5] represent the outs. Indices [2,3] are unused. Therefore, logical_pin_index_[92] = 5.

**Nets**

The pieces of unique net information stored are: net_global_: Boolean mapping whether the net is global net_fixed_: Boolean mapping whether the net is fixed (i.e. constant)

**Implementation**

For all create_* functions, the ClusteredNetlist will wrap and call the Netlist’s version as it contains additional information that the base Netlist does not know about.

All functions with suffix *_impl() follow the Non-Virtual Interface (NVI) idiom. They are called from the base Netlist class to simplify pre/post condition checks and prevent Fragile Base Class (FBC) problems.

Refer to netlist.h for more information.

class ClusteredNetlist : public Netlist<ClusterBlockId, ClusterPortId, ClusterPinId, ClusterNetId>

**Public Functions**

ClusteredNetlist(std::string name = "", std::string id = "")

Constructs a netlist.

**Parameters**

- **name** – the name of the netlist (e.g. top-level module)
- **id** – a unique identifier for the netlist (e.g. a secure digest of the input file)

```
t_pb *block_pb(const ClusterBlockId id) const

Returns the physical block.
```
t_logical_block_type_ptr block_type(const ClusterBlockId id) const
  Returns the type of CLB (Logic block, RAM, DSP, etc.)

ClusterNetId block_net(const ClusterBlockId blk_id, const int pin_index) const
  Returns the net of the block attached to the specific pin index.

int block_pin_net_index(const ClusterBlockId blk_id, const int pin_index) const
  Returns the count on the net of the block attached.

ClusterPinId block_pin(const ClusterBlockId blk, const int logical_pin_index) const
  Returns the logical pin Id associated with the specified block and logical pin index.

bool block_contains_primary_output(const ClusterBlockId blk) const
  Returns true if the specified block contains a primary output (e.g. BLIF .output primitive)

int pin_logical_index(const ClusterPinId pin_id) const
  Returns the logical pin index (i.e. pin index on the t_logical_block_type) of the cluster pin.

int net_pin_logical_index(const ClusterNetId net_id, int net_pin_index) const
  Finds the net_index'th net pin (e.g. the 6th pin of the net) and returns the logical pin index (i.e. pin index on the t_logical_block_type) of the block to which the pin belongs.

Parameters
- net_id – The net
- net_pin_index – The index of the pin in the net

bool net_is_ignored(const ClusterNetId id) const
  Returns whether the net is ignored i.e. not routed.

bool net_is_global(const ClusterNetId id) const
  Returns whether the net is global.

ClusterBlockId create_block(const char *name, t_pb *pb, t_logical_block_type_ptr type)
  Create or return an existing block in the netlist.

Parameters
- name – The unique name of the block
- pb – The physical representation of the block
- type – The type of the CLB

ClusterPortId create_port(const ClusterBlockId blk_id, const std::string name, BitIndex width, PortType type)
  Create or return an existing port in the netlist.

Parameters
- blk_id – The block the port is associated with
- name – The name of the port (must match the name of a port in the block’s model)
- width – The width (number of bits) of the port
- type – The type of the port (INPUT, OUTPUT, or CLOCK)

ClusterPinId create_pin(const ClusterPortId port_id, BitIndex port_bit, const ClusterNetId net_id, const PinType pin_type, int pin_index, bool is_const = false)
  Create or return an existing pin in the netlist.

Parameters
- port_id – The port this pin is associated with
• **port_bit** – The bit index of the pin in the port
• **net_id** – The net the pin drives/sinks
• **pin_type** – The type of the pin (driver/sink)
• **pin_index** – The index of the pin relative to its block, excluding OPEN pins)
• **is_const** – Indicates whether the pin holds a constant value (e.g. vcc/gnd)

ClusterNetId `create_net(const std::string name)`
Create an empty, or return an existing net in the netlist.

Parameters: `name` – The unique name of the net

void `set_net_is_ignored(ClusterNetId net_id, bool state)`
Sets the flag in net_ignored_ = state.

void `set_net_is_global(ClusterNetId net_id, bool state)`
Sets the flag in net_is_global_ = state.

### 16.2.3 Atom Netlist

**Overview**

The **AtomNetlist** is derived from the **Netlist** class, and contains information on the primitives. This includes basic components (Blocks, Ports, Pins, & Nets), and physical descriptions (t_model) of the primitives.

Most of the functionality relevant to components and their accessors/cross-accessors is implemented in the **Netlist** class. Refer to netlist.(h|tpp) for more information.

**Components**

There are 4 components in the **Netlist**: Blocks, Ports, Pins, and Nets. Each component has a unique ID in the netlist, as well as various associations to their related components (e.g. A pin knows which port it belongs to, and what net it connects to)

**Blocks**

Blocks refer to the atoms (AKA primitives) that are in the the netlist. Each block contains input/output/clock ports. Blocks have names, and various functionalities (LUTs, FFs, RAMs, …) Each block has an associated t_model, describing the physical properties.

**Ports**

Ports are composed of a set of pins that have specific directionality (INPUT, OUTPUT, or CLOCK). The ports in the **AtomNetlist** are respective to the atoms. (i.e. the **AtomNetlist** does not contain ports of a Clustered Logic Block). Each port has an associated t_model_port, describing the physical properties.

**Pins**

Pins are single-wire input/outputs. They are part of a port, and are connected to a single net.

**Nets**

Nets in the **AtomNetlist** track the wiring connections between the atoms.

**Models**

There are two main models, the primitive itself (t_model) and the ports of that primitive (t_model_ports). The models are created from the architecture file, and describe the physical properties of the atom.

**Truth Table**

The **AtomNetlist** also contains a TruthTable for each block, which indicates what the LUTs contain.
Implementation

For all create_* functions, the AtomNetlist will wrap and call the Netlist’s version as it contains additional information that the base Netlist does not know about.

All functions with suffix *_impl() follow the Non-Virtual Interface (NVI) idiom. They are called from the base Netlist class to simplify pre/post condition checks and prevent Fragile Base Class (FBC) problems.

Refer to netlist.h for more information.

class AtomNetlist : public Netlist<AtomBlockId, AtomPortId, AtomPinId, AtomNetId>

Public Functions

AtomNetlist(std::string name = "", std::string id = "")

Constructs a netlist.

Parameters

- name – the name of the netlist (e.g. top-level module)
- id – a unique identifier for the netlist (e.g. a secure digest of the input file)

AtomBlockType block_type(const AtomBlockId id) const

Returns the type of the specified block.

const t_model *block_model(const AtomBlockId id) const

Returns the model associated with the block.

const TruthTable &block_truth_table(const AtomBlockId id) const

Returns the truth table associated with the block.

For LUTs the truth table stores the single-output cover representing the logic function.

For FF/Latches there is only a single entry representing the initial state

Note: This is only non-empty for LUTs and Flip-Flops/latches.

const t_model_ports *port_model(const AtomPortId id) const

Returns the model port of the specified port or nullptr if not.

Parameters

- id – The ID of the port to look for

AtomPortId find_atom_port(const AtomBlockId blk_id, const t_model_ports *model_port) const

Returns the AtomPortId of the specified port if it exists or AtomPortId::INVALID() if not.

Note: This method is typically more efficient than searching by name

Parameters

- blk_id – The ID of the block who’s ports will be checked
- model_port – The port model to look for
AtomBlockId **find_atom_pin_driver**(const AtomBlockId blk_id, const t_model_ports *model_port, const BitIndex port_bit) const

Returns the AtomBlockId of the atom driving the specified pin if it exists or AtomBlockId::INVALID() if not.

**Parameters**
- **blk_id** – The ID of the block whose ports will be checked
- **model_port** – The port model to look for
- **port_bit** – The pin number in this port

```cpp
std::unordered_set<std::string> net_aliases(const std::string net_name) const
```

Returns the a set of aliases relative to the net name.

If no aliases are found, returns a set with the original net name.

**Parameters**
- **net_name** – name of the net from which the aliases are extracted

AtomBlockId **create_block**(const std::string name, const t_model *model, const TruthTable truth_table = TruthTable())

Create or return an existing block in the netlist.

**Parameters**
- **name** – The unique name of the block
- **model** – The primitive type of the block
- **truth_table** – The single-output cover defining the block’s logic function. The truth_table is optional and only relevant for LUTs (where it describes the logic function) and Flip-Flops/latches (where it consists of a single entry defining the initial state).

AtomPortId **create_port**(const AtomBlockId blk_id, const t_model_ports *model_port)

Create or return an existing port in the netlist.

**Parameters**
- **blk_id** – The block the port is associated with
- **model_port** – The model port the port is associated with

AtomPinId **create_pin**(const AtomPortId port_id, BitIndex port_bit, const AtomNetId net_id, const PinType pin_type, bool is_const = false)

Create or return an existing pin in the netlist.

**Parameters**
- **port_id** – The port this pin is associated with
- **port_bit** – The bit index of the pin in the port
- **net_id** – The net the pin drives/sinks
- **pin_type** – The type of the pin (driver/sink)
- **is_const** – Indicates whether the pin holds a constant value (e. g. vcc/gnd)

AtomNetId **create_net**(const std::string name)

Create an empty, or return an existing net in the netlist.

**Parameters**
- **name** – The unique name of the net

AtomNetId **add_net**(const std::string name, AtomPinId driver, std::vector<AtomPinId> sinks)

Create a completely specified net from specified driver and sinks.
Parameters

- **name** – The name of the net (Note: must not already exist)
- **driver** – The net’s driver pin
- **sinks** – The net’s sink pins

```cpp
void add_net_alias(const std::string net_name, std::string alias_net_name)
```

Adds a value to the net aliases set for a given net name in the net_aliases_map.
If there is no key/value pair in the net_aliases_map, creates a new set and adds it to the map.

Parameters

- **net_name** – The net to be added to the map
- **alias_net_name** – The alias of the assigned clock net id

### 16.3 Routing Resource Graph

#### 16.3.1 RRGraphView

class RRGraphView

**Public Functions**

```cpp
inline size_t num_nodes() const
```

Return number of nodes. This function is inlined for runtime optimization.

```cpp
inline bool empty() const
```

Is the RR graph currently empty?

```cpp
inline vtr::StrongIdRange<RREdgeId> edge_range(RRNodeId id) const
```

Returns a range of RREdgeId’s belonging to RRNodeId id. If this range is empty, then RRNodeId id has no edges.

```cpp
inline t_rr_type node_type(RRNodeId node) const
```

Get the type of a routing resource node. This function is inlined for runtime optimization.

```cpp
inline const char *node_type_string(RRNodeId node) const
```

Get the type string of a routing resource node. This function is inlined for runtime optimization.

```cpp
inline short node_capacity(RRNodeId node) const
```

Get the capacity of a routing resource node. This function is inlined for runtime optimization.

```cpp
inline Direction node_direction(RRNodeId node) const
```

Get the direction of a routing resource node. This function is inlined for runtime optimization. Direction::INC: wire driver is positioned at the low-coordinate end of the wire. Direction::DEC: wire_driver is positioned at the high-coordinate end of the wire. Direction::BIDIR: wire has multiple drivers, so signals can travel either way along the wire Direction::NONE: node does not have a direction, such as IPIN/OPIN.

```cpp
inline const std::string &node_direction_string(RRNodeId node) const
```

Get the direction string of a routing resource node. This function is inlined for runtime optimization.

```cpp
inline float node_C(RRNodeId node) const
```

Get the capacitance of a routing resource node. This function is inlined for runtime optimization.
inline float node_R(RRNodeId node) const
    Get the resistance of a routing resource node. This function is inlined for runtime optimization.

inline int16_t node_rc_index(RRNodeId node) const
    Get the rc_index of a routing resource node. This function is inlined for runtime optimization.

inline t_edge_size node_fan_in(RRNodeId node) const
    Get the fan in of a routing resource node. This function is inlined for runtime optimization.

inline short node_xlow(RRNodeId node) const
    Get the minimum x-coordinate of a routing resource node. This function is inlined for runtime optimization.

inline short node_xhigh(RRNodeId node) const
    Get the maximum x-coordinate of a routing resource node. This function is inlined for runtime optimization.

inline short node_ylow(RRNodeId node) const
    Get the minimum y-coordinate of a routing resource node. This function is inlined for runtime optimization.

inline short node_yhigh(RRNodeId node) const
    Get the maximum y-coordinate of a routing resource node. This function is inlined for runtime optimization.

inline RREdgeId node_first_edge(RRNodeId node) const
    Get the first out coming edge of resource node. This function is inlined for runtime optimization.

inline RREdgeId node_last_edge(RRNodeId node) const
    Get the last out coming edge of resource node. This function is inlined for runtime optimization.

inline int node_length(RRNodeId node) const
    Get the length (number of grid tile units spanned by the wire, including the endpoints) of a routing resource node. 
    node_length() only applies to CHANX or CHANY and is always a positive number. 
    This function is inlined for runtime optimization.

inline bool node_is_initialized(RRNodeId node) const
    Check if routing resource node is initialized. This function is inlined for runtime optimization.

inline bool nodes_are_adjacent(RRNodeId chanx_node, RRNodeId chany_node) const
    Check if two routing resource nodes are adjacent (must be a CHANX and a CHANY). 
    This function is used for error checking; it checks if two nodes are physically adjacent (could be connected) based on 
    their geometry. It does not check the routing edges to see if they are, in fact, possible to connect in the current 
    routing graph. This function is inlined for runtime optimization.

inline bool node_is_inside_bounding_box(RRNodeId node, vtr::Rect<int> bounding_box) const
    Check if node is within bounding box. To return true, the RRNode must be completely contained within 
    the specified bounding box, with the edges of the bounding box being inclusive. This function is inlined 
    for runtime optimization.

inline bool x_in_node_range(int x, RRNodeId node) const
    Check if x is within x-range spanned by the node, inclusive of its endpoints. This function is inlined for 
    runtime optimization.

inline bool y_in_node_range(int y, RRNodeId node) const
    Check if y is within y-range spanned by the node, inclusive of its endpoints. This function is inlined for 
    runtime optimization.

inline const std::string node_coordinate_to_string(RRNodeId node) const
    Get string of information about routing resource node. The string will contain the following information. 
    type, side, x_low, x_high, y_low, y_high, length, direction, segment_name. 
    This function is inlined for runtime optimization.
runtime optimization.

inline bool \texttt{is\_node\_on\_specific\_side}(\texttt{RRNodeId node}, \texttt{e\_side side}) \const
Check whether a routing node is on a specific side. This function is inlined for runtime optimization.

inline const char *\texttt{node\_side\_string}(\texttt{RRNodeId node}) \const
Get the side string of a routing resource node. This function is inlined for runtime optimization.

inline short \texttt{edge\_switch}(\texttt{RRNodeId id}, \texttt{t\_edge\_size iedge}) \const
Get the switch id that represents the iedge’th outgoing edge from a specific node TODO: We may need to revisit this API and think about higher level APIs, like \texttt{switch\_delay()}

inline \texttt{RRNodeId edge\_sink\_node}(\texttt{RRNodeId id}, \texttt{t\_edge\_size iedge}) \const
Get the destination node for the iedge’th edge from specified RRNodeId. This method should generally not be used, and instead first\_edge and last\_edge should be used.

inline bool \texttt{edge\_is\_configurable}(\texttt{RRNodeId id}, \texttt{t\_edge\_size iedge}) \const
Detect if the edge is a configurable edge (controlled by a programmable routing multiplier or a tri-state switch).

inline \texttt{t\_edge\_size num\_configurable\_edges}(\texttt{RRNodeId node}) \const
Get the number of configurable edges. This function is inlined for runtime optimization.

inline \texttt{t\_edge\_size num\_non\_configurable\_edges}(\texttt{RRNodeId node}) \const
Get the number of non-configurable edges. This function is inlined for runtime optimization.

inline \texttt{edge\_idx\_range configurable\_edges}(\texttt{RRNodeId node}) \const
A configurable edge represents a programmable switch between routing resources, which could be a multiplexer a tri-state buffer a pass gate This API gets ID range for configurable edges. This function is inlined for runtime optimization.

inline \texttt{edge\_idx\_range non\_configurable\_edges}(\texttt{RRNodeId node}) \const
A non-configurable edge represents a hard-wired connection between routing resources, which could be a non-configurable buffer that can not be turned off a short metal connection that can not be turned off This API gets ID range for non-configurable edges. This function is inlined for runtime optimization.

inline \texttt{edge\_idx\_range edges}(\texttt{const RRNodeId \&id}) \const
Get outgoing edges for a node. This API is designed to enable range-based loop to walk through the outgoing edges of a node Example: \texttt{RRGraphView rr\_graph; // A dummy rr\_graph for a short example RRNodeId node; // A dummy node for a short example for (RREdgeId edge : rr\_graph.edges(node)) \{ // Do something with the edge \}.}

inline \texttt{t\_edge\_size num\_edges}(\texttt{RRNodeId node}) \const
Get the number of edges. This function is inlined for runtime optimization.

inline short \texttt{node\_ptc\_num}(\texttt{RRNodeId node}) \const
The \texttt{ptc\_num} carries different meanings for different node types (true in VPR RRG that is currently supported, may not be true in customized RRG) CHANX or CHANY: the track id in routing channels OPIN or IPIN: the index of pins in the logic block data structure SOURCE and SINK: the class id of a pin (indicating logic equivalence of pins) in the logic block data structure.

\textbf{Note:} This API is very powerful and developers should not use it unless it is necessary, e.g the node type is unknown. If the node type is known, the more specific routines, \texttt{node\_pin\_num()}, \texttt{node\_track\_num()} and \texttt{node\_class\_num()}, for different types of nodes should be used.

inline short \texttt{node\_pin\_num}(\texttt{RRNodeId node}) \const
Get the pin num of a routing resource node. This is designed for logic blocks, which are IPIN and OPIN nodes. This function is inlined for runtime optimization.
inline short `node_track_num`(RRNodeId node) const
Get the track num of a routing resource node. This is designed for routing tracks, which are CHANX and CHANY nodes. This function is inlined for runtime optimization.

inline short `node_class_num`(RRNodeId node) const
Get the class num of a routing resource node. This is designed for routing source and sinks, which are SOURCE and SINK nodes. This function is inlined for runtime optimization.

inline RRIndexedDataId `node_cost_index`(RRNodeId node) const
Get the cost index of a routing resource node. This function is inlined for runtime optimization.

inline const t_segment_inf & `rr_segments`(RRSegmentId seg_id) const
Return detailed routing segment information with a given id*.

---

**Note:** The routing segments here may not be exactly same as those defined in architecture file. They have been adapted to fit the context of routing resource graphs.

---

inline size_t `num_rr_segments`() const
Return the number of rr_segments in the routing resource graph.

inline const vtr::vector<RRSegmentId, t_segment_inf> & `rr_segments`() const
Return a read-only list of rr_segments for queries from client functions.

inline const t_rr_switch_inf & `rr_switch_inf`(RRSwitchId switch_id) const
Return the switch information that is categorized in the rr_switch_inf with a given id rr_switch_inf is created to minimize memory footprint of RRGraph class. While the RRG could contain millions (even much larger) of edges, there are only a limited number of types of switches. Hence, we use a flyweight pattern to store switch-related information that differs only for types of switches (switch type, drive strength, R, C, etc.). Each edge stores the ids of the switch that implements it so this additional information can be easily looked up.

**Note:** All the switch-related information, such as R, C, should be placed in rr_switch_inf but NOT directly in the edge-related data of RRGraph. If you wish to create a new data structure to represent switches between routing resources, please follow the flyweight pattern by linking your switch ids to edges only!

---

inline size_t `num_rr_switches`() const
Return the number of rr_segments in the routing resource graph.

inline const vtr::vector<RRSwitchId, t_rr_switch_inf> & `rr_switch`() const
Return the rr_switch_inf_ structure for queries from client functions.

inline const RRSpatialLookup & `node_lookup`() const
Return the fast look-up data structure for queries from client functions.

inline const t_rr_graph_storage & `rr_nodes`() const
Return the node-level storage structure for queries from client functions.

inline MetadataStorage<int> `rr_node_metadata_data`() const
.. warning:: The Metadata should stay as an independent data structure than rest of the internal data, e.g., node_lookup!

inline bool `validate_node`(RRNodeId node_id) const
brief Validate that edge data is partitioned correctly

**Note:** This function is used to validate the correctness of the routing resource graph in terms of graph attributes. Strongly recommend to call it when you finish the building a routing resource graph. If you
need more advance checks, which are related to architecture features, you should consider to use the
check_rr_graph() function or build your own check_rr_graph() function.

### 16.3.2 RRGraphBuilder

The builder does not own the storage but it serves a virtual protocol for

- node_storage: store the node list
- node_lookup: store a fast look-up for the nodes

**Note:**

- This is the only data structure allowed to modify a routing resource graph

class **RRGraphBuilder**

**Public Functions**

```cpp
t_rr_graph_storage & rr_nodes()
Return a writable object for rr_nodes.

RRSpatialLookup & node_lookup()
Return a writable object for update the fast look-up of rr_node.

MetadataStorage<int> & rr_node_metadata()
Return a writable object for the meta data on the nodes.

.. warning:: The Metadata should stay as an independent data structure than rest of the internal data, e.g.,
node_lookup!

MetadataStorage<std::tuple<int, int, short>> & rr_edge_metadata()
Return a writable object for the meta data on the edge.

inline size_t rr_node_metadata_size() const
Return the size for rr_node_metadata.

inline size_t rr_edge_metadata_size() const
Return the size for rr_edge_metadata.

inline vtr::flat_map<int, t_metadata_dict>::const_iterator find_rr_node_metadata(const int &lookup_key) const
Find the node in rr_node_metadata.

inline vtr::flat_map<std::tuple<int, int, short int>, t_metadata_dict>::const_iterator find_rr_edge_metadata(const std::tuple<int, int, short int> &lookup_key) const
Find the edge in rr_edge_metadata.

inline vtr::flat_map<int, t_metadata_dict>::const_iterator end_rr_node_metadata() const
Return the last node in rr_node_metadata.
```
inline vtr::flat_map<
  std::tuple<int, int, short int>,
  t_metadata_dict>::const_iterator end_rr_edge_metadata() const

Return the last edge in rr_edge_metadata.

inline RRSegmentId add_rr_segment(const t_segment_inf &segment_info)
Add a rr_segment to the routing resource graph. Return an valid id if successful.

- Each rr_segment contains the detailed information of a routing track, which is denoted by a node in
  CHANX or CHANY type.

It is frequently used by client functions in timing and routability prediction.

inline vtr::vector<RRSegmentId, t_segment_inf> &rr_segments()
Return a writable list of all the rr_segments .. warning:: It is not recommended to use this API unless you
have to. The API may be deprecated later, and future APIs will designed to return a specific data from the
rr_segments.

TODO

inline RRSwitchId add_rr_switch(const t_rr_switch_inf &switch_info)
Add a rr_switch to the routing resource graph. Return an valid id if successful.

- Each rr_switch contains the detailed information of a routing switch interconnecting two routing
  resource nodes.

It is frequently used by client functions in timing prediction.

inline vtr::vector<RRSwitchId, t_rr_switch_inf> &rr_switch()
Return a writable list of all the rr_switches .. warning:: It is not recommended to use this API unless you
have to. The API may be deprecated later, and future APIs will designed to return a specific data from the
rr_switches.

TODO

inline void set_node_type(RRNodeld id, t_rr_type type)
Set the type of a node with a given valid id.

void add_node_to_all_locs(RRNodeld node)
Add an existing rr_node in the node storage to the node look-up.

The node will be added to the lookup for every side it is on (for OPINs and IPINs) and for every (x,y)
location at which it exists (for wires that span more than one (x,y)).

This function requires a valid node which has already been allocated in the node storage, with

- a valid node id
- valid geometry information: xlow/ylow/xhigh/yhigh
- a valid node type
- a valid node ptc number
- a valid side (applicable to OPIN and IPIN nodes only)

void clear()
Clear all the underlying data storage.

void reorder_nodes(e_rr_node_reorder_algorithm reorder_rr_graph_nodes_algorithm, int
  reorder_rr_graph_nodes_threshold, int reorder_rr_graph_nodes_seed)
reorder all the nodes Reordering the rr-graph nodes may be helpful in

16.3. Routing Resource Graph
• Increasing cache locality during routing

• Improving compile time Reorder RRNodeId’s using one of these algorithms:

• DEGREE_BFS: Order by degree primarily, and BFS traversal order secondarily.

• RANDOM_SHUFFLE: Shuffle using the specified seed. Great for testing. The DEGREE_BFS algorithm was selected because it had the best performance of seven existing algorithms here: https://github.com/SymbiFlow/vtr-rrgraph-reordering-tool. It might be worth further research, as the DEGREE_BFS algorithm is simple and makes some arbitrary choices, such as the starting node. The re-ordering algorithm (DEGREE_BFS) does not speed up the router on most architectures vs. using the node ordering created by the rr-graph builder in VPR, so it is off by default. The other use of this algorithm is for some unit tests; by changing the order of the nodes in the rr-graph before routing we check that no code depends on the rr-graph node order Nonetheless, it does improve performance ~7% for the SymbiFlow Xilinx Artix 7 graph.

NOTE: Re-ordering will invalidate any references to rr_graph nodes, so this should generally be called before creating such references.

inline void set_node_capacity(RRNodeId id, short new_capacity)
Set capacity of this node (number of routes that can use it).

inline void set_node_coordinates(RRNodeId id, short x1, short y1, short x2, short y2)
Set the node coordinate.

inline void set_node_ptc_num(RRNodeId id, short new_ptc_num)
The ptc_num carries different meanings for different node types (true in VPR RRG that is currently supported, may not be true in customized RRG) CHANX or CHANY: the track id in routing channels OPIN or IPIN: the index of pins in the logic block data structure SOURCE and SINK: the class id of a pin (indicating logic equivalence of pins) in the logic block data structure.

Note: This API is very powerful and developers should not use it unless it is necessary, e.g. if the node type is unknown. If the node type is known, the more specific routines, set_node_pin_num(), set_node_track_num() and set_node_class_num(), for different types of nodes should be used.

inline void set_node_pin_num(RRNodeId id, short new_pin_num)
set_node_pin_num() is designed for logic blocks, which are IPIN and OPIN nodes

inline void set_node_track_num(RRNodeId id, short new_track_num)
set_node_track_num() is designed for routing tracks, which are CHANX and CHANY nodes

inline void set_node_class_num(RRNodeId id, short new_class_num)
set_node_class_num() is designed for routing source and sinks, which are SOURCE and SINK nodes

inline void set_node_direction(RRNodeId id, Direction new_direction)
Set the node direction; The node direction is only available of routing channel nodes, such as x-direction routing tracks (CHANX) and y-direction routing tracks (CHANY). For other nodes types, this value is not meaningful and should be set to NONE.

inline void reserve_edges(size_t num_edges)
Reserve the lists of edges to be memory efficient. This function is mainly used to reserve memory space inside RRGraph, when adding a large number of edges in order to avoid memory fragments.

inline void emplace_back_edge(RRNodeId src, RRNodeId dest, short edge_switch)
emplace_back_edge; It add one edge. This method is efficient if reserve_edges was called with the number of edges present in the graph.
inline void `emplace_back()`
  Append 1 more RR node to the RR graph.

inline void `alloc_and_load_edges` (const t_rr_edge_info_set *rr_edges_to_create)
  alloc_and_load_edges; It adds a batch of edges.

inline void `set_node_cost_index` (RRNodeId id, RRIndexedDataId new_cost_index)
  set_node_cost_index gets the index of cost data in the list of cost_indexed_data data structure. It contains the routing cost for different nodes in the RRGraph when used in evaluate different routing paths.

inline void `set_node_rc_index` (RRNodeId id, NodeRCIndex new_rc_index)
  Set the rc_index of routing resource node.

inline void `add_node_side` (RRNodeId id, e_side new_side)
  Add the side where the node physically locates on a logic block. Mainly applicable to IPIN and OPIN nodes.

inline void `remap_rr_node_switch_indices` (const t_arch_switch_fanin &switch_fanin)
  It maps arch_switch_index indicies to rr_switch_index indicies.

inline void `mark_edges_as_rr_switch_ids` ()
  Marks that edge switch values are rr switch indicies.

inline size_t `count_rr_switches` (size_t num_arch_switches, t_arch_switch_inf *arch_switch_inf, t_arch_switch_fanin &arch_switch_fanins)
  Counts the number of rr switches needed based on fan in to support mux size dependent switch delays.

inline void `reserve_nodes` (size_t size)
  Reserve the lists of nodes, edges, switches etc. to be memory efficient. This function is mainly used to reserve memory space inside RRGraph, when adding a large number of nodes/edge/switches/segments, in order to avoid memory fragments.

inline void `resize_nodes` (size_t size)
  This function resize node storage to accomidate size RR nodes.

inline void `resize_switches` (size_t size)
  This function resize rr_switch to accomidate size RR Switch.

inline bool `validate` () const
  Validate that edge data is partitioned correctly.

**Note:** This function is used to validate the correctness of the routing resource graph in terms of graph attributes. Strongly recommend to call it when you finish the building a routing resource graph. If you need more advance checks, which are related to architecture features, you should consider to use the check_rr_graph() function or build your own check_rr_graph() function.

inline void `partition_edges` ()
  Sorts edge data such that configurable edges appears before non-configurable edges.

inline void `init_fan_in` ()
  Init per node fan-in data. Should only be called after all edges have been allocated.

**Note:** This is an expensive, O(N), operation so it should be called once you have a complete rr-graph and not called often.
16.3.3 RRSpatialLookup

A data structure built to find the id of an routing resource node (rr_node) given information about its physical position and type. The data structure is mostly needed during building a routing resource graph.

The data structure allows users to

- Update the look-up with new nodes
- Find the id of a node with given information, e.g., x, y, type etc.

class RRSpatialLookup

Public Functions

RRNodeId find_node(int x, int y, t_rr_type type, int ptc, e_side side = NUM_SIDES) const

Returns the index of the specified routing resource node.

This routine also performs error checking to make sure the node in question exists.

Note: All ptc's start at 0 and are positive. Depending on what type of resource this is, ptc can be

- the class number of a common SINK/SOURCE node of grid, starting at 0 and go up to class_inf
  size - 1 of SOURCES + SINKs in a grid
- pin number of an input/output pin of a grid. They would normally start at 0 and go to the number of
  pins on a block at that (x, y) location
- track number of a routing wire in a channel. They would normally go from 0 to channel_width - 1
  at that (x,y)

Note: An invalid id will be returned if the node does not exist

Note: For segments (CHANX and CHANY) of length > 1, the segment is given an rr_index based on the
(x,y) location at which it starts (i.e. lowest (x,y) location at which this segment exists).

Note: The ‘side’ argument only applies to IPIN/OPIN types, and specifies which side of the grid tile the
node should be located on. The value is ignored for non-IPIN/OPIN types

Parameters

- (x, y) – are the grid location within the FPGA
- rr_type – specifies the type of resource,
- ptc – gives a unique number of resources of that type (e.g. CHANX) at that (x,y).
\begin{verbatim}
std::vector<RRNodeId> find_channel_nodes(int x, int y, t_rr_type type) const
Returns the indices of the specified routing resource nodes, representing routing tracks in a channel.

\textbf{Note:}
- Return an empty list if there are no routing channel at the given (x, y) location
- The node list returned only contain valid ids For example, if the 2nd routing track does not exist in a routing channel at (x, y) location, while the 3rd routing track does exist in a routing channel at (x, y) location, the node list will not contain the node for the 2nd routing track, but the 2nd element in the list will be the node for the 3rd routing track

\textbf{Parameters}
- (x, y) – are the coordinate of the routing channel within the FPGA
- rr_type – specifies the type of routing channel, either x-direction or y-direction

std::vector<RRNodeId> find_nodes_at_all_sides(int x, int y, t_rr_type rr_type, int ptc) const
Like \texttt{find_node()} but returns all matching nodes on all the sides.
This is particularly useful for getting all instances of a specific IPIN/OPIN at a specific grid tile (x,y) location.

std::vector<RRNodeId> find_grid_nodes_at_all_sides(int x, int y, t_rr_type rr_type) const
Returns all matching nodes on all the sides at a specific grid tile (x,y) location.
As this is applicable to grid pins, the type of nodes are limited to SOURCE/SINK/IPIN/OPIN

void reserve_nodes(int x, int y, t_rr_type type, int num_nodes, e_side side = SIDES[0])
Reserve the memory for a list of nodes at (x, y) location with given type and side.

void add_node(RRNodeId node, int x, int y, t_rr_type type, int ptc, e_side side = SIDES[0])
Register a node in the fast look-up.

\textbf{Note:} You must have a valid node id to register the node in the lookup

\textbf{Note:} a node added with this call will not create a node in the rr_graph_storage node list You MUST add the node in the rr_graph_storage so that the node is valid

\textbf{Parameters}
- (x, y) – are the coordinate of the node to be indexable in the fast look-up
- type – is the type of a node
- ptc – is a feature number of a node, which can be
  - the class number of a common SINK/SOURCE node of grid,
  - pin index in a tile when type is OPIN/IPIN
  - track index in a routing channel when type is CHANX/CHANY
- side – is the side of node on the tile, applicable to OPIN/IPIN
\end{verbatim}
void `mirror_nodes`\(\) (const `vtr::Point`<int> &src_coord, const `vtr::Point`<int> &des_coord, t\_rr\_type type, e\_side side)

Mirror the last dimension of a look-up, i.e., a list of nodes, from a source coordinate to a destination coordinate.

This function is mostly need by SOURCE and SINK nodes which are indexable in multiple locations. Considering a bounding box \((x, y)\rightarrow(x + \text{width}, y + \text{height})\) of a multi-height and multi-width grid, \text{SOURCE} and SINK nodes are indexable in any location inside the boundry.

An example of usage:

```cpp
// Create a empty lookup
RRSpatialLookup rr_lookup;
// Adding other nodes ...
// Copy the nodes whose types are SOURCE at (1, 1) to (1, 2)
rr_lookup.mirror_nodes(vtr::Point<int>(1, 1),
                        vtr::Point<int>(1, 2),
                        SOURCE,
                        TOP);
```

**Note:** currently this function only accepts \text{SOURCE}/\text{SINK} nodes. May unlock for the other types depending on needs

void `resize_nodes`\(\) (int x, int y, t\_rr\_type type, e\_side side)

Resize the given 3 dimensions \((x, y, \text{side})\) of the `RRSpatialLookup` data structure for the given type.

This function will keep any existing data

**Note:** Strongly recommend to use when the sizes of dimensions are deterministic

void `reorder`\(\) (const `vtr::vector`<RRNodeId, RRNodeId> dest_order)

Reorder the internal look up to be more memory efficient.

void `clear`\(\)

Clear all the data inside.
17.1 IDs - Ranges

17.1.1 vtr_range

namespace vtr

Functions

template<typename T>
auto make_range(T b, T e)
    Creates a vtr::Range from a pair of iterators.
    Unlike using the vtr::Range() constructor (which requires specifying the template type T, using
    vtr::make_range() infers T from the arguments.
    Example usage: auto my_range = vtr::make_range(my_vec.begin(), my_vec.end());

template<typename Container>
auto make_range(const Container &c)
    Creates a vtr::Range from a container.

template<typename T>
class Range
    #include <vtr_range.h> The vtr::Range template models a range defined by two iterators of type T.
    It allows conveniently returning a range from a single function call without having to explicitly expose the
    underlying container, or make two explicit calls to retrieve the associated begin and end iterators. It also
    enables the easy use of range-based-for loops.
    For example:

    class My Data {
        public:
            typedef std::vector<int>::const_iterator my_iter;
            vtr::Range<my_iter> data();
            ...
        private:
            std::vector<int> data_;
    };
MyDat my_data;

//fill my_data
for (int val : my_data.data()) {
    //work with values stored in my_data
}

The `empty()` and `size()` methods are convenience wrappers around the relevant iterator comparisons. Note that `size()` is only constant time if T is a random-access iterator!

### Public Functions

```cpp
inline Range(T b, T e)
    constructor
inline T begin()
    Return an iterator to the start of the range.
inline T end()
    Return an iterator to the end of the range.
inline const T begin() const
    Return an iterator to the start of the range (immutable)
inline const T end() const
    Return an iterator to the end of the range (immutable)
inline bool empty()
    Return true if empty.
inline size_t size()
    Return the range size.
```

### 17.1.2 vtr_strong_id

This header provides the StrongId class.

It is template which can be used to create strong Id’s which avoid accidental type conversions (generating compiler errors when they occur).

**Motivation**

It is common to use an Id (typically an integer) to identify and represent a component. A basic example (poor style):

```cpp
size_t count_net_terminals(int net_id);
```

Where a plain int is used to represent the net identifier. Using a plain basic type is poor style since it makes it unclear that the parameter is an Id.

A better example is to use a typedef:
typedef int NetId;
size_t count_net_terminals(NetId net_id);

It is now clear that the parameter is expecting an Id.

However this approach has some limitations. In particular, typedef's only create type aliases, and still allow conversions. This is problematic if there are multiple types of Ids. For example:

typedef int NetId;
typedef int BlkId;

size_t count_net_terminals(NetId net_id);

BlkId blk_id = 10;
NetId net_id = 42;

count_net_terminals(net_id); //OK
count_net_terminals(blk_id); //Bug: passed a BlkId as a NetId

Since typedefs are aliases the compiler issues no errors or warnings, and silently passes the BlkId where a NetId is expected. This results in hard to diagnose bugs.

We can avoid this issue by using a StrongId:

struct net_id_tag; //Phantom tag for NetId
struct blk_id_tag; //Phantom tag for BlkId

typedef StrongId<net_id_tag> NetId;
typedef StrongId<blk_id_tag> BlkId;

size_t count_net_terminals(NetId net_id);

BlkId blk_id = 10;
NetId net_id = 42;

count_net_terminals(net_id); //OK
count_net_terminals(blk_id); //Compiler Error: NetId expected!

StrongId is a template which implements the basic features of an Id, but disallows silent conversions between different types of Ids. It uses another ‘tag’ type (passed as the first template parameter) to uniquely identify the type of the Id (preventing conversions between different types of Ids).

Usage

The StrongId template class takes one required and three optional template parameters:

1. Tag - the unique type used to identify this type of Ids [Required]
2. T - the underlying integral id type (default: int) [Optional]
3. T sentinel - a value representing an invalid Id (default: -1) [Optional]

If no value is supplied during construction the StrongId is initialized to the invalid/sentinel value.

Example 1: default definition
struct net_id_tag;
typedef StrongId<net_id_tag> NetId; //Internally stores an integer Id, -1 represents invalid

Example 2: definition with custom underlying type

struct blk_id_tag;
typedef StrongId<blk_id_tag, size_t> BlkId; //Internally stores a size_t Id, -1 represents invalid

Example 3: definition with custom underlying type and custom sentinel value

struct pin_id_tag;
typedef StrongId<pin_id_tag, size_t, 0> PinId; //Internally stores a size_t Id, 0 represents invalid

Example 4: Creating Ids

MyId my_id; //Defaults to the sentinel value (-1 by default)
MyId my_other_id = 5; //Explicit construction
MyId my_thrid_id(25); //Explicit construction

Example 5: Comparing Ids

my_id_one == my_id_also_one; //True
my_id_one == my_id; //False
my_id_one == my_id_two; //False
my_id_one != my_id_two; //True

Example 5: Checking for invalid Ids

my_id_one == MyId::INVALID(); //True
my_id_one == MyId::INVALID(); //False
Example 6: Indexing data structures

```cpp
struct my_id_tag;
typedef StrongId<net_id_tag> MyId; //Internally stores an integer Id, -1 represents invalid
std::vector<int> my_vec = {0, 1, 2, 3, 4, 5};
MyId my_id = 2;
my_vec[size_t(my_id)]; //Access the third element via explicit conversion
```

namespace vtr

namespace std

  Specialize std::hash for StrongId's (needed for std::unordered_map-like containers)

template<
  typename tag, 
  typename T = int,
  T sentinel = T(-1)
>
  class vtr::StrongId
  Class template definition with default template parameters.

  **Public Functions**

  inline constexpr StrongId()
  Default to the sentinel value.

  inline explicit constexpr StrongId(T id)
  Only allow explicit constructions from a raw Id (no automatic conversions)

  inline explicit operator bool() const
  Allow explicit conversion to bool (e.g. if(id))

  inline explicit operator std::size_t() const
  Allow explicit conversion to size_t (e.g. my_vector[size_t(strong_id)])
Public Static Functions

static inline constexpr StrongId INVALID()  
   Gets the invalid Id.

Friends

friend bool operator==(const StrongId<tag, T, sentinel>& lhs, const StrongId<tag, T, sentinel>& rhs)  
   To enable comparisons between Ids.
   Note that since these are templated functions we provide an empty set of template parameters after the function name (i.e. <>)

friend bool operator!=(const StrongId<tag, T, sentinel>& lhs, const StrongId<tag, T, sentinel>& rhs)  
   != operator

friend bool operator<(const StrongId<tag, T, sentinel>& lhs, const StrongId<tag, T, sentinel>& rhs)  
   < operator

17.1.3 vtr_strong_id_range

This header defines a utility class for StrongId’s.

StrongId’s are described in vtr_strong_id.h. In some cases, StrongId’s be considered like random access iterators, but not all StrongId’s have this property. In addition, there is utility in refering to a range of id’s, and being able to iterator over that range.

namespace vtr

template<typename StrongId>

class StrongIdIterator
   #include <vtr_strong_id_range.h> StrongIdIterator class.
   StrongIdIterator allows a StrongId to be treated like a random access iterator. Whether this is a correct use of the abstraction is up to the called.

Public Functions

StrongIdIterator() = default  
   constructor

StrongIdIterator &operator=(const StrongIdIterator &other) = default  
   copy constructor

StrongIdIterator(const StrongIdIterator &other) = default  
   copy constructor

inline explicit StrongIdIterator(StrongId id)  
   constructor

inline Strong &operator*()  
   Dereference operator (*)

inline StrongIdIterator &operator+=(ssize_t n)  
   += operator
inline StrongIdIterator &operator-=(ssize_t n)
    -= operator
inline StrongIdIterator &operator++()
    ++ operator
inline StrongIdIterator &operator--()
    Decrement operator.
inline StrongId operator[](ssize_t offset) const
    Indexing operator [].

template<typename StrongId>
class StrongIdRange
#include <vtr_strong_id_range.h> StrongIdRange class.
StrongIdRange allows a pair of StrongId’s to defines a contiguous range of ids. The “end” StrongId is excluded from this range.

Public Functions

inline StrongIdRange(StrongId b, StrongId e)
    constructor
inline StrongIdIterator<StrongId> begin() const
    Returns a StrongIdIterator to the first strongId in the range.
inline StrongIdIterator<StrongId> end() const
    Returns a StrongIdIterator referring to the past-the-end element in the vector container.
inline bool empty()
    Returns true if the range is empty.
inline size_t size()
    Returns the size of the range.

17.2 Containers

17.2.1 vtr_vector

namespace vtr

template<typename K, typename V, typename Allocator = std::allocator<V>>

class vector : private std::vector<V, Allocator>
#include <vtr_vector.h> A std::vector container which is indexed by K (instead of size_t).
The main use of this container is to behave like a std::vector which is indexed by a vtr::StrongId. It assumes that K is explicitly convertable to size_t (i.e. via operator size_t()), and can be explicitly constructed from a size_t.
It includes all the following std::vector functions:
• begin
• cbegin
If you need more std::map-like (instead of std::vector-like) behaviour see \texttt{vtr::vector\_map}.

class \textbf{key\_iterator} : public \texttt{std::iterator<\texttt{std::bidirectional\_iterator\_tag, key\_type>}}

\smallskip
\texttt{#include <vtr\_vector.h>} \texttt{Iterator class which is convertible to the} \texttt{key\_type}.

This allows end-users to call the parent class’s \texttt{keys()} member to iterate through the keys with a range-based for loop.
Public Functions

inline key_iterator(key_iterator::value_type init)
constructor
inline key_iterator operator++()
++ operator
inline key_iterator operator--()
decrement operator
inline reference operator*()
dereference operator
inline pointer operator->()
-> operator

Public Functions

inline V *data()
Returns a pointer to the vector’s data.
inline const V *data() const
Returns a pointer to the vector’s data (immutable)
inline reference operator[](const key_type id)
[] operator
inline const_reference operator[](const key_type id) const
[] operator immutable
inline reference at(const key_type id)
at() operator
inline const_reference at(const key_type id) const
at() operator immutable
inline void swap(vector<K, V, Allocator> &other)
swap function
inline key_range keys() const
Returns a range containing the keys.

17.2.2 vtr_small_vector

template<class T, class S = uint32_t>
class vtr::small_vector
vtr::small_vector is a std::vector like container which:

• consumes less memory: sizeof(vtr::small_vector) < sizeof(std::vector)
• possibly stores elements in-place (i.e. within the object)
On a typical LP64 system a vtr::small_vector consumes 16 bytes by default and supports vectors up to \(2^{32}\) elements long, while a std::vector consumes 24 bytes and supports up to \(2^{64}\) elements. The type used to store the size and capacity is configurable, and set by the second template parameter argument. Setting it to size_t will replicate std::vector’s characteristics.
For short vectors vtr::small_vector will try to store elements in-place (i.e. within the vtr::small_vector object) instead of dynamically allocating an array (by re-using the internal storage for the pointer, size and capacity). Whether this is possible depends on the size and alignment requirements of the value type, as compared to vtr::small_vector. If in-place storage is not possible (e.g. due to a large value type, or a large number of elements) a dynamic buffer is allocated (similar to std::vector).

This is a highly specialized container. Unless you have specifically measured it’s usefulness you should use std::vector.

**Public Functions**

inline small_vector()  
constructor

inline small_vector(size_type nelem)  
constructor

inline const_iterator begin() const  
Return a const_iterator to the first element.

inline const_iterator end() const  
Return a const_iterator pointing to the past-the-end element in the container.

inline const_reverse_iterator rbegin() const  
Return a const_reverse_iterator pointing to the last element in the container (i.e., its reverse beginning).

inline const_reverse_iterator rend() const  
Return a const_reverse_iterator pointing to the theoretical element preceding the first element in the container (which is considered its reverse end).

inline const_iterator cbegin() const  
Return a const_iterator pointing to the first element in the container.

inline const_iterator cend() const  
a const_iterator pointing to the past-the-end element in the container.

inline const_reverse_iterator crbegin() const  
Return a const_reverse_iterator pointing to the last element in the container (i.e., its reverse beginning).

inline const_reverse_iterator crend() const  
Return a const_reverse_iterator pointing to the theoretical element preceding the first element in the container (which is considered its reverse end).

inline size_type size() const  
return the vector size (Padding ensures long/short format sizes are always aligned)

inline size_t max_size() const  
Return the maximum size.

inline size_type capacity() const  
Return the vector capacity.

inline bool empty() const  
Return true if empty.

inline const_reference operator[](size_t i) const  
Immutable indexing operator [].

inline const_reference at(size_t i) const  
Immutable at() operator.
inline const_reference front() const
    Return a constant reference to the first element.
inline const_reference back() const
    Return a constant reference to the last element.
inline const_pointer data() const
    Return a constant pointer to the vector data.
inline iterator begin()
    Return an iterator pointing to the first element in the sequence.
inline iterator end()
    Return an iterator referring to the past-the-end element in the vector container.
inline reverse_iterator rbegin()
    Return a reverse iterator pointing to the last element in the vector (i.e., its reverse beginning).
inline reverse_iterator rend()
    Return a reverse iterator pointing to the theoretical element preceding the first element in the vector (which is considered its reverse end).
inline void resize(size_type n)
    Resizes the container so that it contains n elements.
inline void resize(size_type n, value_type val)
    Resizes the container so that it contains n elements and fills it with val.
inline void reserve(size_type num_elems)
    Reserve memory for a specific number of elements.
    Don’t change capacity unless requested number of elements is both:
    • More than the short capacity (no need to reserve up to short capacity)
    • Greater than the current size (capacity can never be below size)
inline void shrink_to_fit()
    Requests the container to reduce its capacity to fit its size.
inline reference operator[](size_t i)
    Indexing operator [].
inline reference at(size_t i)
    
    at() operator
    inline reference front()
    Returns a reference to the first element in the vector.
inline reference back()
    Returns a reference to the last element in the vector.

template<class InputIterator>
inline void assign(InputIterator first, InputIterator last)
    Assigns new contents to the vector, replacing its current contents, and modifying its size accordingly.
    Input iterators to the initial and final positions in a sequence. The range used is [first,last), which includes all the elements between first and last, including the element pointed by first but not the element pointed by last.
inline void assign(size_type n, const value_type &val)
    Assigns new contents to the vector, replacing its current contents, and modifying its size accordingly.
    Resize the vector to n and fill it with val
Assigns new contents to the vector, replacing its current contents, and modifying its size accordingly.

The compiler will automatically construct such objects from initializer list declarators (il)

Construct default value_type at new location.

Removes the last element in the vector, effectively reducing the container size by one.

The vector is extended by inserting new elements before the element at the specified position, effectively increasing the container size by the number of elements inserted.

Insert a new value.

Location of position as an index, which will be unchanged if the underlying storage is reallocated

Insert n elements at position position and fill them with value val.

Removes from the vector a single element (position).

Removes from the vector either a range of elements ([first,last)).

Exchanges the content of the container by the content of x, which is another vector object of the same type.

Sizes may differ.

Removes all elements from the vector (which are destroyed), leaving the container with a size of 0.

Inserts a new element at the end of the vector, right after its current last element. This new element is constructed in place using args as the arguments for its constructor.

This function serves as a destructor.

A copy constructor.

A copy and swap constructor.

Swaps two vectors.

== operator

< operator
inline friend bool \textbf{operator\nequal{}}(const \texttt{small\_vector<T, S>} \&lhs, const \texttt{small\_vector<T, S>} \&rhs)

\textbf{operator\nequal{}}

inline friend bool \textbf{operator\textgreater{}}(const \texttt{small\_vector<T, S>} \&lhs, const \texttt{small\_vector<T, S>} \&rhs)

\textbf{operator\textgreater{}}

inline friend bool \textbf{operator\textless{}}(const \texttt{small\_vector<T, S>} \&lhs, const \texttt{small\_vector<T, S>} \&rhs)

\textbf{operator\textless{}}

inline friend bool \textbf{operator\greater{}}(const \texttt{small\_vector<T, S>} \&lhs, const \texttt{small\_vector<T, S>} \&rhs)

\textbf{operator\textgreater{}}

\subsection{17.2.3 \texttt{vtr\_vector\_map}}

namespace \texttt{vtr}

template<typename \texttt{K}, typename \texttt{V}, typename \texttt{Sentinel} = \texttt{DefaultSentinel<\texttt{V}>}}
class \texttt{vector\_map}

\texttt{\#include <vtr\_vector\_map.h>} A vector-like container which is indexed by \texttt{K} (instead of \texttt{size\_t} as in \texttt{std::vector}).

The main use of this container is to behave like a \texttt{std::vector} which is indexed by \texttt{vtr::StrongId}.

Requires that \texttt{K} be convertible to \texttt{size\_t} with the \texttt{size\_t} operator (i.e. \texttt{size\_t()}), and that the conversion results in a linearly increasing index into the underlying vector.

This results in a container that is somewhat similar to a \texttt{std::map} (i.e. converts from one type to another), but requires contiguously ascending (i.e. linear) keys. Unlike \texttt{std::map} only the values are stored (at the specified index/key), reducing memory usage and improving cache locality. Furthermore, \texttt{operator[]} and \texttt{find()} return the value or iterator directly associated with the value (like \texttt{std::vector}) rather than a \texttt{std::pair} (like \texttt{std::map}). \texttt{insert()} takes both the key and value as separate arguments and has no return value.

Additionally, \texttt{vector\_map} will silently create values for ‘gaps’ in the index range (i.e. those elements are initialized with \texttt{Sentinel::INVALID()}).

If you need a fully featured \texttt{std::map} like container without the above differences see \texttt{vtr::linear\_map}.

If you do not need \texttt{std::map}-like features see \texttt{vtr::vector}. Note that \texttt{vtr::vector\_map} is very similar to \texttt{vtr::vector}. Unless there is a specific reason that \texttt{vtr::vector\_map} is needed, it is better to use \texttt{vtr::vector}.

Note that it is possible to use \texttt{vector\_map} with sparse/non-contiguous keys, but this is typically memory inefficient as the underlying vector will allocate space for \texttt{[0..size\_t(max\_key)-1]}, where \texttt{max\_key} is the largest key that has been inserted.

As with a \texttt{std::vector}, it is the caller’s responsibility to ensure there is sufficient space when a given index/key before it is accessed. The exception to this are the \texttt{find()}, \texttt{insert()} and \texttt{update()} methods which handle non-existing keys gracefully.

\section{17.2. Containers}
**Public Functions**

template<typename ...Args>
inline vector_map(Args&&... args)
Constructor.

inline const_iterator begin() const
  Returns an iterator referring to the first element in the map container.

inline const_iterator end() const
  Returns an iterator referring to the past-the-end element in the map container.

inline const_reverse_iterator rbegin() const
  Returns a reverse iterator pointing to the last element in the container (i.e., its reverse beginning).

inline const_reverse_iterator rend() const
  Returns a reverse iterator pointing to the theoretical element right before the first element in the map container (which is considered its reverse end).

inline const_reference operator[](const K n) const
  operator immutable

inline const_iterator find(const K key) const
  Searches the container for an element with a key equivalent to k and returns an iterator to it if found, otherwise it returns an iterator to vector_map::end.

inline std::size_t size() const
  Returns the number of elements in the container.

inline bool empty() const
  Returns true if the container is empty.

inline bool contains(const K key) const
  Returns true if the container contains key.

inline size_t count(const K key) const
  Returns 1 if the container contains key, 0 otherwise.

template<typename ...Args>
inline void push_back(Args&&... args)
  push_back function

template<typename ...Args>
inline void emplace_back(Args&&... args)
  emplace_back function

template<typename ...Args>
inline void resize(Args&&... args)
  resize function

inline void clear()
  clears the container

inline size_t capacity() const
  Returns the capacity of the container.

inline void shrink_to_fit()
  Requests the container to reduce its capacity to fit its size.

inline iterator begin()
  Returns an iterator referring to the first element in the map container.
inline iterator **end()**

Returns an iterator referring to the past-the-end element in the map container.

inline reference **operator[](const K n)**

Indexing.

inline iterator **find(const K key)**

Returns an iterator to the first element in the container that compares equal to val. If no such element is found, the function returns **end()**.

inline void **insert(const K key, const V value)**

Extends the container by inserting new elements, effectively increasing the container size by the number of elements inserted.

inline void **update(const K key, const V value)**

Inserts the new key value pair in the container.

### 17.2.4 vtr_linear_map

namespace **vtr**

```cpp
template<class K, class T, class Sentinel = DefaultSentinel<K>>
class linear_map
    #include <vtr_linear_map.h> A std::map-like container which is indexed by K.

    The main use of this container is to behave like a std::map which is optimized to hold mappings between
    a dense linear range of keys (e.g. vtr::StrongId).

    Requires that K be convertible to size_t with the size_t operator (i.e. size_t()), and that the conversion
    results in a linearly increasing index into the underlying vector. Also requires that K() return the sentinel
    value used to mark invalid entries.

    If you only need to access the value associated with the key consider using vtr::vector_map instead, which
    provides a similar but more std::vector-like interface.

    Note that it is possible to use linear_map with sparse/non-contiguous keys, but this is typically memory
    inefficient as the underlying vector will allocate space for [0..size_t(max_key)-1], where max_key is the
    largest key that has been inserted.

    As with a std::vector, it is the caller’s responsibility to ensure there is sufficient space when a given in-
    dex/key before it is accessed. The exception to this are the find() and insert() methods which handle
    non-existing keys gracefully.
```

**Public Functions**

```cpp
linear_map() = default
    Standard big 5 constructors.

linear_map(const linear_map&) = default

linear_map(linear_map&&) = default
```

```
linear_map &operator=(const linear_map&) = default
```
linear_map &operator=(linear_map&&) = default

inline linear_map(size_t num_keys)

inline iterator begin()
   Return an iterator to the first element.

inline const_iterator begin() const
   Return a constant iterator to the first element.

inline iterator end()
   Return an iterator to the last element.

inline const_iterator end() const
   Return a constant iterator to the last element.

inline reverse_iterator rbegin()
   Return a reverse iterator to the last element.

inline const_reverse_iterator rbegin() const
   Return a constant reverse iterator to the last element.

inline reverse_iterator rend()
   Return a reverse iterator pointing to the theoretical element preceding the first element.

inline const_reverse_iterator rend() const
   Return a constant reverse iterator pointing to the theoretical element preceding the first element.

inline const_iterator cbegin() const
   Return a const iterator to the first element.

inline const_iterator cend() const
   Return a const_iterator pointing to the past-the-end element in the container.

inline reverse_iterator crbegin() const
   Return a reverse_iterator pointing to the last element in the container (i.e., its reverse beginning).

inline const_reverse_iterator crend() const
   Return a const_reverse_iterator pointing to the theoretical element preceding the first element in the container (which is considered its reverse end).

inline bool empty() const
   Return true if the container is empty.

inline size_type size() const
   Return the size of the container.

inline size_type max_size() const
   Return the maximum size of the container.

inline mapped_type &operator[](const key_type &key)
   [] operator

inline mapped_type &at(const key_type &key)
   at() operator

inline const mapped_type &at(const key_type &key) const
   constant at() operator

inline std::pair<iterator, bool> insert(const value_type &value)
   Insert value.
template<class InputIterator>
inline void insert(InputIterator first, InputIterator last)
    Insert range.

inline void erase(const key_type &key)
    Erase by key.

inline void erase(const_iterator position)
    Erase at iterator.

inline void erase(const_iterator first, const_iterator last)
    Erase range.

inline void swap(linear_map &other)
    Swap two linear maps.

inline void clear()
    Clear the container.

template<class ...Args>
inline std::pair<iterator, bool> emplace(const key_type &key, Args&&... args)
    Emplace.

inline void reserve(size_type n)
    Requests that the underlying vector capacity be at least enough to contain n elements.

inline void shrink_to_fit()
    Reduces the capacity of the container to fit its size and destroys all elements beyond the capacity.

inline iterator find(const key_type &key)
    Returns an iterator to the first element in the range [first,last) that compares equal to val. If no such
    element is found, the function returns last.

inline const_iterator find(const key_type &key) const
    Returns a constant iterator to the first element in the range [first,last) that compares equal to val. If
    no such element is found, the function returns last.

inline size_type count(const key_type &key) const
    Returns the number of elements in the range [first,last) that compare equal to val.

inline iterator lower_bound(const key_type &key)
    Returns an iterator pointing to the first element in the range [first,last) which does not compare less
    than val.

inline const_iterator lower_bound(const key_type &key) const
    Returns a constant iterator pointing to the first element in the range [first,last) which does not com-
    pare less than val.

inline iterator upper_bound(const key_type &key)
    Returns an iterator pointing to the first element in the range [first,last) which compares greater than
    val.

inline const_iterator upper_bound(const key_type &key) const
    Returns a constant iterator pointing to the first element in the range [first,last) which compares greater
    than val.

inline std::pair<iterator, iterator> equal_range(const key_type &key)
    Returns the bounds of the subrange that includes all the elements of the range [first,last) with values
    equivalent to val.
inline std::pair<const_iterator, const_iterator> equal_range(const key_type &key) const
Returns constant bounds of the subrange that includes all the elements of the range [first,last) with
values equivalent to val.

inline size_type valid_size() const
Return the size of valid elements.

17.2.5 vtr_flat_map

template<class K, class T, class Compare, class Storage>
class vtr::flat_map
flat_map is a (nearly) std::map compatible container
It uses a vector as it’s underlying storage. Internally the stored elements are kept sorted allowing efficient look-up
in O(logN) time via binary search.
This container is typically useful in the following scenarios:
• Reduced memory usage if key/value are small (std::map needs to store pointers to other BST nodes which
can add substantial overhead for small keys/values)
• Faster search/iteration by exploiting data locality (all elements are in contiguous memory enabling better
spatial locality)
The container deviates from the behaviour of std::map in the following important ways:
• Insertion/erase takes O(N) instead of O(logN) time
• Iterators may be invalidated on insertion/erase (i.e. if the vector is reallocated)
The slow insertion/erase performance makes this container poorly suited to maps that frequently add/remove new
keys. If this is required you likely want std::map or std::unordered_map. However if the map is constructed once
and then repeatedly queried, consider using the range or vector-based constructors which initializes the flat_map
in O(NlogN) time.
Subclassed by vtr::flat_map2< K, T, Compare, Storage >

Public Functions

flat_map() = default
Standard constructors.

template<class InputIterator>
inline flat_map(InputIterator first, InputIterator last)
range constructor

inline explicit flat_map(Storage &&values)
direct vector constructor

inline void assign(Storage &&values)
Move the values.
Should be more efficient than the range constructor which must copy each element

inline void assign_sorted(Storage &&values)
By moving the values this should be more efficient than the range constructor which must copy each
element.

inline iterator begin()
Return an iterator pointing to the first element in the sequence:

inline const_iterator begin() const
Return a constant iterator pointing to the first element in the sequence:
inline iterator **end()**
Returns an iterator referring to the past-the-end element in the vector container.

inline const_iterator **end()** const
Returns a constant iterator referring to the past-the-end element in the vector container.

inline reverse_iterator **rbegin()**
Returns a reverse iterator which points to the last element of the map.

inline const_reverse_iterator **rbegin()** const
Returns a constant reverse iterator which points to the last element of the map.

inline reverse_iterator **rend()**
Returns a reverse iterator pointing to the theoretical element preceding the first element in the vector (which is considered its reverse end).

inline const_reverse_iterator **rend()** const
Returns a constant reverse iterator pointing to the theoretical element preceding the first element in the vector (which is considered its reverse end).

inline const_iterator **cbegin()** const
Returns a constant iterator to the first element in the underlying vector.

inline const_iterator **cend()** const
Returns a const_iterator pointing to the past-the-end element in the container.

inline const_reverse_iterator **crbegin()** const
Returns a const_reverse_iterator pointing to the last element in the container (i.e., its reverse beginning).

inline const_reverse_iterator **crend()** const
Returns a const_reverse_iterator pointing to the theoretical element preceding the first element in the container (which is considered its reverse end).

inline bool **empty()** const
Return true if the underlying vector is empty.

inline size_type **size()** const
Return the container size.

inline size_type **max_size()** const
Return the underlying vector’s max size.

inline const mapped_type &**operator[]**(const key_type &key) const
The constant version of operator [].

inline mapped_type &**operator[]**(const key_type &key)
operator []

inline mapped_type &**at**(const key_type &key) const
The constant version of at() operator.

inline std::pair<iterator, bool> **insert**(const value_type &value)
Insert value.

inline std::pair<iterator, bool> **emplace**(const value_type &&value)
Emplace function.

inline iterator **insert**(const_iterator position, const value_type &value)
Insert value with position hint.
inline iterator **emplace**(const_iterator position, const value_type &value)
   Emplace value with position hint.

template<class InputIterator>
inline void **insert**(InputIterator first, InputIterator last)
   Insert range.

inline void **erase**(const key_type &key)
   Erase by key.

inline void **erase**(const_iterator position)
   Erase at iterator.

inline void **erase**(const_iterator first, const_iterator last)
   Erase range.

inline void **swap**(flat_map &other)
   swap two flat maps

inline void **clear**()
   clear the flat map

template<class ...Args>
inline iterator **emplace**(const key_type &key, Args&&... args)
   templated emplace function

template<class ...Args>
inline iterator **emplace_hint**(const_iterator position, Args&&... args)
   templated emplace_hint function

inline void **reserve**(size_type n)
   Reserve a minimum capacity for the underlying vector.

inline void **shrink_to_fit**()
   Reduce the capacity of the underlying vector to fit its size.

inline iterator **find**(const key_type &key)
   Find a key and return an iterator to the found key.

inline const_iterator **find**(const key_type &key) const
   Find a key and return a constant iterator to the found key.

inline size_type **count**(const key_type &key) const
   Return the count of occurrences of a key.

inline iterator **lower_bound**(const key_type &key)
   lower bound function

inline const_iterator **lower_bound**(const key_type &key) const
   Return a constant iterator to the lower bound.

inline iterator **upper_bound**(const key_type &key)
   upper bound function

inline const_iterator **upper_bound**(const key_type &key) const
   Return a constant iterator to the upper bound.

inline std::pair<iterator, iterator> **equal_range**(const key_type &key)
   Returns a range containing all elements equivalent to “key”.

inline std::pair<const_iterator, const_iterator> **equal_range**(const key_type &key) const
   Returns a constant range containing all elements equivalent to “key”.
Friends

inline friend void swap(flat_map &lhs, flat_map &rhs)
Swaps 2 flat maps.

class value_compare
A class to perform the comparison operation for the flat map.

namespace vtr

Functions

template<class K, class V>
flat_map<K, V> make_flat_map(std::vector<std::pair<K, V>> &&vec)
A function to create a flat map.

template<class K, class V>
flat_map2<K, V> make_flat_map2(std::vector<std::pair<K, V>> &&vec)
Same as make_flat_map but for flat_map2.

17.2.6 vtr_bimap

The vtr_bimap.h header provides a bi-directional mapping between key and value which means that it can be addressed by either the key or the value.

It provides this bi-directional feature for all the map-like containers defined in vtr:

- unordered map
- flat map
- linear map

One example where this container might be so useful is the mapping between the atom and clustered net Id. See atom_lookup.h
namespace vtr

template<class K, class V, template<typename...> class Map = std::map, template<typename...> class InvMap = std::map>

class bimap
    #include <vtr_bimap.h> A map-like class which provides a bi-directional mapping between key and value.
    Keys and values can be looked up directly by passing either the key or value. The indexing operator will throw if the key/value does not exist.

Public Functions

inline iterator begin() const
    Return an iterator to the begin of the map.
inline iterator end() const
    Return an iterator to the end of the map.
inline inverse_iterator inverse_begin() const
    Return an iterator to the begin of the inverse map.
inline inverse_iterator inverse_end() const
    Return an iterator to the end of the inverse map.
inline iterator find(const K key) const
    Return an iterator to the key-value pair matching key, or end() if not found.
inline inverse_iterator find(const V value) const
    Return an iterator to the value-key pair matching value, or inverse_end() if not found.
inline const V& operator[](const K key) const
    Return an immutable reference to the value matching key (throw an exception if key is not found)
inline const K& operator[](const V value) const
    Return an immutable reference to the key matching value (throw an exception if value is not found)
inline std::size_t size() const
    Return the number of key-value pairs stored.
inline bool empty() const
    Return true if there are no key-value pairs stored.
inline bool contains(const K key) const
    Return true if the specified key exists.
inline bool contains(const V value) const
    Return true if the specified value exists.
inline void clear()
    Drop all stored key-values.
inline std::pair<iterator, bool> insert(const K key, const V value)
    Insert a key-value pair, if not already in map.
inline void update(const K key, const V value)
    Update a key-value pair, will insert if not already in map.
inline void erase(const K key)
    Remove the specified key (and it’s associated value)
inline void **erase**(const **V** val)
Remove the specified value (and it’s associated key)

**Typedefs**

using **unordered_bimap** = **bimap**<K, V, std::unordered_map, std::unordered_map>

using **flat_bimap** = **bimap**<K, V, vtr::flat_map, vtr::flat_map>

using **linear_bimap** = **bimap**<K, V, vtr::linear_map, vtr::linear_map>

### 17.2.7 vtr_vec_id_set

namespace vtr

template<typename T>
class vec_id_set

#include <vtr_vec_id_set.h> Implements a set-like interface which supports multiple operations.

The supported operations are:

- insertion
- iteration
- membership test all in constant time.

It assumes the element type (T) is convertible to size_t. Usually, elements are vtr::StrongIds.

Iteration through the elements is not strictly ordered, usually insertion order, unless sort() has been previously called.

The underlying implementation uses a vector for element storage (for iteration), and a bit-set for membership tests.

**Public Functions**

inline auto **begin**() const

Returns an iterator to the first element in the sequence.

inline auto **end**() const

Returns an iterator referring to the past-the-end element in the vector container.

inline auto **cbegin**() const

Returns a constant iterator to the first element in the sequence.

inline auto **cend**() const

Returns a constant iterator referring to the past-the-end element in the vector container.

inline bool **insert**(T val)

Insert val in the set.
template<typename Iter>
inline void insert(Iter first, Iter last)
    Iterators specifying a range of elements. Copies of the elements in the range [first,last) are inserted in the container.
inline size_t count(T val) const
    Count elements with a specific value.
inline size_t size() const
    Returns the size of the container.
inline void sort()
    Sort elements in the container.
inline void clear()
    Clears the container

17.2.8 vtr_list

Linked lists of void pointers and integers, respectively.

namespace vtr

    struct t_linked_vptr
    #include <vtr_list.h> Linked list node struct.
    t_linked_vptr *vtr::insert_in_vptr_list(t_linked_vptr *head, void *vptr_to_add)
        Inserts a node to a list.
    t_linked_vptr *vtr::delete_in_vptr_list(t_linked_vptr *head)
        Delete a list.

17.2.9 vtr_ragged_matrix

template<
type T, typename Index0 = size_t, typename Index1 = size_t>

class vtr::FlatRaggedMatrix
A 2 dimensional ‘ragged’ matrix with rows indexed by Index0, and each row of variable length (indexed by Index1)
Example:

    std::vector<int> row_sizes = {1, 5, 3, 10};
    FlatRaggedMatrix<float> matrix(row_sizes);

    //Fill in all entries with ascending values
    float value = 1.;
    for (size_t irow = 0; irow < row_sizes.size(); ++irow) {
        for (size_t icol = 0; icol < row_sizes[irow]; ++icol) {
            matrix[irow][icol] = value;
            value += 1.;
        }
    }
For efficiency, this class uses a flat memory layout, where all elements are laid out contiguously (one row after another).

Expects Index0 and Index1 to be convertable to size_t.

**Public Functions**

\[\text{FlatRaggedMatrix}() = \text{default}\]

default constructor

template<class Callback>
inline \[\text{FlatRaggedMatrix}(\text{size}_t \text{nrows}, \text{Callback} \&\text{row_length_callback}, T \text{ default_value} = T())\]
Constructs matrix with ‘nrows’ rows.

The row length is determined by calling ‘row_length_callback’ with the associated row index.

template<class Container>
inline \[\text{FlatRaggedMatrix}(\text{Container} \text{container}, T \text{ default_value} = T())\]
Constructs matrix from a container of row lengths.

template<class Iter>
inline \[\text{FlatRaggedMatrix}(\text{Iter} \text{row_size_first}, \text{Iter} \text{row_size_last}, T \text{ default_value} = T())\]
Constructs matrix from an iterator range.

The length of the range is the number of rows, and iterator values are the row lengths.

inline auto \[\text{begin}()\]
Iterators to all elements.

inline auto \[\text{end}()\]
Iterator to the last element of the matrix.

inline auto \[\text{begin}() \text{const}\]
Iterator to the first element of the matrix (immutable)

inline auto \[\text{end}() \text{const}\]
Iterator to the last element of the matrix (immutable)

inline size_t \[\text{size}() \text{const}\]
Return the size of the matrix.

inline bool \[\text{empty}() \text{const}\]
Return true if empty.

inline vtr::array_view<T> \[\text{operator[]}(\text{Index0} \text{i})\]
Indexing operators for the first dimension.

inline vtr::array_view<const T> \[\text{operator[]}(\text{Index0} \text{i}) \text{const}\]
Indexing operators for the first dimension (immutable)

inline void \[\text{clear}()\]
Clears the matrix.

inline void \[\text{swap}((\text{FlatRaggedMatrix}<T, \text{Index0}, \text{Index1}> \&\text{other})\]
Swaps two matrices.
Friends

inline friend void swap(FlatRaggedMatrix<T, Index0, Index1> &lhs, FlatRaggedMatrix<T, Index0, Index1> &rhs)

Swaps two matrices.

template<typename U>
class ProxyRow
Proxy class used to represent a ‘row’ in the matrix.

Public Functions

inline ProxyRow(U *first, U *last)
constructor

inline U *begin()
Return iterator to the first element.

inline U *end()
Return iterator to the last element.

inline const U *begin() const
Return iterator to the first element (immutable)

inline const U *end() const
Return iterator to the last element (immutable)

inline size_t size() const
Return the size of the row.

inline U &operator[](Index1 j)
indexing [] operator

inline const U &operator[](Index1 j) const
indexing [] operator (immutable)

inline U *data()
Return iterator to the first element.

inline U *data() const
Return iterator to the first element (immutable)

17.2.10 vtr_ndmatrix

namespace vtr

template<typename T, size_t N>
class NdMatrix : public vtr::NdMatrixBase<T, N>
#include <vtr_ndmatrix.h> An N-dimensional matrix supporting arbitrary (continuous) index ranges per dimension.

Examples:
//A 2-dimensional matrix with indicies [0..4][0..9]
NdMatrix<int,2> m1({5,10});

//Accessing an element
int i = m1[3][5];

//Setting an element
m1[2][8] = 0;

//A 3-dimensional matrix with indicies [0..4][0..9][0..19]
NdMatrix<int,3> m2({5,10,20});

//A 2-dimensional matrix with indicies [0..4][0..9], with all entries
//initialized to 42
NdMatrix<int,2> m3({5,10}, 42);

//Filling all entries with value 101
m3.fill(101);

//Resizing an existing matrix (all values reset to default constructed value)
m3.resize({5,5})

//Resizing an existing matrix (all elements set to value 88)
m3.resize({15,55}, 88)

### Public Functions

inline const NdMatrixProxy<T, N - 1> operator[](size_t index) const
Access an element.

Returns a proxy-object to allow chained array-style indexing (N >= 2 case)

inline NdMatrixProxy<T, N - 1> operator[](size_t index)
Access an element.

Returns a proxy-object to allow chained array-style indexing

template<

type_name T>

class NdMatrix<T, 1> : public vtr::NdMatrixBase<T, 1>
#include <vtr_ndmatrix.h> A 1-dimensional matrix supporting arbitrary (continuous) index ranges per

dimension.

This is considered a specialization for N=1
Public Functions

inline const T &operator[](size_t index) const
Access an element (immutable)

inline T &operator[](size_t index)
Access an element (mutable)

template<typename T, size_t N>
class NdMatrixBase
#include <vtr_ndmatrix.h> Base class for an N-dimensional matrix.

Base class for an N-dimensional matrix supporting arbitrary index ranges per dimension. This class implements all of the matrix handling (lifetime etc.) except for indexing (which is implemented in the NdMatrix class). Indexing is split out to allow specialization (of indexing for N = 1).

Implementation:
This class uses a single linear array to store the matrix in c-style (row major) order. That is, the right-most index is laid out contiguous memory.

This should improve memory usage (no extra pointers to store for each dimension), and cache locality (less indirection via pointers, predictable strides).

The indices are calculated based on the dimensions to access the appropriate elements. Since the indexing calculations are visible to the compiler at compile time they can be optimized to be efficient.

Public Functions

inline NdMatrixBase()
An empty matrix (all dimensions size zero)

inline NdMatrixBase(std::array<size_t, N> dim_sizes, T value = T())
Specified dimension sizes:
[0..dim_sizes[0]) [0..dim_sizes[1]) … with optional fill value

inline size_t size() const
Returns the size of the matrix (number of elements)

inline bool empty() const
Returns true if there are no elements in the matrix.

inline size_t ndims() const
Returns the number of dimensions (i.e. N)

inline size_t dim_size(size_t i) const
Returns the size of the ith dimension.

inline size_t begin_index(size_t i) const
Returns the starting index of ith dimension.

inline size_t end_index(size_t i) const
Returns the one-past-the-end index of the ith dimension.

inline const T &get(size_t i) const
const Flat accessors of NdMatrix

inline T &get(size_t i)
Flat accessors of NdMatrix.
inline void **fill**(T value)
Set all elements to ‘value’.

inline void **resize**(std::array<size_t, N> dim_sizes, T value = T())
Resize the matrix to the specified dimension ranges.
If ‘value’ is specified all elements will be initialized to it, otherwise they will be default constructed.

inline void **clear**()
Reset the matrix to size zero.

inline **NdMatrixBase**(const NdMatrixBase &other)
Copy constructor.

inline **NdMatrixBase**(NdMatrixBase &&other)
Move constructor.

inline NdMatrixBase & operator=(NdMatrixBase rhs)
Copy/move assignment.
Note that rhs is taken by value (copy-swap idiom)

template<typename T, size_t N>
class NdMatrixProxy
#include <vtr_ndmatrix.h> Proxy class for a sub-matrix of a NdMatrix class.
This is used to allow chaining of array indexing [] operators in a natural way.
Each instance of this class peels off one-dimension and returns a NdMatrixProxy representing the resulting sub-matrix. This is repeated recursively until we hit the 1-dimensional base-case.
Since this expansion happens at compiler time all the proxy classes get optimized away, yielding both high performance and generality.
Recursive case: N-dimensional array

**Public Functions**

inline NdMatrixProxy(const size_t *dim_sizes, const size_t *dim_strides, T *start)
Construct a matrix proxy object.

Parameters

- **dim_sizes** – Array of dimension sizes
- **idim** – The dimension associated with this proxy
- **dim_stride** – The stride of this dimension (i.e. how many elements in memory between indices of this dimension)
- **start** – Pointer to the start of the sub-matrix this proxy represents

NdMatrixProxy<T, N> & operator=(const NdMatrixProxy<T, N> &other) = delete

inline const NdMatrixProxy<T, N - 1> operator[](size_t index) const
const [] operator

inline NdMatrixProxy<T, N - 1> operator[](size_t index)
[[]] operator

template<typename T>
class *NdMatrixProxy<T, 1>*

```
#include <vtr_ndmatrix.h>  // Base case: 1-dimensional array.
```

**Public Functions**

```
inline NdMatrixProxy(const size_t *dim_sizes, const size_t *dim_stride, T *start)
```
Construct a 1-d matrix proxy object.

**Parameters**

- **dim_sizes** – Array of dimension sizes
- **dim_stride** – The stride of this dimension (i.e. how many element in memory between indicies of this dimension)
- **start** – Pointer to the start of the sub-matrix this proxy represents

```
NdMatrixProxy<T, 1> &operator=(const NdMatrixProxy<T, 1> &other) = delete
```

```
inline const T &operator[](size_t index) const
const [] operator
```
```
inline T &operator[](size_t index)
[] operator
```
```
inline const T *data() const
```
Backward compitability.

For legacy compatibility (i.e. code expecting a pointer) we allow this base dimension case to retrieve a raw pointer to the last dimension elements.

Note that it is the caller’s responsibility to use this correctly; care must be taken not to clobber elements in other dimensions.

```
inline T *data()
```
same as above but allow update the value

**Typedefs**

```
using Matrix = NdMatrix<T, 2>
```
Convenient short forms for common NdMatricies.

### 17.2.11 *vtr_ndoffsetmatrix*

```
namespace vtr
```

```
class DimRange
#include <vtr_ndoffsetmatrix.h>  // A half-open range specification for a matrix dimension [begin_index, last_index)
```
It comes with valid indicies from `begin_index() ... end_index()-1`, provided `size() > 0.`
Public Functions

**DimRange()** = default
default constructor

inline **DimRange(size_t begin, size_t end)**
a constructor with begin_index, end_index

inline size_t **begin_index()** const
Return the begin index.

inline size_t **end_index()** const
Return the end index.

inline size_t **size()** const
Return the size.

template<typename T, size_t N>
class NdOffsetMatrix: public vtr::NdOffsetMatrixBase<T, N>

#include<vtr_ndoffsetmatrix.h> An N-dimensional matrix supporting arbitrary (continuous) index ranges per dimension.

If no second template parameter is provided defaults to a 2-dimensional matrix

Examples:

```cpp
// A 2-dimensional matrix with indices [0..4][0..9]
NdOffsetMatrix<int, 2> m1({5,10});

// Accessing an element
int i = m4[3][5];

// Setting an element
m4[6][20] = 0;

// A 2-dimensional matrix with indices [2..6][5..9]
// Note that C++ requires one more set of curly brace than you would expect
NdOffsetMatrix<int, 2> m2({{{2,7},{5,10}}});

// A 3-dimensional matrix with indices [0..4][0..9][0..19]
NdOffsetMatrix<int, 3> m3({5,10,20});

// A 3-dimensional matrix with indices [2..6][1..19][50..89]
NdOffsetMatrix<int, 3> m4({{{2,7}, {1,20}, {50,90}}});

// A 2-dimensional matrix with indices [2..6][1..20], with all entries
// initialized to 42
NdOffsetMatrix<int, 2> m4({{{2,7}, {1,21}}}, 42);

// A 2-dimensional matrix with indices [0..4][0..9], with all entries
// initialized to 42
NdOffsetMatrix<int, 2> m1({5,10}, 42);

// Filling all entries with value 101
m1.fill(101);
```

(continues on next page)
public:

inline const NdOffsetMatrixProxy< T, 1 - 1 > operator[]( size_t index ) const
Access an element.

Returns a proxy-object to allow chained array-style indexing (N >= 2 case) template<typename T = typename std::enable_if<N >= 2>::type, typename T1 = T>

inline NdOffsetMatrixProxy< T, 1 - 1 > operator[]( size_t index )
Access an element.

Returns a proxy-object to allow chained array-style indexing

template<typename T>
class NdOffsetMatrix< T, 1 > : public vtr::NdOffsetMatrixBase< T, 1 >
#include <vtrNdOffsetMatrix.h> A 1-dimensional matrix supporting arbitrary (continuous) index ranges per dimension.
This is considered a specialization for N=1

public:

inline const T &operator[]( size_t index ) const
Access an element (immutable)

inline T &operator[]( size_t index )
Access an element (mutable)

template<typename T, size_t N>
class NdOffsetMatrixBase
#include <vtrNdOffsetMatrix.h> Base class for an N-dimensional matrix supporting arbitrary index ranges per dimension.

This class implements all of the matrix handling (lifetime etc.) except for indexing (which is implemented in the NdOffsetMatrix class). Indexing is split out to allows specialization of indexing for N = 1.

Implementation:

This class uses a single linear array to store the matrix in c-style (row major) order. That is, the right-most index is laid out contiguous memory.

This should improve memory usage (no extra pointers to store for each dimension), and cache locality (less indirection via pointers, predictable strides).

The indicies are calculated based on the dimensions to access the appropriate elements. Since the indexing calculations are visible to the compiler at compile time they can be optimized to be efficient.
Public Functions

inline NdOffsetMatrixBase()
   An empty matrix (all dimensions size zero)

inline NdOffsetMatrixBase(std::array<size_t, N> dim_sizes, T value = T())
   Specified dimension sizes:
   [0..dim_sizes[0]) [0..dim_sizes[1]) ... with optional fill value

inline NdOffsetMatrixBase(std::array<DimRange, N> dimRanges, T value = T())
   Specified dimension index ranges:
   [dim_ranges[0].begin_index() ... dim_ranges[1].end_index()) [dim_ranges[1].begin_index() ... dim_ranges[1].end_index()) ... with optional fill value

inline size_t size() const
   Returns the size of the matrix (number of elements)

inline bool empty() const
   Returns true if there are no elements in the matrix.

inline size_t ndims() const
   Returns the number of dimensions (i.e. N)

inline size_t dim_size(size_t i) const
   Returns the size of the ith dimension.

inline size_t begin_index(size_t i) const
   Returns the starting index of ith dimension.

inline size_t end_index(size_t i) const
   Returns the one-past-the-end index of the ith dimension.

inline void fill(T value)
   Set all elements to ‘value’.

inline void resize(std::array<size_t, N> dim_sizes, T value = T())
   Resize the matrix to the specified dimensions.
   If ‘value’ is specified all elements will be initialized to it, otherwise they will be default constructed.

inline void resize(std::array<DimRange, N> dim_ranges, T value = T())
   Resize the matrix to the specified dimension ranges.
   If ‘value’ is specified all elements will be initialized to it, otherwise they will be default constructed.

inline void clear()
   Reset the matrix to size zero.

inline NdOffsetMatrixBase(const NdOffsetMatrixBase &other)
   Copy constructor.

inline NdOffsetMatrixBase(NdOffsetMatrixBase &&other)
   Move constructor.

inline NdOffsetMatrixBase &operator=(NdOffsetMatrixBase rhs)
   Copy/move assignment.
   Note that rhs is taken by value (copy-swap idiom)

template<typename T, size_t N>
class NdOffsetMatrixProxy

#include <vtr_ndoffsetmatrix.h> Proxy class for a sub-matrix of a NdOffsetMatrix class.

This is used to allow chaining of array indexing [] operators in a natural way.

Each instance of this class peels off one-dimension and returns a NdOffsetMatrixProxy representing the resulting sub-matrix. This is repeated recursively until we hit the 1-dimensional base-case.

Since this expansion happens at compiler time all the proxy classes get optimized away, yielding both high performance and generality.

Recursive case: N-dimensional array

Public Functions

inline NdOffsetMatrixProxy(const DimRange *dim_ranges, size_t idim, size_t dim_stride, T *start) Construct a matrix proxy object.

dim_ranges: Array of DimRange objects
idim: The dimension associated with this proxy
dim_stride: The stride of this dimension (i.e. how many element in memory between indicies of this dimension)
start: Pointer to the start of the sub-matrix this proxy represents

inline const NdOffsetMatrixProxy operator[](size_t index) const

inline NdOffsetMatrixProxy operator[](size_t index)

template<typename T>

class NdOffsetMatrixProxy<T, 1> Base case: 1-dimensional array.

Public Functions

inline NdOffsetMatrixProxy(const DimRange *dim_ranges, size_t idim, size_t dim_stride, T *start) Construct a matrix proxy object.

• dim_ranges: Array of DimRange objects
• dim_stride: The stride of this dimension (i.e. how many element in memory between indicies of this dimension)
• start: Pointer to the start of the sub-matrix this proxy represents

inline const T &operator[](size_t index) const

inline T &operator[](size_t index)
Typedefs

using OffsetMatrix = NdOffsetMatrix<T, 2>
Convenient short forms for common NdMatricies.

17.2.12 vtr_array_view

template<typename K, typename V>
class vtr::array_view_id: private vtr::array_view<V>
    Implements a fixed length view to an array which is indexed by vtr::StrongId.
    The main use of this container is to behave like a std::span which is indexed by a vtr::StrongId instead of size_t. It assumes that K is explicitly convertable to size_t (i.e. via operator size_t()), and can be explicitly constructed from a size_t.

Public Functions

inline V &operator[](const key_type id) [ ] operator
inline const V &operator[](const key_type id) const
    constant [] operator
inline V &at(const key_type id)
    at() operator
inline const V &at(const key_type id) const
    constant at() operator
inline key_range keys() const
    Returns a range containing the keys.

class key_iterator : public std::iterator<std::bidirectional_iterator_tag, key_type>
    Iterator class which is convertable to the key_type.
    This allows end-users to call the parent class’s keys() member to iterate through the keys with a range-based for loop

Public Types

using my_iter = typename std::iterator<std::bidirectional_iterator_tag, K>
    Intermediate type my_iter.
    We use the intermediate type my_iter to avoid a potential ambiguity for which clang generates errors and warnings
Public Functions

inline key_iterator operator++()
Note.

\textit{vtr::vector} assumes that the key time is convertible to size_t and that all the underlying IDs are zero-based and contiguous. That means we can just increment the underlying Id to build the next key.

inline key_iterator operator--()
decrement the iterator

inline reference operator*()
dereference operator (*)

inline pointer operator->()
-> operator

template<typename T>

class \textit{vtr::array_view}
An array view class to avoid copying data.

Public Functions

inline explicit constexpr array_view()
default constructor

inline explicit constexpr array_view(T *str, size_t size)
A constructor with data initialization.

inline constexpr T &operator[](size_t pos)
[] operator

inline constexpr const T &operator[](size_t pos) const
constant [] operator

inline T &at(size_t pos)
\textit{at}() operator

inline const T &at(size_t pos) const
const \textit{at}() operator

inline constexpr T &front()
get the first element of the array

inline constexpr const T &front() const
get the first element of the array (can’t update it)

inline constexpr T &back()
get the last element of the array

inline constexpr const T &back() const
get the last element of the array (can’t update it)

inline constexpr T *data()
return the underlying pointer

inline constexpr const T *data() const
return the underlying pointer (constant pointer)
inline constexpr size_t \texttt{size}() const noexcept
    return the array size

inline constexpr size_t \texttt{length}() const noexcept
    return the array size

inline constexpr bool \texttt{empty}() const noexcept
    check if the array is empty

inline constexpr T* \texttt{begin}() noexcept
    return a pointer to the first element of the array

inline constexpr const T* \texttt{begin}() const noexcept
    return a constant pointer to the first element of the array

inline constexpr const T* \texttt{cbegin}() const noexcept
    return a constant pointer to the first element of the array

inline constexpr T* \texttt{end}() noexcept
    return a pointer to the last element of the array

inline constexpr const T* \texttt{end}() const noexcept
    return a constant pointer to the last element of the array

inline constexpr const T* \texttt{cend}() const noexcept
    return a constant pointer to the last element of the array

\section{17.2.13 vtr\_string\_view}

\begin{Verbatim}
\texttt{class vtr::string\_view}
\end{Verbatim}

Implements a view to a fixed length string (similar to std::basic\_string\_view).

The underlying string does not need to be NULL terminated.

\subsection{Public Functions}

inline explicit constexpr \texttt{string\_view}()
    constructor

inline explicit \texttt{string\_view}(const char *str)
    constructor

inline explicit constexpr \texttt{string\_view}(const char *str, size_t size)
    constructor

inline constexpr \texttt{string\_view} \&\texttt{operator=}(const \texttt{string\_view} \&view)
    noexcept
    copy constructor

inline constexpr char \texttt{operator[]}(size_t pos) const
    indexing [] operator (immutable)

inline const char \&\texttt{at}(size_t pos) const
    \texttt{aT()} operator (immutable)

inline constexpr const char \&\texttt{front}() const
    Returns the first character of the string.

inline constexpr const char \&\texttt{back}() const
    Returns the last character of the string.
inline constexpr const char *data() const
Returns a pointer to the string data.

inline constexpr size_t size() const noexcept
Returns the string size.

inline constexpr size_t length() const noexcept
Returns the string size.

inline constexpr bool empty() const noexcept
Returns true if empty.

inline constexpr const char *begin() const noexcept
Returns a pointer to the begin of the string.

inline constexpr const char *cbegin() const noexcept
Same as begin()

inline constexpr const char *end() const noexcept
Returns a pointer to the end of the string.

inline constexpr const char *cend() const noexcept
Same as end()

inline void swap(string_view &v) noexcept
Swaps two string views.

inline string_view substr(size_t pos = 0, size_t count = npos)
Returns a newly constructed string object with its value initialized to a copy of a substring of this object.

## 17.2.14 vtr_cache

template<typename CacheKey, typename CacheValue>
class vtr::Cache
An implementation of a simple cache.

### Public Functions

inline void clear()
Clear cache.

inline const CacheValue *get(const CacheKey &key) const
Check if the cache is valid.

Returns the cached value if present and valid. Returns nullptr if the cache is invalid.

inline const CacheValue *set(const CacheKey &key, std::unique_ptr<CacheValue> value)
Update the cache.
17.2.15 vtr_dynamic_bitset

template<typename \texttt{Index} = \texttt{size\_t}, \texttt{typename Storage} = \texttt{unsigned\ int}>

\texttt{class vtr::}\texttt{::dynamic\_bitset}

A container to represent a set of flags either they are set or reset.

It allocates any required length of bit at runtime. It is very useful in bit manipulation

**Public Functions**

inline void \texttt{resize(size\_t size)}

Reize to the determined size.

inline void \texttt{clear()}

Clear all the bits.

inline \texttt{size\_t size()} const

Return the size of the bitset (total number of bits)

inline void \texttt{fill(bool set)}

Fill the whole bitset with a specific value (0 or 1)

inline void \texttt{set(Index index, bool val)}

Set a specific bit in the bit set to a specific value (0 or 1)

inline bool \texttt{get(Index index)} const

Return the value of a specific bit in the bitset.

**Public Static Attributes**

static constexpr \texttt{size\_t kWidth} = \texttt{std::numeric\_limits<Storage>::digits}

Bits in underlying storage.

17.3 Container Utils

17.3.1 vtr_hash

namespace \texttt{vtr}

**Functions**

\texttt{template<class T>}

inline void \texttt{hash\_combine(std::size\_t &seed, const T &v)}

Hashes v and combines it with seed (as in boost)

This is typically used to implement std::hash for composite types.

struct \texttt{hash\_pair}

\texttt{#include \textless \texttt{vtr\_hash.h}\textgreater}
Public Functions

template<class \texttt{T1}, class \texttt{T2}>
inline \texttt{std::size_t} \texttt{operator() (const std::pair<\texttt{T1}, \texttt{T2}> &pair)} const noexcept

17.3.2 \texttt{vtr\_memory}

namespace \texttt{vtr}

template<class \texttt{T}>

struct \texttt{aligned\_allocator}
\
\texttt{#include <vtr\_memory.h>} \texttt{aligned\_allocator} is a STL allocator that allocates memory in an aligned fashion
working if supported by the platform
It is worth noting the C++20 \texttt{std::allocator} does aligned allocations, but C++20 has poor support.

struct \texttt{t\_chunk}
\
\texttt{#include <vtr\_memory.h>} This structure keeps track to chunks of memory
This structure is to keep track of chunks of memory that is being allocated to save overhead when allocating very small memory pieces. For a complete description, please see the comment in chunk_malloc

Functions

template<typename \texttt{Container}>
void \texttt{release\_memory (Container &container)}
\begin{itemize}
\item This function will force the container to be cleared.
\item It release it’s held memory. For efficiency, STL containers usually don’t release their actual heap-allocated memory until destruction (even if Container::clear() is called).
\end{itemize}

template<typename \texttt{T}>
\texttt{T *chunk\_new (t\_chunk *chunk\_info)}
\begin{itemize}
\item Like chunk_malloc, but with proper C++ object initialization.
\end{itemize}

template<typename \texttt{T}>
void \texttt{chunk\_delete (T *obj, t\_chunk*)}
\begin{itemize}
\item Call the destructor of an obj which must have been allocated in the specified chunk.
\end{itemize}

inline int \texttt{memalign (void **ptr\_out, size\_t align, size\_t size)}

template<typename \texttt{T}>
bool \texttt{operator== (const aligned\_allocator<\texttt{T}> &, const aligned\_allocator<\texttt{T}> &)}
\begin{itemize}
\item compare two aligned allocators.
\end{itemize}

Since the allocator doesn’t have any internal state, all allocators for a given type are the same.
17.3.3 vtr_pair_util

namespace vtr

template<typename PairIter>

class pair_first_iter
    #include <vtr_pair_util.h> Iterator which dereferences the ‘first’ element of a std::pair iterator.

    Public Functions
    
    inline pair_first_iter(PairIter init)
        constructor

    inline auto operator++()
        increment operator (++)

    inline auto operator--()
        decrement operator (–)

    inline auto operator*()
        dereference * operator

    inline auto operator->()
        -> operator

template<typename PairIter>

class pair_second_iter
    #include <vtr_pair_util.h> Iterator which dereferences the ‘second’ element of a std::pair iterator

    Public Functions
    
    inline pair_second_iter(PairIter init)
        constructor

    inline auto operator++()
        increment operator (++)

    inline auto operator--()
        decrement operator (–)

    inline auto operator*()
        dereference * operator

    inline auto operator->()
        -> operator
17.3.4 vtr_map_util

namespace vtr

Typedefs

using map_key_iter = pair_first_iter<Iter>
    An iterator who wraps a std::map iterator to return it’s key.

using map_value_iter = pair_second_iter<Iter>
    An iterator who wraps a std::map iterator to return it’s value.

Functions

template<typename T>
    auto make_key_range(T b, T e)
        Returns a range iterating over a std::map’s keys.

template<typename Container>
    auto make_key_range(const Container &c)
        Returns a range iterating over a std::map’s keys.

template<typename T>
    auto make_value_range(T b, T e)
        Returns a range iterating over a std::map’s values.

template<typename Container>
    auto make_value_range(const Container &c)
        Returns a range iterating over a std::map’s values.

17.4 Logging - Errors - Assertions

17.4.1 vtr_log

This header defines useful logging macros for VTR projects.

Message Type

Three types of log message types are defined:

• VTR_LOG : The standard ‘info’ type log message

• VTR_LOG_WARN : A warning log message. This represents unusual condition that may indicate an issue but execution continues

• VTR_LOG_ERROR : An error log message. This represents a clear issue that should result in stopping the program execution. Please note that using this log message will not actually terminate the program. So a VtrError should be thrown after all the neccessary VTR_LOG_ERROR messages are printed.

For example:
Conditional Logging

Each of the three message types also have a VTR_LOGV_* variant, which will cause the message to be logged if a
user-defined condition is satisfied.

For example:

```c
VTR_LOGV(verbosity > 5, "This message will be logged only if verbosity is greater than \%d\n", 5);
VTR_LOGV_WARN(verb, "This warning message will be logged if verb is true\n");
VTR_LOGV_ERROR(false, "This error message will never be logged\n");
```

Custom Location Logging

Each of the three message types also have a VTR_LOGF_* variant, which will cause the message to be logged for a
custom file and

For example:

```c
VTR_LOGF("my_file.txt", "This message will be logged from file 'my_file.txt' line \%d\n", ˓→42);
```

Debug Logging

For debug purposes it may be useful to have additional logging. This is supported by VTR_LOG_DEBUG() and
VTR_LOGV_DEBUG().

To avoid run-time overhead, these are only enabled if VTR_ENABLE_DEBUG_LOGGING is defined (disabled by
default).

namespace vtr

Functions

void add_warnings_to_suppress(std::string function_name)
   The following data structure and functions allow to suppress noisy warnings and direct them into an external file,
   if specified.

void set_noisy_warn_log_file(std::string log_file_name)
   This function creates a new log file to hold the suppressed warnings. If the file already exists, it is cleared out
   first.

void print_or_suppress_warning(const char *pszFileName, unsigned int lineNum, const char *pszFuncName,
   const char *pszMessage, ...)
   This function checks whether to print or to suppress warning.

   This function checks whether the function from which the warning has been called is in the set of warn-
   ings_to_suppress. If so, the warning is printed on the noisy_warn_log_file, otherwise it is printed on stdout
   (or the regular log file)
17.4.2 vtr_error

A utility container that can be used to identify VTR execution errors.

The recommended usage is to store information in this container about the error during an error event and and then throwing an exception with the container. If the exception is not handled (exception is not caught), this will result in the termination of the program.

Error information can be displayed using the information stored within this container.

namespace vtr

class VtrError : public runtime_error

#include <vtr_error.h> Container that holds information related to an error.

It holds different info related to a VTR error:
- error message
- file name associated with the error
- line number associated with the error

Example Usage:

```cpp
// creating and throwing an exception with a VtrError container that has an error occuring in file "error_file.txt" at line number 1
throw vtr::VtrError("This is a program terminating error!", "error_file.txt", 1);
```

Public Functions

inline VtrError(std::string msg = "", std::string new_filename = "", size_t new_linenumber = -1) VtrError constructor.

inline std::string filename() const
gets the filename

Returns the filename associated with this error. Returns an empty string if none is specified.

inline const char *filename_c_str() const
gets the filename

same as filename() but returns in c style string

inline size_t line() const
get the line number

Returns the line number associated with this error. Returns zero if none is specified.
17.4.3 vtr_assertion

The header vtr_assert.h defines useful assertion macros for VTR projects.

Four types of assertions are defined:

<table>
<thead>
<tr>
<th>Assertion</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VTR_ASSERT_OPT</td>
<td>Low overhead assertions that should always be enabled</td>
</tr>
<tr>
<td>VTR_ASSERT</td>
<td>Medium overhead assertions that are usually enabled</td>
</tr>
<tr>
<td>VTR_ASSERT_SAFE</td>
<td>High overhead assertions typically enabled only for debugging</td>
</tr>
<tr>
<td>VTR_ASSERT_DEBUG</td>
<td>Very high overhead assertions typically enabled only for extreme debugging</td>
</tr>
</tbody>
</table>

Each of the above assertions also have a *_MSG variants (e.g. VTR_ASSERT_MSG(expr, msg)) which takes an additional argument specifying additional message text to be shown. By convention the message should state the condition being checked (and not the failure condition), since that the condition failed is obvious from the assertion failure itself.

The macro VTR_ASSERT_LEVEL specifies the level of assertion checking desired and is updated in CMAKE compilation:

<table>
<thead>
<tr>
<th>VTR_ASSERT_LEVEL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>VTR_ASSERT_OPT, VTR_ASSERT, VTR_ASSERT_SAFE, VTR_ASSERT_DEBUG enabled</td>
</tr>
<tr>
<td>3</td>
<td>VTR_ASSERT_OPT, VTR_ASSERT, VTR_ASSERT_SAFE enabled</td>
</tr>
<tr>
<td>2</td>
<td>VTR_ASSERT_OPT, VTR_ASSERT enabled</td>
</tr>
<tr>
<td>1</td>
<td>VTR_ASSERT_OPT enabled</td>
</tr>
<tr>
<td>0</td>
<td>No assertion checking enabled</td>
</tr>
</tbody>
</table>

That an assertion levels beyond 4 are currently treated the same as level 4 and the default assertion level is 2.

17.4.4 vtr_time

class **ScopedStartFinishTimer** : public vtr::ScopedActionTimer

Scoped elapsed time class which prints out the action when initialized and again both the action and elapsed time.

when destructed. For example:

```cpp
{
    vtr::ScopedStartFinishTimer timer("my_action") //Will print: 'my_action'
    //Do other work
    //Will print 'my_action took X.XX seconds' when out of scope
}
```

class **ScopedFinishTimer** : public vtr::ScopedActionTimer

Scoped elapsed time class which prints the time elapsed for the specified action when it is destructed.

For example:

```cpp
{
    vtr::ScopedFinishTimer timer("my_action");
    //Do other work
}
```
class ScopedActionTimer : public vtr::Timer  
Scoped time class which prints the time elapsed for the specific action.  
Subclassed by vtr::ScopedFinishTimer, vtr::ScopedStartFinishTimer

class Timer  
Class for tracking time elapsed since construction.  
Subclassed by vtr::ScopedActionTimer

17.5 Geometry

17.5.1 vtr_geometry

This file include different geometry classes.

template<class T>  
class vtr::Point  
A point in 2D space.  
This class represents a point in 2D space. Hence, it holds both x and y components of the point.

Public Functions

T x() const  
x coordinate

T y() const  
y coordinate

void set(T x_val, T y_val)  
Set x and y values.

void set_x(T x_val)  
set x value

void set_y(T y_val)  
set y value

void swap()  
Swap x and y values.
Friends

friend bool operator==(Point<T> lhs, Point<T> rhs) == operator

friend bool operator!=(Point<T> lhs, Point<T> rhs) != operator

friend bool operator<(Point<T> lhs, Point<T> rhs) < operator

template<class T>

class vtr::Rect

A 2D rectangle.

This class represents a 2D rectangle. It can be created with its 4 points or using the bottom left and the top rights ones only

Public Functions

Rect()

default constructor

Rect(T left_val, T bottom_val, T right_val, T top_val)

construct using 4 vertex

Rect(Point<T> bottom_left_val, Point<T> top_right_val)

construct using the bottom left and the top right vertex

template<typename U = T, typename std::enable_if<std::is_integral<U>::value>::type...>

Rect(Point<U> point)

Constructs a rectangle that only contains the given point.

Rect(p1).contains(p2) => p1 == p2 It is only enabled for integral types, because making this work for floating point types would be difficult and brittle. The following line only enables the constructor if std::is_integral<T>::value == true

T xmin() const

xmin coordinate

T xmax() const

xmax coordinate

T ymin() const

ymin coordinate

T ymax() const

ymax coordinate

Point<T> bottom_left() const

Return the bottom left point.

Point<T> top_right() const

Return the top right point.

T width() const

Return the rectangle width.

T height() const

Return the rectangle height.
bool contains(Point<T> point) const
    Returns true if the point is fully contained within the rectangle (excluding the top-right edges)

bool strictly_contains(Point<T> point) const
    Returns true if the point is strictly contained within the region (excluding all edges)

bool coincident(Point<T> point) const
    Returns true if the point is coincident with the rectangle (including the top-right edges)

bool contains(const Rect<T> &other) const
    Returns true if other is contained within the rectangle (including all edges)

bool empty() const
    Checks whether the rectangle is empty.
    Returns true if no points are contained in the rectangle rect.empty() => not exists p. rect.contains(p) This also implies either the width or height is 0.

void set_xmin(T xmin_val)
    set xmin to a point

void set_ymin(T ymin_val)
    set ymin to a point

void set_xmax(T xmax_val)
    set xmax to a point

void set_ymax(T ymax_val)
    set ymax to a point

Rect<T> &expand_bounding_box(const Rect<T> &other)
    Equivalent to *this = bounding_box(*this, other)

Friends

friend bool operator==(const Rect<T> &lhs, const Rect<T> &rhs)
    == operator

friend bool operator!=(const Rect<T> &lhs, const Rect<T> &rhs)
    != operator

template<class T>
class vtr::Line
    A 2D line.
    It is constructed using a vector of the line points

Public Functions

Line(std::vector<Point<T>> line_points)
    constructor

Rect<T> bounding_box() const
    Returns the bounding box.

point_range points() const
    Returns a range of constituent points.
class **vtr::RectUnion**
A union of 2d rectangles.

**Public Functions**

**RectUnion**(*std::vector<Rect<T>> rects*)
Construct from a set of rectangles.

**Rect<T> bounding_box() const**
Returns the bounding box of all rectangles in the union.

**bool contains(Point<T> point) const**
Returns true if the point is fully contained within the region (excluding top-right edges)

**bool strictly_contains(Point<T> point) const**
Returns true if the point is strictly contained within the region (excluding all edges)

**bool coincident(Point<T> point) const**
Returns true if the point is coincident with the region (including the top-right edges)

**rect_range rects() const**
Returns a range of all constituent rectangles.

**Friends**

friend **bool operator==(const RectUnion<T> &lhs, const RectUnion<T> &rhs)**
Checks whether two RectUnions have identical representations.
Note: does not check whether the representations they are equivalent

friend **bool operator!=(const RectUnion<T> &lhs, const RectUnion<T> &rhs)**
!= operator

### 17.6 Other

**17.6.1 vtr_expr_eval**

This file implements an expression evaluator.

The expression evaluator is capable of performing many operations on given variables, after parsing the expression. The parser goes character by character and identifies the type of char or chars. (e.g bracket, comma, number, operator, variable). The supported operations include addition, subtraction, multiplication, division, finding max, min, gcd, lcm, as well as boolean operators such as &&, ||, ==, >=, <= etc. The result is returned as an int value and operation precedence is taken into account. (e.g given 3-2*4, the result will be -5). This class is also used to parse expressions indicating breakpoints. The breakpoint expressions consist of variable names such as move_num, temp_num, from_block etc, and boolean operators (e.g move_num == 3). Multiple breakpoints can be expressed in one expression...
Functions

BreakpointStateGlobals *get_bp_state_globals()

returns the global variable that holds all values that can trigger a breakpoint and are updated by the router and placer

namespace vtr

Enums

text

enum e_formula_obj
Used to identify the type of symbolic formula object.

Values:

enumerator E_FML_UNDEFINED

denumerator E_FML_NUMBER

denumerator E_FML_BRACKET

denumerator E_FML_COMMA

denumerator E_FML_OPERATOR

denumerator E_FML_VARIABLE

denumerator E_FML_NUM_FORMULA_OBJS

tenum e_operator
Used to identify an operator in a formula.

Values:

enumerator E_OP_UNDEFINED

denumerator E_OP_ADD

denumerator E_OP_SUB
enumerator **E_OP_MULT**

enumerator **E_OP_DIV**

enumerator **E_OP_MIN**

enumerator **E_OP_MAX**

enumerator **E_OP_GCD**

enumerator **E_OP_LCM**

enumerator **E_OP_AND**

enumerator **E_OP_OR**

enumerator **E_OP_GT**

enumerator **E_OP_LT**

enumerator **E_OP_GTE**

enumerator **E_OP_LTE**

enumerator **E_OP_EQ**

enumerator **E_OP_MOD**

enumerator **E_OP_AA**

enumerator **E_OP_NUM_OPS**

**enum e_compound_operator**

Used to identify operators with more than one character.
Values:

enumerator E_COM_OP_UNDEFINED
enumerator E_COM_OP_AND
enumerator E_COM_OP_OR
enumerator E_COM_OP_EQ
enumerator E_COM_OP_AA
enumerator E_COM_OP_GTE
enumerator E_COM_OP_LTE

class vtr::Formula_Object
A class represents an object in a formula.
This object can be any of the following:
• a number
• a bracket
• an operator
• a variable

Public Functions

inline Formula_Object() constructor
inline std::string to_string() const
  convert enum to string

Public Members

t_formula_obj type
  indicates the type of formula object this is

union u_Data
  object data, accessed based on what kind of object this is
Public Members

int num
for number objects

t_operator op
for operator objects

bool left_bracket
for bracket objects & #8212; specifies if this is a left bracket

class vtr::FormulaParser
A class to parse formula.

Public Functions

int parse_formula(std::string formula, const t_formula_data &mydata, bool is_breakpoint = false)
returns integer result according to specified formula and data

int parse_piecewise_formula(const char *formula, const t_formula_data &mydata)
returns integer result according to specified piece-wise formula and data

Public Static Functions

static bool is_piecewise_formula(const char *formula)
checks if the specified formula is piece-wise defined

class vtr::t_formula_data
a class to hold the formula data

Public Functions

inline void clear()
clears all the formula data

inline void set_var_value(vtr::string_view var, int value)
set the value of a specific part of the formula

inline void set_var_value(const char *var, int value)
set the value of a specific part of the formula (the var can be c-style string)

inline int get_var_value(const std::string &var) const
get the value of a specific part of the formula

inline int get_var_value(vtr::string_view var) const
get the value of a specific part of the formula (the var can be c-style string)
17.6.2 vtr_color_map

namespace vtr

template<class T>
struct Color
    #include <vtr_color_map.h> A container to save the rgb components of a color.

class ColorMap
    #include <vtr_color_map.h> A class that holds a complete color map.

Public Functions

ColorMap(float min, float max, const std::vector<Color<float>> &color_data)
    color map constructor

virtual ~ColorMap() = default
    color map destructor

Color<float> color(float value) const
    Returns the full color corresponding to the input value.

float min() const
    Return the min Color of this color map.

float max() const
    Return the max color of this color map.

float range() const
    Return the range of the color map.

class InfernoColorMap : public vtr::ColorMap
    #include <vtr_color_map.h>

Public Functions

InfernoColorMap(float min, float max)

class PlasmaColorMap : public vtr::ColorMap
    #include <vtr_color_map.h>
Public Functions

PlasmaColorMap(float min, float max)

class ViridisColorMap : public vtr::ColorMap
#include <vtr_color_map.h>

Public Functions

ViridisColorMap(float min, float max)

17.6.3 vtr_digest

std::string vtr::secure_digest_file(const std::string &filepath)
Generate a secure hash of the file at filepath.

std::string vtr::secure_digest_stream(std::istream &is)
Generate a secure hash of a stream.

17.6.4 vtr_logic

namespace vtr

Enums

enum LogicValue
This class represents the different supported logic values.
Values:

enumerator FALSE

enumerator TRUE

enumerator DONT_CARE

enumerator UNKNOWN
17.6.5 vtr_math

This file defines some math operations.

namespace vtr

Functions

constexpr int nint(float val)
   Integer rounding conversion for floats.

template<typename T>
T safe_ratio(T numerator, T denominator)
   Returns a ‘safe’ ratio which evaluates to zero if the denominator is zero.

template<typename InputIterator>
double median(InputIterator first, InputIterator last)
   Returns the median of the elements in range [first, last].

template<typename Container>
double median(Container c)
   Returns the median of a whole container.

template<typename InputIterator>
double geomean(InputIterator first, InputIterator last, double init = 1.)
   Returns the geometric mean of the elements in range [first, last]
   To avoid potential round-off issues we transform the standard formula:

   \[
   \text{geomean} = \left( v_1 \times v_2 \times \ldots \times v_n \right)^{1/n}
   \]

   by taking the log:

   \[
   \text{geomean} = \exp\left( \frac{1}{n} \times \left( \log(v_1) + \log(v_2) + \ldots + \log(v_n) \right) \right)
   \]


template<typename Container>
double geomean(Container c)
   Returns the geometric mean of a whole container.

template<typename InputIterator>
double arithmean(InputIterator first, InputIterator last, double init = 0.)
   Returns the arithmetic mean of the elements in range [first, last].

template<typename Container>
double arithmean(Container c)
   Returns the arithmetic mean of a whole container.

template<typename T>
static T gcd(T x, T y)
   Returns the greatest common divisor of x and y.
   Note that T should be an integral type

template<typename T>
T lcm(T x, T y)
   Returns the least common multiple of x and y.
   Note that T should be an integral type
template<class T>
bool isclose(T a, T b, T rel_tol, T abs_tol)
    Return true if a and b values are close to each other.

namespace vtr

Functions

int ipow(int base, int exp)
    Calculates the value pow(base, exp)

float median(std::vector<float> vector)
    Returns the median of an input vector.

template<typename X, typename Y>
    Y linear_interpolate_or_extrapolate(const std::map<X, Y> *xy_map, X requested_x)
    Linear interpolation/Extrapolation.
    Performs linear interpolation or extrapolation on the set of (x,y) values specified by the xy_map. A requested x value is passed in, and we return the interpolated/extrapolated y value at this requested value of x. Meant for maps where both key and element are numbers. This is specifically enforced by the explicit instantiations below this function. i.e. only templates using those types listed in the explicit instantiations below are allowed

template double linear_interpolate_or_extrapolate (const std::map<int, double> *xy_map, int requested_x)

template double linear_interpolate_or_extrapolate (const std::map<double, double> *xy_map, double requested_x)

17.6.6 vtr_ostream_guard

namespace vtr

class OsFormatGuard
#include <vtr_ostream_guard.h> A RAII guard class to ensure restoration of output stream format.
Public Functions

inline explicit OsFormatGuard(std::ostream &os)
constructor

inline ~OsFormatGuard()
destructor

OsFormatGuard(const OsFormatGuard&) = delete

OsFormatGuard &operator=(const OsFormatGuard&) = delete

OsFormatGuard(const OsFormatGuard&&) = delete

OsFormatGuard &operator=(const OsFormatGuard&&) = delete

17.6.7 vtr_path

This file defines some useful utilities to handle paths.

std::array<std::string, 2> vtr::split_ext(const std::string &filename)
Splits off the name and extension (including “.”) of the specified filename.

std::string vtr::basename(const std::string &path)
Returns the basename of path (i.e. the last filename component)
For example, the path “/home/user/my_files/test.blif” -> “test.blif”

std::string vtr::dirname(const std::string &path)
Returns the dirname of path (i.e. everything except the last filename component)
For example, the path “/home/user/my_files/test.blif” -> “/home/user/my_files/”

std::string vtr::.getcwd()
Returns the current working directory.

17.6.8 vtr_random

namespace vtr

Functions

template<typename Iter>
void shuffle(Iter first, Iter last, RandState &rand_state)
Portable/invariant version of std::shuffle.

Note that std::shuffle relies on std::uniform_int_distribution which can produce different sequences accross different compilers/compiler versions.

This version should be deterministic/invariant. However, since it uses vtr::irand(), may not be as well distributed as std::shuffle.
namespace vtr

**Functions**

void **srandom**(int seed)
   The pseudo-random number generator is initialized using the argument passed as seed.

RandState **get_random_state**()
   Return The random number generator state.

int **irand**(int imax, RandState &rand_state)
   Return a randomly generated integer less than or equal imax using the generator (rand_state)

int **irand**(int imax)
   Return a randomly generated integer less than or equal imax.

float **frand**()
   Return a randomly generated float number between [0,1].

**17.6.9 vtr_rusage**

namespace vtr

**Functions**

size_t **get_max_rss**()
   Returns the maximum resident set size in bytes, or zero if unable to determine.

**17.6.10 vtr_sentinels**

This header defines different sentinal value classes.

namespace vtr

    template<class T, T val>
    class **CustomSentinel**
    #include <vtr_sentinels.h> The sentile value is a specified value of the type.
    template<class T>
    class **DefaultSentinel**
    #include <vtr_sentinels.h> The Default sentinal value class.

Some specialized containers like **vtr::linear_map** and **vtr::vector_map** require sentinel values to mark invalid/uninitialized values. By convention, such containers query the sentinel objects static INVALID() member function to retrieve the sentinel value.

These classes allows users to specify a custom sentinel value.

Usually the containers default to **DefaultSentinel**
The sentinel value is the default constructed value of the type

template<class T>

class DefaultSentinel<T*>

#include <vtr_sentinels.h> Specialization for pointer types.

17.6.11 vtr_string_interning

Provides basic string interning, along with pattern splitting suitable for use with FASM.

For reference, string interning refers to keeping a unique copy of a string in storage, and then handing out an id to that storage location, rather than keeping the string around. This deduplicates memory overhead for strings.

This string internment has an additional feature that is splitting the input string into “parts” based on ‘.’, which happens to be the feature separator for FASM. This means the string “TILE.CLB.A” and “TILE.CLB.B” would be made up of the intern ids for {“TILE”, “CLB”, “A”} and {“TILE”, “CLB”, “B”} respectively, allowing some internal deduplication.

Strings can contain up to kMaxParts, before they will be interned as their whole string.

Interned strings (interned_string) that come from the same internment object (string_internment) can safely be checked for equality and hashed without touching the underlying string. Lexigraphical comprisions (e.g. <) requires reconstructing the string.

Basic usage:

1. Create a string_internment

2. Invoke string_internment::intern_string, which returns the interned_string object that is the interned string’s unique identifier. This identifier can be checked for equality or hashed. If string_internment::intern_string is called with the same string, a value equivalent interned_string object will be returned.

3. If the original string is required, interned_string::get can be invoked to copy the string into a std::string. interned_string also provides iteration via begin/end, however the begin method requires a pointer to original string_internment object. This is not suitable for range iteration, so the method interned_string::bind can be used to create a bound_interned_string that can be used in a range iteration context.

For reference, the reason that interned_string’s does not have a reference back to the string_internment object is to keep their memory footprint lower.

class vtr::string_internment

Storage of interned string, and object capable of generating new interned_string objects.

Public Functions

inline interned_string intern_string(vtr::string_view view)

Intern a string, and return a unique identifier to that string.

If interned_string is ever called with two strings of the same value, the interned_string will be equal.

inline vtr::string_view get_string(StringId id) const

Retrieve a string part based on id.

This method should not generally be called directly.

inline size_t unique_strings() const

Number of unique string parts stored.
class `vtr::interned_string`

Interned string value returned from a `string_internment` object.

This is a value object without allocation. It can be checked for equality and hashed safely against other `interned_string`'s generated from the same `string_internment`.

**Public Functions**

inline `interned_string(std::array<StringId, kMaxParts> intern_ids, size_t n)`

constructor

inline void `get(const string_internment *internment, std::string *output)` const

Copy the underlying string into output.

internment must the object that generated this `interned_string`.

inline `std::string get(const string_internment *internment)` const

Returns the underlying string as a `std::string`.

This method will allocated memory.

inline `bound_interned_string bind(const string_internment *internment)` const

Bind the parent `string_internment` and return a `bound_interned_string` object.

That `bound_interned_string` lifetime must be shorter than this `interned_string` object lifetime, as `bound_interned_string` contains a reference this object, along with a reference to the internment object.

inline `interned_string_iterator begin(const string_internment *internment)` const

`begin()` function

inline `interned_string_iterator end()` const

`end()` function

**Friends**

friend bool `operator==(interned_string lhs, interned_string rhs)` noexcept

== operator

friend bool `operator!=(interned_string lhs, interned_string rhs)` noexcept

!= operator

class `vtr::bound_interned_string`

A `interned_string` bound to it's `string_internment` object.

This object is heavier than just an `interned_string`. This object holds a pointer to `interned_string`, so its lifetime must be shorter than the parent `interned_string`.

17.6. Other
Public Functions

inline bound_interned_string(const string_internment *internment, const interned_string *str) noconstructor

inline interned_string_iterator begin() const
return an iterator to the first part of the interned_string

inline interned_string_iterator end() const
return an iterator to the last part of the interned_string

class vtr::interned_string_iterator
Iterator over interned string.
This object is much heavier memory wise than interned_string, so do not store these.
This iterator only accomidates the forward_iterator concept.
Do no construct this iterator directly. Use either bound_interned_string::begin/end or interned_string::begin/end.

Public Functions

inline interned_string_iterator(const string_internment *internment, std::array<StringId, kMaxParts> intern_ids, size_t n)
constructor for interned string iterator.
Do no construct this iterator directly. Use either bound_interned_string::begin/end or interned_string::begin/end.

inline interned_string_iterator &operator++()
Increment operator for interned_string_iterator.

inline interned_string_iterator operator++(int)
Increment operator for interned_string_iterator.

Friends

friend bool operator==(const interned_string_iterator &lhs, const interned_string_iterator &rhs) == operator

17.6.12 vtr_token

Tokenizer.

Author Jason Luu July 22, 2009
Enums

enum e_token_type
    Token types.
    Values:

    enumerator TOKEN_NULL

    enumerator TOKEN_STRING

    enumerator TOKEN_INT

    enumerator TOKEN_OPEN_SQUARE_BRACKET

    enumerator TOKEN_CLOSE_SQUARE_BRACKET

    enumerator TOKEN_OPEN_SQUIG_BRACKET

    enumerator TOKEN_CLOSE_SQUIG_BRACKET

    enumerator TOKEN_COLON

    enumerator TOKEN_DOT

Functions

t_token *GetTokensFromString(const char *inString, int *num_tokens)
    Returns a token list of the text for a given string.

void freeTokens(t_token *tokens, const int num_tokens)
    Free (tokens)

bool checkTokenType(const t_token token, enum e_token_type token_type)
    Returns true if the token's type equals to token_type.

void my_atof_2D(float **matrix, const int max_i, const int max_j, const char *instring)
    Returns a 2D array representing the atof result of all the input string entries seperated by whitespace.

bool check_my_atof_2D(const int max_i, const int max_j, const char *instring, int *num_entries)
    Checks if the number of entries (separated by whitespace) matches the the expected number (max_i * max_j)
    can be used before calling my_atof_2D
struct t_token
    
    #include <vtr_token.h> Token structure.

Public Members

enum e_token_type type

char *data

17.6.13 vtr_util

namespace vtr

Functions

template<typename Iter>
std::string join(Iter begin, Iter end, std::string delim)
    Joins a sequence by a specified delimeter.
    
    Template join function implementation.
    
    For example the sequence {"home", "user", "my_files", "test.blif"} with delim="/" would return
    "home/user/my_files/test.blif"

template<typename Container>
std::string join(Container container, std::string delim)

template<typename T>
std::string join(std::initializer_list<T> list, std::string delim)

template<typename Container>
void uniquify(Container container)
    Template uniquify function implementation.
    
    Removes repeated elements in the container

namespace vtr
**Functions**

```cpp
std::vector<std::string> split(const char *text, const std::string delims)
Splits the c-style string 'text' along the specified delimiter characters in 'delims'.
The split strings (excluding the delimiters) are returned
```

```cpp
std::vector<std::string> split(const std::string &text, const std::string delims)
Splits the string 'text' along the specified delimiter characters in 'delims'.
The split strings (excluding the delimiters) are returned
```

```cpp
std::string replace_first(const std::string &input, const std::string &search, const std::string &replace)
Returns ‘input’ with the first instance of ‘search’ replaced with ‘replace’.
```

```cpp
std::string replace_all(const std::string &input, const std::string &search, const std::string &replace)
Returns ‘input’ with all instances of ‘search’ replaced with ‘replace’.
```

```cpp
bool starts_with(const std::string &str, const std::string &prefix)
Retruns true if str starts with prefix.
```

```cpp
std::string string_fmt(const char *fmt, ...)
Returns a std::string formatted using a printf-style format string.
```

```cpp
std::string vstring_fmt(const char *fmt, va_list args)
Returns a std::string formatted using a printf-style format string taking an explicit va_list.
```

```cpp
char *strncpy(char *dest, const char *src, size_t size)
An alternate for strncpy since strncpy doesn’t work as most people would expect. This ensures null termination.
```

```cpp
char *strdup(const char *str)
Legacy c-style function replacements.
Typically these add extra error checking and/or correct 'unexpected' behaviour of the standard c-functions
```

```cpp
T atoT(const std::string &value, const std::string &type_name)
Legacy c-style function replacements.
Typically these add extra error checking and/or correct 'unexpected' behaviour of the standard c-functions
```

```cpp
int atoi(const std::string &value)
Legacy c-style function replacements.
Typically these add extra error checking and/or correct 'unexpected' behaviour of the standard c-functions
```

```cpp
double atod(const std::string &value)
Legacy c-style function replacements.
Typically these add extra error checking and/or correct 'unexpected' behaviour of the standard c-functions
```

```cpp
float atof(const std::string &value)
Legacy c-style function replacements.
Typically these add extra error checking and/or correct 'unexpected' behaviour of the standard c-functions
```

```cpp
unsigned atou(const std::string &value)
Legacy c-style function replacements.
Typically these add extra error checking and/or correct 'unexpected' behaviour of the standard c-functions
```
char *\texttt{strtok}(char *\texttt{ptr}, const char *\texttt{tokens}, FILE *\texttt{fp}, char *\texttt{buf})
Get next token, and wrap to next line if \texttt{`} \texttt{at end of line.}

There is a bit of a “gotcha” in \texttt{strtok}. It does not make a * copy of the character array which you pass by pointer on the first call. Thus, you must make sure this array exists for as long as you are using \texttt{strtok} to parse that line. Don’t use local buffers in a bunch of subroutines calling each other; the local buffer may be overwritten when the stack is restored after return from the subroutine.

\texttt{FILE *\texttt{fopen}(const char *\texttt{fname}, const char *\texttt{flag})}
The legacy fopen function with extra error checking.

\texttt{int \texttt{fclose}(FILE *\texttt{f})}
The legacy fclose function.

\texttt{char *\texttt{fgets}(char *\texttt{buf}, int max\_size, FILE *\texttt{fp})}
Get an input line, update the line number and cut off any comment part.

A \texttt{`} \texttt{at the end of a line with no comment part (\#) means continue.} \texttt{vtr::fgets} should give identical results for Windows ( ) and Linux ( ) newlines, since it replaces each carriage return by a newline character.

. Returns NULL after EOF.

\texttt{char *\texttt{getline}(char *&\_lineptr, FILE *\_stream)}
to get an arbitrary long input line and cut off any comment part
the \texttt{getline} function is exaly like the \_\_get\_delim function in GNU with ‘’ delimiter. As a result, to make the function behaviour identical for Windows ( ) and Linux ( ) compiler macros for checking operating systems have been used.

\textbf{Note:} user need to take care of the given pointer, which will be dynamically allocated by getdelim

\texttt{int get\_file\_line\_number\_of\_last\_opened\_file()}
Returns line number of last opened and read file.
File utilities.

\texttt{bool \texttt{file\_exists}(const char *\texttt{filename})}

\texttt{bool \texttt{check\_file\_name\_extension}(const char *\texttt{file\_name}, const char *\texttt{file\_extension})}
Checks the file extension of an file to ensure correct file format.
Returns true if format is correct, and false otherwise.

\textbf{Note:} This is probably a fragile check, but at least should prevent common problems such as swapping architecture file and blif file on the VPR command line.

\texttt{std::vector<\texttt{std::string}> \texttt{ReadLineTokens}(FILE *\texttt{InFile, int *LineNum})}
Legacy ReadLine Tokening.
int `get_pid()`

    Returns pid if os is unix, -1 otherwise.
CHAPTER
EIGHTEEN

INDICES AND TABLES

• genindex
• search
BIBLIOGRAPHY


<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>comment</td>
<td>178</td>
</tr>
<tr>
<td>\</td>
<td>linecontinued</td>
<td></td>
</tr>
<tr>
<td>*(</td>
<td>wildcard</td>
<td></td>
</tr>
<tr>
<td>)</td>
<td>stringescape</td>
<td></td>
</tr>
<tr>
<td>$VTR_ROOT</td>
<td></td>
<td>389</td>
</tr>
<tr>
<td>--absorb_buffer_luts</td>
<td>vpr command line option</td>
<td>142</td>
</tr>
<tr>
<td>--acc_fac</td>
<td>vpr command line option</td>
<td>151</td>
</tr>
<tr>
<td>--activity_file</td>
<td>vpr command line option</td>
<td>164</td>
</tr>
<tr>
<td>--allow_unrelated_clustering</td>
<td>vpr command line option</td>
<td>144</td>
</tr>
<tr>
<td>--alpha_clustering</td>
<td>vpr command line option</td>
<td>144</td>
</tr>
<tr>
<td>--alpha_decay</td>
<td>vpr command line option</td>
<td>144</td>
</tr>
<tr>
<td>--alpha_max</td>
<td>vpr command line option</td>
<td>148</td>
</tr>
<tr>
<td>--alpha_min</td>
<td>vpr command line option</td>
<td>148</td>
</tr>
<tr>
<td>--alpha_t</td>
<td>vpr command line option</td>
<td>148</td>
</tr>
<tr>
<td>--analysis</td>
<td>vpr command line option</td>
<td>138</td>
</tr>
<tr>
<td>--anneal_success_min</td>
<td>vpr command line option</td>
<td>148</td>
</tr>
<tr>
<td>--anneal_success_target</td>
<td>vpr command line option</td>
<td>149</td>
</tr>
<tr>
<td>--astar_fac</td>
<td>vpr command line option</td>
<td>153</td>
</tr>
<tr>
<td>--auto</td>
<td>vpr command line option</td>
<td>139</td>
</tr>
<tr>
<td>--balance_block_type_utilization</td>
<td>vpr command line option</td>
<td>145</td>
</tr>
<tr>
<td>--base_cost_type</td>
<td>vpr command line option</td>
<td>151</td>
</tr>
<tr>
<td>--bb_factor</td>
<td>vpr command line option</td>
<td>151</td>
</tr>
<tr>
<td>--bend_cost</td>
<td>vpr command line option</td>
<td>151</td>
</tr>
<tr>
<td>--beta_clustering</td>
<td>vpr command line option</td>
<td>144</td>
</tr>
<tr>
<td>--circuit_file</td>
<td>vpr command line option</td>
<td>141</td>
</tr>
<tr>
<td>--circuit_format</td>
<td>vpr command line option</td>
<td>141</td>
</tr>
<tr>
<td>--clock_modeling</td>
<td>vpr command line option</td>
<td>141</td>
</tr>
<tr>
<td>--cluster_seed_type</td>
<td>vpr command line option</td>
<td>144</td>
</tr>
<tr>
<td>--clustering_pin_feasibility_filter</td>
<td>vpr command line option</td>
<td>144</td>
</tr>
<tr>
<td>--congested_routing_iteration_threshold</td>
<td>vpr command line option</td>
<td>154</td>
</tr>
<tr>
<td>--connection_driven_clustering</td>
<td>vpr command line option</td>
<td>144</td>
</tr>
<tr>
<td>--const_gen_inference</td>
<td>vpr command line option</td>
<td>143</td>
</tr>
<tr>
<td>--constant_net_method</td>
<td>vpr command line option</td>
<td>140</td>
</tr>
<tr>
<td>--criticality_exp</td>
<td>vpr command line option</td>
<td>153</td>
</tr>
<tr>
<td>--device</td>
<td>vpr command line option</td>
<td>140</td>
</tr>
<tr>
<td>--disp</td>
<td>vpr command line option</td>
<td>139</td>
</tr>
<tr>
<td>--echo_dot_timing_graph_node</td>
<td>vpr command line option</td>
<td>164</td>
</tr>
<tr>
<td>--echo_file</td>
<td>vpr command line option</td>
<td>140</td>
</tr>
<tr>
<td>--enable_timing_computations</td>
<td>vpr command line option</td>
<td>140</td>
</tr>
<tr>
<td>--exit_before_pack</td>
<td>vpr command line option</td>
<td>147</td>
</tr>
<tr>
<td>--exit_t</td>
<td>vpr command line option</td>
<td>147</td>
</tr>
<tr>
<td>--first_iter_pres_fac</td>
<td>vpr command line option</td>
<td>147</td>
</tr>
<tr>
<td>--fix_clusters</td>
<td>vpr command line option</td>
<td>147</td>
</tr>
<tr>
<td>--fix_pins</td>
<td>vpr command line option</td>
<td>148</td>
</tr>
<tr>
<td>--full_stats</td>
<td>vpr command line option</td>
<td>147</td>
</tr>
<tr>
<td>--full_stats</td>
<td>vpr command line option</td>
<td>156</td>
</tr>
<tr>
<td>Command Line Option</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>--gen_post_synthesis_netlist</td>
<td>156</td>
<td></td>
</tr>
<tr>
<td>--generate_rr_node_overuse_report</td>
<td>153</td>
<td></td>
</tr>
<tr>
<td>--graphics_commands</td>
<td>139</td>
<td></td>
</tr>
<tr>
<td>--help</td>
<td>140</td>
<td></td>
</tr>
<tr>
<td>--incremental_reroute_delay_ripup</td>
<td>153</td>
<td></td>
</tr>
<tr>
<td>--init_t</td>
<td>147</td>
<td></td>
</tr>
<tr>
<td>--initial_pres_fac</td>
<td>151</td>
<td></td>
</tr>
<tr>
<td>--inner_loop_recompute_divider</td>
<td>149</td>
<td></td>
</tr>
<tr>
<td>--inner_num</td>
<td>147</td>
<td></td>
</tr>
<tr>
<td>--max_criticality</td>
<td>153</td>
<td></td>
</tr>
<tr>
<td>--max_logged_overused_rr_nodes</td>
<td>152</td>
<td></td>
</tr>
<tr>
<td>--max_router_iterations</td>
<td>150</td>
<td></td>
</tr>
<tr>
<td>--min_incremental_reroute_fanout</td>
<td>152</td>
<td></td>
</tr>
<tr>
<td>--min_route_chan_width_hint</td>
<td>152</td>
<td></td>
</tr>
<tr>
<td>--net_file</td>
<td>142</td>
<td></td>
</tr>
<tr>
<td>--netlist_verbosity</td>
<td>143</td>
<td></td>
</tr>
<tr>
<td>--num_workers</td>
<td>140</td>
<td></td>
</tr>
<tr>
<td>--outfile_prefix</td>
<td>142</td>
<td></td>
</tr>
<tr>
<td>--pack</td>
<td>138</td>
<td></td>
</tr>
<tr>
<td>--pack_feasible_block_array_size</td>
<td>146</td>
<td></td>
</tr>
<tr>
<td>--pack_high_fanout_threshold</td>
<td>146</td>
<td></td>
</tr>
<tr>
<td>--pack_prioritize_transitive_connectivity</td>
<td>146</td>
<td></td>
</tr>
<tr>
<td>--pack_transitive_fanout_threshold</td>
<td>146</td>
<td></td>
</tr>
<tr>
<td>--pack verbosity</td>
<td>146</td>
<td></td>
</tr>
<tr>
<td>--place</td>
<td>138</td>
<td></td>
</tr>
<tr>
<td>--place_algorithm</td>
<td>148</td>
<td></td>
</tr>
<tr>
<td>--place_chan_width</td>
<td>148</td>
<td></td>
</tr>
<tr>
<td>--place_delay_model</td>
<td>149</td>
<td></td>
</tr>
<tr>
<td>--place_delay_model_reducer</td>
<td>150</td>
<td></td>
</tr>
<tr>
<td>--place_delay_offset</td>
<td>150</td>
<td></td>
</tr>
<tr>
<td>--place_delay_ramp_delta_threshold</td>
<td>150</td>
<td></td>
</tr>
<tr>
<td>--place_delay_ramp_slope</td>
<td>150</td>
<td></td>
</tr>
<tr>
<td>--place_file</td>
<td>142</td>
<td></td>
</tr>
<tr>
<td>--place_quench_algorithm</td>
<td>142</td>
<td></td>
</tr>
<tr>
<td>--place_rlim_escape</td>
<td>148</td>
<td></td>
</tr>
<tr>
<td>--place_tsl_abs_margin</td>
<td>148</td>
<td></td>
</tr>
<tr>
<td>--place_tsl_rel_margin</td>
<td>150</td>
<td></td>
</tr>
<tr>
<td>--post_place_timing_report</td>
<td>150</td>
<td></td>
</tr>
<tr>
<td>--post_synth_netlist_unconn_inputs</td>
<td>156</td>
<td></td>
</tr>
<tr>
<td>--post_synth_netlist_unconn_outputs</td>
<td>156</td>
<td></td>
</tr>
<tr>
<td>--power</td>
<td>164</td>
<td></td>
</tr>
<tr>
<td>--pres_fac_mult</td>
<td>151</td>
<td></td>
</tr>
<tr>
<td>--read_rr_graph</td>
<td>142</td>
<td></td>
</tr>
<tr>
<td>--recompute_crit_iter</td>
<td>149</td>
<td></td>
</tr>
<tr>
<td>--route</td>
<td>138</td>
<td></td>
</tr>
<tr>
<td>--route_bb_update</td>
<td>154</td>
<td></td>
</tr>
<tr>
<td>--route_chan_width</td>
<td>151</td>
<td></td>
</tr>
<tr>
<td>--route_file</td>
<td>142</td>
<td></td>
</tr>
<tr>
<td>--route_type</td>
<td>151</td>
<td></td>
</tr>
<tr>
<td>--router_algorithm</td>
<td>151</td>
<td></td>
</tr>
<tr>
<td>--router_debug_net</td>
<td>155</td>
<td></td>
</tr>
<tr>
<td>--router_debug_sink_rr</td>
<td>328</td>
<td></td>
</tr>
<tr>
<td>--router_first_iter_timing_report</td>
<td>155</td>
<td></td>
</tr>
<tr>
<td>--router_high_fanout_threshold</td>
<td>155</td>
<td></td>
</tr>
</tbody>
</table>
vpr command line option, 154
--router_init_wirelength_abort_threshold
  vpr command line option, 153
--router_initial_timing
  vpr command line option, 155
--router_lookahead
  vpr command line option, 155
--router_max_convergence_count
  vpr command line option, 155
--router_reconvergence_cpd_threshold
  vpr command line option, 155
--router_update_lower_bound_delays
  vpr command line option, 155
--routing_budgets_algorithm
  vpr command line option, 154
--routing_failure_predictor
  vpr command line option, 154
--save_graphics
  vpr command line option, 139
--save_routing_per_iteration
  vpr command line option, 154
--sdc_file
  vpr command line option, 142
--seed
  vpr command line option, 147
--sink_rr_node
  command line option, 328
--source_rr_node
  command line option, 328
--strict_checks
  vpr command line option, 141
--sweep_constant_primary_outputs
  vpr command line option, 143
--sweep_dangling_blocks
  vpr command line option, 143
--sweep_dangling_nets
  vpr command line option, 143
--sweep_dangling_primary_ios
  vpr command line option, 143
--target_ext_pin_util
  vpr command line option, 145
--target_utilization
  vpr command line option, 140
--td_place_exp_first
  vpr command line option, 149
--td_place_exp_last
  vpr command line option, 149
--tech_properties
  vpr command line option, 164
--timing_analysis
  vpr command line option, 140
--timing_driven_clustering
  vpr command line option, 144
--timing_report_detail
  vpr command line option, 157
--timing_report_npaths
  vpr command line option, 156
--timing_report_skew
  vpr command line option, 156
--timing_tradeoff
  vpr command line option, 164
--two_stage_clock_routing
  vpr command line option, 149
--verify_binary_search
  vpr command line option, 152
--verify_file_digests
  vpr command line option, 140
--version
  vpr command line option, 140
--write_block_usage
  vpr command line option, 146
--write_rr_graph
  vpr command line option, 142
--write_timing_summary
  vpr command line option, 153
--adder_cin_global
  run_vtr_flow.py command line option, 50
--check_golden
  parse_vtr_task.py command line option, 54
--clock<virtualornetlistclock>
  SDC Option, 176
--cmos_tech
  run_vtr_flow.py command line option, 49
--create_golden
  parse_vtr_task.py command line option, 54
--delete_intermediate_files
  run_vtr_flow.py command line option, 49
--delete_result_files
  run_vtr_flow.py command line option, 49
--early
  SDC Option, 177
--ending_stage
  run_vtr_flow.py command line option, 49
--exclusive
  SDC Option, 172
--from[get_clocks<clocklistorregexes>]
  SDC Option, 173, 174, 176
--from[get_pins<pinlistorregexes>]
  SDC Option, 178
--group{<clocklistorregexes>}
  SDC Option, 172
--h
  vpr command line option, 140
--hold
  SDC Option, 174, 177
--j
  run_vtr_task.py command line option, 51
  vpr command line option, 140
-l
  parse_vtr_task.py command line option, 54
  run_vtr_task.py command line option, 51
-late
  SDC Option, 177
-limit_memory_usage
  run_vtr_flow.py command line option, 49
-max
  SDC Option, 176
-min
  SDC Option, 176
-min_hard_adder_size
  run_vtr_flow.py command line option, 50
-min_hard_mult_size
  run_vtr_flow.py command line option, 50
-name<string>
  SDC Option, 172
-odin_xml
  run_vtr_flow.py command line option, 50
-period<float>
  SDC Option, 171
-power
  run_vtr_flow.py command line option, 49
-s
  run_vtr_task.py command line option, 51
-setup
  SDC Option, 174, 177
-source
  SDC Option, 177
-starting_stage
  run_vtr_flow.py command line option, 49
-system
  run_vtr_task.py command line option, 52
-temp_dir
  run_vtr_flow.py command line option, 50
-timeout
  run_vtr_flow.py command line option, 50
-to[get_clocks<clocklistorregexes>]
  SDC Option, 173–175, 177
-to[get_pins<pinlistorregexes>]
  SDC Option, 175, 178
-track_memory_usage
  run_vtr_flow.py command line option, 49
-valgrind
  run_vtr_flow.py command line option, 49
-waveform{<float><float>}
  SDC Option, 171
<T_clock_to_Qmax="float"min="float"port="string">clock<clocklistorregexes>}
  Tag Attribute, 103
<T_holdvalue="float"port="string"clock="string">clock<geosc_node="int"sink_node="int"switch_id="int"/>
  Tag Attribute, 103
<T_setupvalue="float"port="string"clock="string">clock<equivalent_sites>
  Tag Attribute, 103
<Tdelnum_inputs=int"delay="float"/>
  <areagrid_logic_tile_area="float"/>
  <auto_layoutaspect_ratio="float"/>
  <block_type_id="int"name="unique_identifier"width="int"height="int"/>
  <bufferslogical_effort_factor="float"/>
  <cbtype="pattern">intlist</cb>
  <channelchan_width_max="int"x_min="int"y_min="int"x_max="int"y_max="int"/>
  <clockC_wire="float"C_wire_per_m="float"buffer_size="{float"auto"}”/>
  <clock_networkname="string"num_inst="integer"/>
  <clockname="string"num_pins="int”equivalent="{none|full”}”/>
  <colotype="string”priority="int”startx="expr”repeatx="expr”x_offset="int”y_offset="int”z_offset="int”switch_name="string”/>
  <completename="string”input="string”output="string”/>
  <complexblocklist>content</complexblocklist>
  <connection_blockinput_switch_name="string”/>
  <cornerstype="string”priority="int”/>
  <default_fcin_type="{frac|abs”in_val="{int|float”out_type="{frac|abs”out_val="{int|float”}”>
  <delay_constantmax="float”min="float”in_port="string”out_port="string”/>
  <delay_matrixtype="{max|min”in_port="string”out_port="string”matrix”>
  <delay>
  <directfrom="string”to="string”>
  <directname="string”from_pin="string”to_pin="string”x_offset="int”y_offset="int”z_offset="int”switch_name="string”>
  <fcoverridefc_type="{frac|abs”fc_val="{int|float”port="string”over_per_instance="float”C_internal="float”>
  Tag Attribute, 105
  Tag Attribute, 103
  Tag Attribute, 103
  Tag Attribute, 199
  Tag Attribute, 85
  Tag Attribute, 65
  Tag Attribute, 79
  Tag Attribute, 173
  Tag Attribute, 162
  Tag Attribute, 114
  Tag Attribute, 197
  Tag Attribute, 114
  Tag Attribute, 107
  Tag Attribute, 79
  Tag Attribute, 195
  Tag Attribute, 108
  Tag Attribute, 111
  Tag Attribute, 85, 96
  Tag Attribute, 67
  Tag Attribute, 97
  Tag Attribute, 62
  Tag Attribute, 77
  Tag Attribute, 65
  Tag Attribute, 79
  Tag Attribute, 102
  Tag Attribute, 102
  Tag Attribute, 173, 176
  Tag Attribute, 61
  Tag Attribute, 86
  Tag Attribute, 114
  Tag Attribute, 97
  Tag Attribute, 105
  Tag Attribute, 103
  Tag Attribute, 103
  Tag Attribute, 85
  Tag Attribute, 79
Verilog-to-Routing Documentation, Release 8.1.0-dev

ClusteringContext::clb_nlist (C++ member), 394
command line option
  --router_debug_sink_rr, 328
  --sink_rr_node, 328
  --source_rr_node, 328
Context (C++ struct), 394
create_clock
  SDC Command, 171

D
delta_override}
  vpr command line option, 149
DeviceContext (C++ struct), 394
DeviceContext::chan_width (C++ member), 395
DeviceContext::grid (C++ member), 395
DeviceContext::has_multiple_equivalent_tiles
  (C++ member), 395
DeviceContext::read_rr_graph_filename (C++ member), 395
DeviceContext::rr_node_to_non_config_node_set
  (C++ member), 395
DeviceContext::rr_non_config_node_sets (C++ member), 395
DeviceContext::rr_rc_data (C++ member), 395
DeviceContext::switch_fanin_remap (C++ member), 395
DeviceContext::virtual_clock_network_root_idx
  (C++ member), 395
dynamic}
  vpr command line option, 154

E
e_token_type (C++ enum), 489
e_token_type::TOKEN_CLOSE_SQUARE_BRACKET
  (C++ enumerator), 489
e_token_type::TOKEN_CLOSE_SQUIG_BRACKET
  (C++ enumerator), 489
e_token_type::TOKEN_COLON (C++ enumerator), 489
e_token_type::TOKEN_DOT (C++ enumerator), 489
e_token_type::TOKEN_INT (C++ enumerator), 489
e_token_type::TOKEN_NULL (C++ enumerator), 489
e_token_type::TOKEN_OPEN_SQUARE_BRACKET
  (C++ enumerator), 489
e_token_type::TOKEN_OPEN_SQUIG_BRACKET (C++
  enumerator), 489
e_token_type::TOKEN_STRING (C++ enumerator), 489

F
freeTokens (C++ function), 489

G
geomean}
  vpr command line option, 150
get_bp_state Globals (C++ function), 476
getTokensFromString (C++ function), 489

M
map}
  vpr command line option, 155
  max
    vpr command line option, 150
  median
    vpr command line option, 150
  MWTA, 389
my_atof_2D (C++ function), 489

N
Netlist (C++ class), 404
Netlist::block_attrs (C++ function), 405
Netlist::block_clock_pins (C++ function), 405
Netlist::block_clock_ports (C++ function), 405
Netlist::block_input_pins (C++ function), 405
Netlist::block_input_ports (C++ function), 405
Netlist::block_is_combinational (C++ function),
  405
Netlist::block_name (C++ function), 405
Netlist::block_output_pins (C++ function), 405
Netlist::block_output_ports (C++ function), 405
Netlist::block_params (C++ function), 405
Netlist::block_pins (C++ function), 405
Netlist::block_ports (C++ function), 405
Netlist::blocks (C++ function), 405
Netlist::compress (C++ function), 410
Netlist::find_block (C++ function), 408
Netlist::find_net (C++ function), 408
Netlist::find_pin (C++ function), 408
Netlist::find_port (C++ function), 408
Netlist::is_compressed (C++ function), 405
Netlist::is_dirty (C++ function), 405
Netlist::merge_nets (C++ function), 409
Netlist::net_driver (C++ function), 407
Netlist::net_driver_block (C++ function), 407
Netlist::net_is_constant (C++ function), 407
Netlist::net_name (C++ function), 407
Netlist::net_pin (C++ function), 407
Netlist::net_pin_block (C++ function), 407
Netlist::net_pins (C++ function), 407
Netlist::net_sinks (C++ function), 407
Netlist::netlist_id (C++ function), 405
Netlist::netlist_name (C++ function), 405
Netlist::nets (C++ function), 408
Netlist::pin_block (C++ function), 407
Netlist::pin_is_constant (C++ function), 407
Netlist::pin_name (C++ function), 406
Netlist::pin_net (C++ function), 406
Netlist::pin_net_index (C++ function), 406
Netlist::pin_port (C++ function), 407

Index

507
Netlist::remove_port
Netlist::remove_pin
Netlist::remove_net_pin
Netlist::remove_net
Netlist::remove_block
Netlist::print_stats
Netlist::set_block_attr
Netlist::remove_and_compress
Netlist::ports
Netlist::port_width
Netlist::port_type
Netlist::port_pins
Netlist::port_pin
Netlist::port_net
Netlist::port_name
Netlist::port_block
Netlist::pins
Netlist::set_block_param
Netlist::set_block_name
Netlist::pin_type
Netlist::pin_port_type
Netlist::pin_port_bit
Netlist::set_pin_net
Netlist::set_pin_is_constant
Netlist::verify

off

off

off

parse_vtr_task.py command line option

vpr command line option

vpr command line option

RoutingContext (C++ struct), 396
RoutingContext::cached_router_lookahead_ (C++ member), 397
RoutingContext::net_status (C++ member), 396
RoutingContext::non_configurable_bitset (C++ member), 396
RoutingContext::route_bb (C++ member), 397
RoutingContext::routing_id (C++ member), 397
RRGraphBuilder (C++ class), 420
RRGraphBuilder::add_node_side (C++ function), 423
RRGraphBuilder::add_node_to_all_locs (C++ function), 421
RRGraphBuilder::add_rr_segment (C++ function), 421
RRGraphBuilder::add_rr_switch (C++ function), 421
RRGraphBuilder::alloc_and_load_edges (C++ function), 423
RRGraphBuilder::clear (C++ function), 421
RRGraphBuilder::count_rr_switches (C++ function), 423
RRGraphBuilder::emplace_back (C++ function), 422
RRGraphBuilder::emplace_back_edge (C++ function), 422
RRGraphBuilder::end_rr_edge_metadata (C++ function), 420
RRGraphBuilder::end_rr_node_metadata (C++ function), 420
RRGraphBuilder::find_rr_edge_metadata (C++ function), 420
RRGraphBuilder::find_rr_node_metadata (C++ function), 420
RRGraphBuilder::init_fan_in (C++ function), 423
RRGraphBuilder::mark_edges_as_rr_switch_ids (C++ function), 423
RRGraphBuilder::node_lookup (C++ function), 420
RRGraphBuilder::partition_edges (C++ function), 423
RRGraphBuilder::remap_rr_node_switch_indices (C++ function), 423
RRGraphBuilder::reorder_nodes (C++ function), 421
RRGraphBuilder::reserve_edges (C++ function), 422
RRGraphBuilder::reserve_nodes (C++ function), 423
RRGraphBuilder::resize_nodes (C++ function), 423
RRGraphBuilder::resize_switches (C++ function), 423
RRGraphBuilder::rr_edge_metadata (C++ function), 420
RRSpatialLookup::reserve_nodes (C++ function), 425
RRSpatialLookup::resize_nodes (C++ function), 426
run_vtr_flow.py command line option
- adder_cin_global, 50
- cmos_tech, 49
- delete_intermediate_files, 49
- delete_result_files, 49
- ending_stage, 49
- limit_memory_usage, 49
- min_hard_adder_size, 50
- min_hard_mult_size, 50
- odin_xml, 50
- power, 49
- starting_stage, 49
- temp_dir, 50
- timeout, 50
- track_memory_usage, 49
- valgrind, 50
run_vtr_task.py command line option
- j, 51
- l, 51
- s, 51
- system, 52
S
SDC Command
#(comment), \/(linecontinued), *( wildcard), 178
create_clock, 171
set_clock_groups, 172
set_clock_latency, 177
set_clock_uncertainty, 176
set_disable_timing, 178
set_false_path, 173
set_input_delay/set_output_delay, 176
set_max_delay/set_min_delay, 173
set_multicycle_path, 174
SDC Option
- clock<virtualornetlistclock>, 176
- early, 177
- exclusive, 172
- from[get_clocks<clocklistorregexes>], 173, 174, 176
- from[get_pins<pinlistorregexes>], 178
- group{<clocklistorregexes>}, 172
- hold, 174, 177
- late, 177
- max, 176
- min, 176
- name<string>, 172
- period<float>, 171
- setup, 174, 177
- source, 177
- to[get_clocks<clocklistorregexes>], 173–175, 177
- to[get_pins<pinlistorregexes>], 175, 178
- waveform{<float><float>}, 171
- delay, 173, 176
- latency, 177
- netlistclocklistorregexes, 172
- path_multiplier, 175
- uncertainty, 177
- [get_clocks<clocklistorregexes>], 177
- [get_ports{<I/Olistorregexes>}], 176
set_clock_groups
SDC Command, 172
set_clock_latency
SDC Command, 177
set_clock_uncertainty
SDC Command, 176
set_disable_timing
SDC Command, 178
set_false_path
SDC Command, 173
set_input_delay/set_output_delay
SDC Command, 176
set_max_delay/set_min_delay
SDC Command, 173
set_multicycle_path
SDC Command, 174
set_noisy_warn_log_file (C++ function), 469
std (C++ type), 431
T
t_token (C++ struct), 489
t_token::data (C++ member), 490
t_token::type (C++ member), 490
Tag Attribute
<T_clock_to_Qmax="float" min="float" port="string" clock="string"/>
103
<T_holdvalue="float" port="string" clock="string"/>
103
<T_setupvalue="float" port="string" clock="string"/>
103
<Tdelnum_inputs="int" delay="float"/>
81
<areagrid_logic_tile_area="float"/>
79
<auto_layoutaspect_ratio="float"/>
62
<block_typeid="int" name="unique_identifier" width="int"/>
197
<bufferslogical_effort_factor="float"/>
114
<cbtype="pattern">intlist</cb>, 107
<chan_width_distr>content</chan_width_distr>, 79
<x_channelchan_width_max="int" x_min="int" y_min="int" x_max="int" y_max="int"/>
195
--pack, 138
--pack_feasible_block_array_size, 146
--pack_high_fanout_threshold, 146
--pack_prioritize_transitive_connectivity, 146
--pack_transitive_fanout_threshold, 146
--pack verbosity, 146
--place, 138
--place_algorithm, 148
--place chan width, 148
--place delay model, 149
--place delay model reducer, 150
--place delay offset, 150
--place delay ramp delta threshold, 150
--place delay ramp slope, 150
--place file, 142
--place quench algorithm, 148
--place rlim escape, 148
--place tsu abs margin, 150
--place tsu rel margin, 150
--post place timing report, 150
--post synth netlist unconn inputs, 156
--post synth netlist unconn outputs, 156
--power, 164
--pres fac mult, 151
--read rr graph, 142
--recompute crit iter, 149
--route, 138
--route bb update, 154
--route chan width, 151
--route file, 142
--route type, 151
--router algorithm, 152
--router debug net, 155
--router debug sink rr, 155
--router first iter timing report, 155
--router high fanout threshold, 154
--router init wirelength abort threshold, 153
--router initial timing, 155
--router lookahead, 155
--router max convergence count, 155
--router reconvergence cpd threshold, 155
--router update lower bound delays, 155
--routing budgets algorithm, 154
--routing failure predictor, 154
--save graphics, 139
--save routing per iteration, 154
--sdc file, 142
--seed, 147
--strict checks, 141
--sweep constant primary outputs, 143
--sweep dangling blocks, 143
--sweep dangling nets, 143
--sweep dangling primary ios, 143
--target ext pin util, 145
--target utilization, 140
--td place exp first, 149
--td place exp last, 149
--tech properties, 164
--timing analysis, 140
--timing driven clustering, 144
--timing report detail, 157
--timing report npaths, 156
--timing report skew, 164
--timing tradeoff, 149
--two stage clock routing, 141
--verify binary search, 152
--verify file digests, 140
--version, 140
--write block usage, 146
--write rr graph, 142
--write timing summary, 153
--h, 140
--j, 140
--architecture, 138
--arithmetic, 150
--auto}, 145
--circuit, 138
--delta override}, 149
--dynamic}, 154
--geometric}, 150
--map}, 155
--max, 150
--median, 150
--off, 145
--off), 141

VprContext (C++ class), 393
vtr::aligned_allocator (C++ struct), 466
vtr::arithmean (C++ function), 482
vtr::array_view (C++ class), 462
vtr::array_view::array_view (C++ function), 462
vtr::array_view::at (C++ function), 462
vtr::array_view::back (C++ function), 462
vtr::array_view::begin (C++ function), 463
vtr::array_view::begin (C++ function), 463
vtr::array_view::end (C++ function), 463
vtr::array_view::empty (C++ function), 463
vtr::array_view::front (C++ function), 463
vtr::array_view::length (C++ function), 463
vtr::array_view::operator[] (C++ function), 462
vtr::array_view::size (C++ function), 462
vtr::array_view_id (C++ class), 461
vtr::array_view_id::at (C++ function), 461
Index
vtr::geomean (C++ function), 482
vtr::get_file_line_number_of_last_opened_file (C++ function), 492
vtr::get_max_rss (C++ function), 485
vtr::get_uid (C++ function), 492
vtr::get_random_state (C++ function), 485
vtr::getcwd (C++ function), 484
vtr::getline (C++ function), 492
vtr::hash_combine (C++ function), 465
vtr::hash_pair (C++ struct), 465
vtr::hash_pair::operator() (C++ function), 466
vtr::InfernoColorMap (C++ class), 480
vtr::InfernoColorMap::InfernoColorMap (C++ function), 480
vtr::insert_in_vptr_list (C++ function), 450
vtr::interned_string (C++ class), 486
vtr::interned_string::begin (C++ function), 487
vtr::interned_string::bind (C++ function), 487
vtr::interned_string::end (C++ function), 487
vtr::interned_string::get (C++ function), 487
vtr::interned_string::interned_string (C++ function), 487
vtr::interned_string::operator!= (C++ function), 487
vtr::interned_string::operator== (C++ function), 487
vtr::interned_string_iterator (C++ class), 488
vtr::interned_string_iterator::interned_string (C++ function), 488
vtr::interned_string_iterator::operator++ (C++ function), 488
vtr::interned_string_iterator::operator== (C++ function), 488
vtr::ipow (C++ function), 483
vtr::irand (C++ function), 485
vtr::isclose (C++ function), 482, 483
vtr::join (C++ function), 490
vtr::lcm (C++ function), 482
vtr::Line (C++ class), 474
vtr::Line::bounding_box (C++ function), 474
vtr::Line::Line (C++ function), 474
vtr::Line::points (C++ function), 474
vtr::linear_bimap (C++ type), 449
vtr::linear_interpolate_or_extrapolate (C++ function), 483
vtr::linear_map (C++ class), 441
vtr::linear_map::at (C++ function), 442
vtr::linear_map::begin (C++ function), 442
vtr::linear_map::cbegin (C++ function), 442
vtr::linear_map::end (C++ function), 442
vtr::linear_map::clear (C++ function), 443
vtr::linear_map::count (C++ function), 443
vtr::linear_map::crbegin (C++ function), 443
vtr::linear_map::crend (C++ function), 442
vtr::linear_map::emplace (C++ function), 443
vtr::linear_map::empty (C++ function), 442
vtr::linear_map::end (C++ function), 442
vtr::linear_map::equal_range (C++ function), 443
vtr::linear_map::erase (C++ function), 443
vtr::linear_map::find (C++ function), 443
vtr::linear_map::insert (C++ function), 442
vtr::linear_map::linear_map (C++ function), 441, 442
vtr::linear_map::lower_bound (C++ function), 443
vtr::linear_map::max_size (C++ function), 442
vtr::linear_map::operator= (C++ function), 441
vtr::linear_map::operator[] (C++ function), 442
vtr::linear_map::rbegin (C++ function), 442
vtr::linear_map::rend (C++ function), 442
vtr::linear_map::reserve (C++ function), 443
vtr::linear_map::shrink_to_fit (C++ function), 443
vtr::linear_map::size (C++ function), 442
vtr::linear_map::swap (C++ function), 443
vtr::linear_map::upper_bound (C++ function), 442
vtr::linear_map::valid_size (C++ function), 444
vtr::LogicValue (C++ enum), 481
vtr::LogicValue::DONT_CARE (C++ enumerator), 481
vtr::LogicValue::FALSE (C++ enumerator), 481
vtr::LogicValue::TRUE (C++ enumerator), 481
vtr::LogicValue::UNKNOWN (C++ enumerator), 481
vtr::make_flat_map (C++ function), 447
vtr::make_flat_map2 (C++ function), 447
vtr::make_key_range (C++ function), 468
vtr::make_range (C++ function), 427
vtr::make_value_range (C++ function), 468
vtr::map_key_iter (C++ type), 468
vtr::map_value_iter (C++ type), 468
vtr::Matrix (C++ type), 456
vtr::median (C++ function), 482, 483
vtr::memalign (C++ function), 466
vtr::NdMatrix (C++ class), 452
vtr::NdMatrix::operator[] (C++ function), 453
vtr::NdMatrix<T, 1>::operator[] (C++ function), 445
vtr::NdMatrixBase (C++ class), 454
vtr::NdMatrixBase::begin_index (C++ function), 454
vtr::NdMatrixBase::clear (C++ function), 455
vtr::NdMatrixBase::dim_size (C++ function), 454
vtr::NdMatrixBase::empty (C++ function), 454
vtr::NdMatrixBase::end_index (C++ function), 454
vtr::NdMatrixBase::fill (C++ function), 454
vtr::NdMatrixBase::get (C++ function), 454
vtr::NdMatrixBase::ndims (C++ function), 454
vtr::NdMatrixBase::NdMatrixBase (C++ function), 454, 455
vtr::NdMatrixBase::operator=(C++ function), 455
vtr::NdMatrixBase::resize (C++ function), 455
vtr::NdMatrixBase::size (C++ function), 454
vtr::NdMatrixProxy (C++ class), 455
vtr::NdMatrixProxy::NdMatrixProxy (C++ function), 455
vtr::NdMatrixProxy::operator= (C++ function), 455
vtr::NdMatrixProxy::operator[] (C++ function), 455
vtr::NdMatrixProxy<T, 1> (C++ class), 455
vtr::NdMatrixProxy<T, 1>::data (C++ function), 456
vtr::NdMatrixProxy<T, 1>::operator= (C++ function), 456
vtr::NdMatrixProxy<T, 1>::operator[] (C++ function), 456
vtr::NdMatrixProxy<T, 1>::operator (C++ function), 456
vtr::NdMatrixProxy<T, 1>::operator= (C++ function), 456
vtr::NdMatrixProxy<T, 1>::operator[] (C++ function), 456
vtr::OffsetMatrixProxy (C++ class), 457
vtr::OffsetMatrixProxy (C++ function), 457
vtr::OffsetMatrixProxy<T, 1> (C++ class), 457
vtr::OffsetMatrixProxy<T, 1>::operator= (C++ function), 458
vtr::OffsetMatrixProxy<T, 1>::operator[] (C++ function), 458
vtr::OffsetMatrixProxy<T, 1>::operator (C++ function), 458
vtr::OffsetMatrixProxy<T, 1>::operator= (C++ function), 458
vtr::OffsetMatrixProxy<T, 1>::operator[] (C++ function), 458
vtr::OffsetMatrixBase (C++ class), 458
vtr::OffsetMatrixBase::begin_index (C++ function), 459
vtr::OffsetMatrixBase::clear (C++ function), 459
vtr::OffsetMatrixBase::dim_size (C++ function), 459
vtr::OffsetMatrixBase::empty (C++ function), 459
vtr::OffsetMatrixBase::end_index (C++ function), 459
vtr::OffsetMatrixBase::fill (C++ function), 459
vtr::OffsetMatrixBase::ndims (C++ function), 459
vtr::OffsetMatrixBase::NdOffsetMatrixBase (C++ function), 459
vtr::OffsetMatrixBase::operator= (C++ function), 459
vtr::OffsetMatrixBase::operator (C++ function), 459
vtr::OffsetMatrixBase::resize (C++ function), 459
vtr::OffsetMatrixBase::size (C++ function), 459
vtr::OffsetMatrixProxy (C++ class), 459
vtr::OffsetMatrixProxy::NdOffsetMatrixProxy (C++ function), 460
vtr::OffsetMatrixProxy::operator[] (C++ function), 460
vtr::OffsetMatrixProxy<T, 1>::operator[] (C++ function), 460
vtr::OffsetMatrixProxy<T, 1>::operator= (C++ function), 460
vtr::OffsetMatrixProxy<T, 1>::operator[] (C++ function), 460
vtr::OffsetMatrixProxy<T, 1>::operator (C++ function), 460
vtr::OffsetMatrixProxy<T, 1>::operator= (C++ function), 460
vtr::OffsetMatrixProxy<T, 1>::operator[] (C++ function), 460
vtr::OffsetMatrixProxy<T, 1>::operator (C++ function), 460
vtr::OsFormatGuard (C++ class), 483
vtr::OsFormatGuard::OsFormatGuard (C++ function), 484
vtr::OsFormatGuard::operator= (C++ function), 484
vtr::OsFormatGuard::OsFormatGuard (C++ function), 484
vtr::pair_first_iter (C++ class), 467
vtr::pair_first_iter::operator* (C++ function), 467
vtr::pair_first_iter::operator++ (C++ function), 467
vtr::pair_first_iter::operator-- (C++ function), 467
vtr::pair_first_iter::operator-> (C++ function), 467
vtr::pair_first_iter::pair_first_iter (C++ function), 467
vtr::pair_second_iter (C++ class), 467
vtr::pair_second_iter::operator* (C++ function), 467
vtr::pair_second_iter::operator++ (C++ function), 467
vtr::pair_second_iter::operator-- (C++ function), 467
vtr::pair_second_iter::operator-> (C++ function), 467
vtr::pair_second_iter::pair_second_iter (C++ function), 467
vtr::PlasmaColorMap (C++ class), 480
vtr::PlasmaColorMap::PlasmaColorMap (C++ function), 481
vtr::Point (C++ class), 472
vtr::Point::operator! = (C++ function), 473
vtr::Point::operator== (C++ function), 473
vtr::Point::operator< (C++ function), 473
vtr::Point: :set (C++ function), 472
vtr::Point: :set_x (C++ function), 472
vtr::Point: :set_y (C++ function), 472
vtr::Point::swap (C++ function), 472
vtr::Point::x (C++ function), 472
vtr::Point::y (C++ function), 472
vtr::Range (C++ class), 427
vtr::Range::begin (C++ function), 428
vtr::vector_map::shrink_to_fit (C++ function), 440
vtr::vector_map::size (C++ function), 440
vtr::vector_map::update (C++ function), 441
vtr::vector_map::vector_map (C++ function), 440
vtr::ViridisColorMap (C++ class), 481
vtr::ViridisColorMap::ViridisColorMap (C++ function), 481
vtr::vstring_fmt (C++ function), 491
vtr::VtrError (C++ class), 470
vtr::VtrError::filename (C++ function), 470
vtr::VtrError::filename_c_str (C++ function), 470
vtr::VtrError::line (C++ function), 470
vtr::VtrError::VtrError (C++ function), 470