Verilog-to-Routing Documentation

Release 8.0.0-dev

VTR Developers

Jun 28, 2018

Usage

Form more information on the Verilog-to-Routing (VTR) project see *VTR* and *VTR CAD Flow*.

For documentation and tutorials on the FPGA architecture description langauge see: *FPGA Architecture Description*.

For more specific documentation about VPR see *VPR*.

CHAPTER 1

VTR

The Verilog-to-Routing (VTR) project *[RLY+12][LAK+14]* is a world-wide collaborative effort to provide a 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 perfoms:

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

Generating FPGA speed and area results.

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

1.1 Get VTR

The official VTR release is available from:

http://www.eecg.utoronto.ca/vtr/terms.html

1.1.1 How to Cite

The following paper may be used as a general citation for VTR:

10. Luu, J. Goeders, M. Wainberg, A. Somerville, T. Yu, K. Nasartschuk, M. Nasr, S. Wang, T. Liu, N. Ahmed, K. B. Kent, J. Anderson, J. Rose and V. Betz "VTR 7.0: Next Generation Architecture and CAD System for FPGAs," ACM TRETS, Vol. 7, No. 2, June 2014, pp. 6:1 - 6:30.

1.1.2 Release

The VTR 8.0 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 8.0 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.

1.1.3 Development Trunk

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

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

Unlike the nicely packaged offical releases the code 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.

1.2 Install VTR

1. *Download* the VTR release
2. Unpack the release in a directory of your choice (herafter referred to as *$VTR_ROOT*)
3. Navigate to *$VTR_ROOT* and run

   ```
   make
   ```

   which will build all the required tools.

Warning: *$VTR_ROOT* should be replaced with the path to the root of VTR source tree on your machine.

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.

See also:

More information about building VTR can be found in the *Developer Guide*

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.

1.2.1 Verifying Installation

To verfiy that VTR has been installed correctly run:

```
$VTR_ROOT/vtr_flow/scripts/run_vtr_task.pl basic_flow
```

The expected output is:

```
k6_N10_memSize16384_memData64_40nm_timing/ch_intrinsics...OK
```

1.3 VTR CAD Flow

Fig. 1.1 illustrates the CAD flow typically used in VTR.

First, *Odin II* converts a Verilog Hardware Destription Language (HDL) design into a flattened netlist consisting of logic gates and blackboxes which represent heterogeneous blocks *[JKGS10]*.

Next, the *ABC* synthesis package is used to perform technology-independent logic optimization, and then technology-maps the circuit into LUTs and flip-flops *[SG][PHMB07][CCMB07]*. The output of ABC is a *.blif format* netlist of LUTs, flip flops, and blackboxes.

VPR then packs this netlist into more coarse-grained logic blocks, places and then routes the circuit *[BRM99][Bet98][BR96a][BR96b][BR97b][BR97a][MBR99][MBR00][BR00]*. Generating *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.

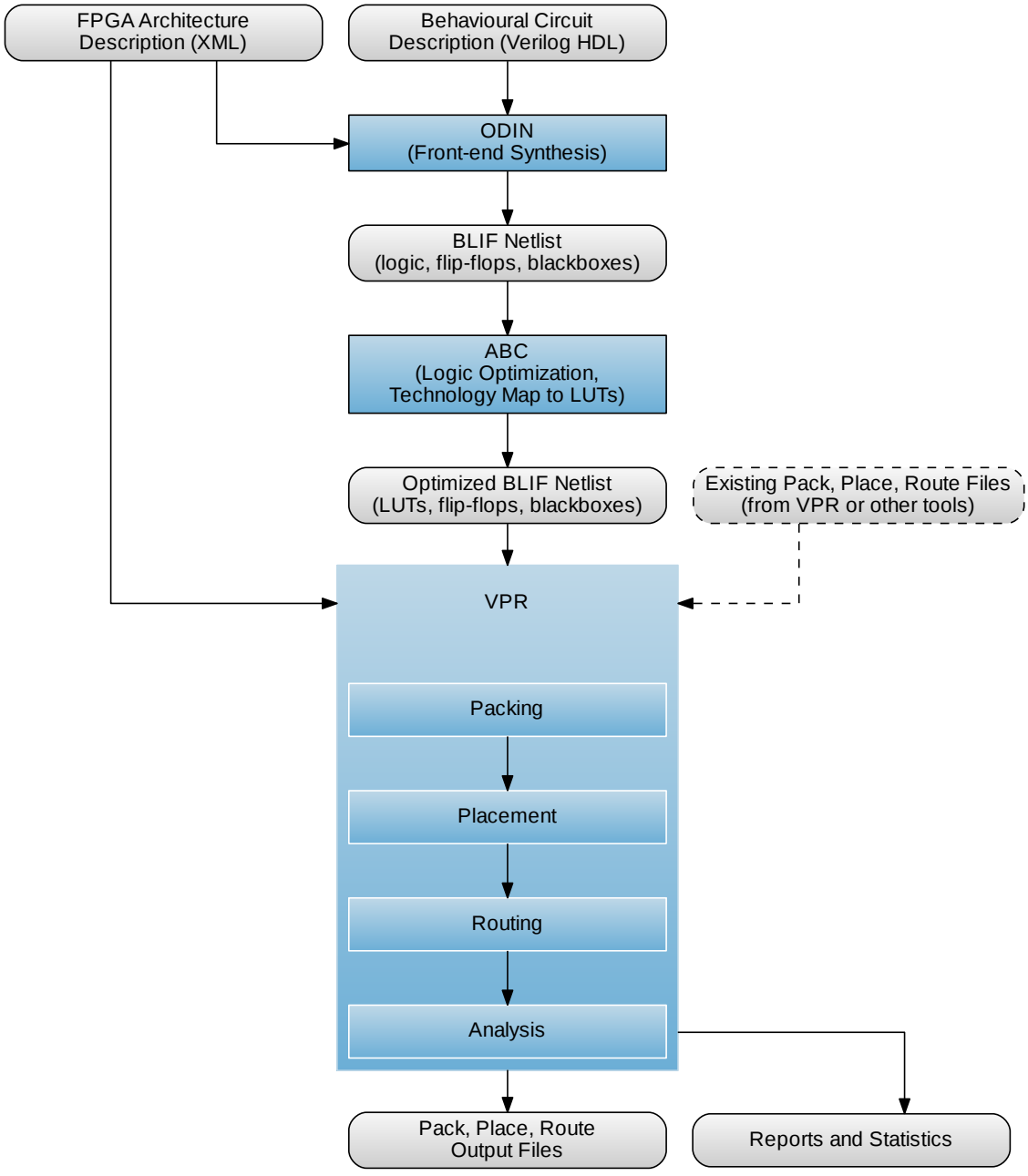


Fig. 1.1: Typical VTR CAD Flow

1.3.1 CAD Flow Variations

Many variations on this CAD flow are possible. It is possible to use other high-level synthesis tools to generate the blif files that are passed into ABC. Also, one can use different logic optimizers and technology mappers than ABC; just put the output netlist from your technology-mapper into .blif format and feed it into VPR.

Alternatively, 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.

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 want only to analyze an implementation produced by another tool with VPR, 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 create an *rr_graph.xml file*, which can be loaded directly into VPR.

1.4 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.

1.4.1 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.pl <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.pl`.

1.4.2 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:

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

Tasks can be executed using *run_vtr_task*:

```
$VTR_ROOT/vtr_flow/scripts/run_vtr_task.pl <task_name>
```

See also:

run_vtr_task for the detailed command line options of `run_vtr_task.pl`.

See also:

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

1.4.3 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.

1.5 Benchmarks

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

1.5.1 VTR Benchmarks

The VTR benchmarks *[RLY+12][LAK+14]* 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 1.1: The VTR 7.0 Benchmarks.

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

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
```

1.5.2 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.

1.5.3 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:

```
$VTR_ROOT/vtr_flow/benchmarks/blif/<#>
```

where K = <#>.

Table 1.2: The MCNC20 benchmarks.

Benchmark	Approximate Number of Netlist Primitives
alu4	934
apex2	1116
apex4	916
bigkey	1561
clma	3754
des	1199
diffeq	1410
dsip	1559
elliptic	3535
ex1010	2669
ex5p	824
frisc	3291
misex3	842
pdc	2879
s298	732
s38417	4888
s38584.1	4726
seq	1041
spla	2278
tseng	1583

1.6 Power Estimation

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

Fig. 1.2 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.

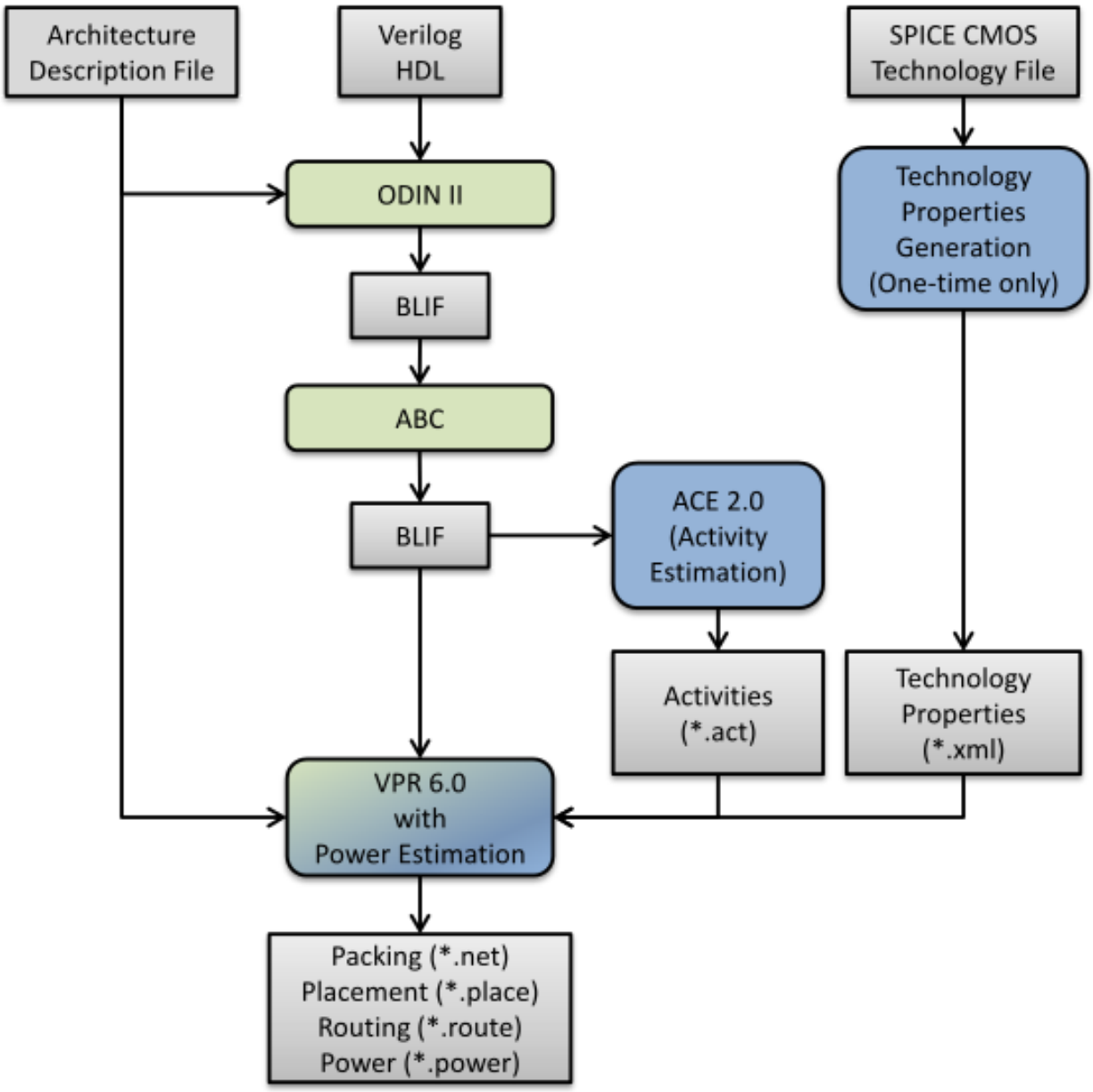


Fig. 1.2: Power Estimation in the VTR Flow

1.6.1 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.pl -power*
- *run_vtr_flow.pl -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.

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*.

1.6.2 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 Celcius.

ACE 2.0 Activity Estimation

Power estimation requires activity information for the entire netlist. This ativity 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 $P_1(clk) = 0.5$.

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 $A_S(clk) = 2$.

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 modifed 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> -o <activities.act> -n <new.blif>
```

where

- `<abc.blif>`: Is the input BLIF file produced by ABC.
- `<activities.act>`: Is the activity file to be created.
- `<new.blif>`: The new BLIF file.

  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.

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>
```

1.6.3 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 modelleling.

   The user supplies all buffer sizes and wire-lengths. Any not provided by the user are ignored.

2. `auto-size`: Detailed transistor level modelleling.

   The user can supply buffer sizes and wire-lengths; however, they will be automatically inserted when not provided.

3. `pin-toggle`: Higher-level modelling.

   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. 1.3).

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

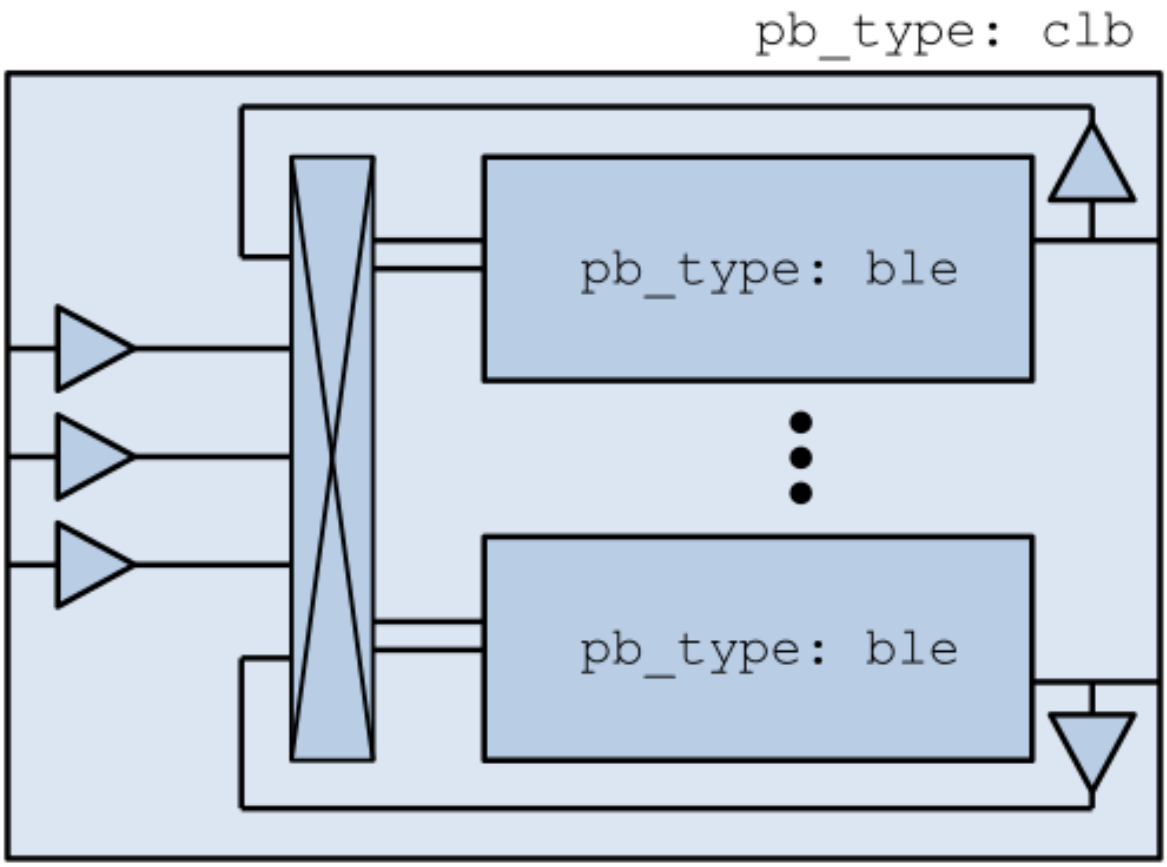


Fig. 1.3: Sample Block

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:

```
<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 seperate 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.
- `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. 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_calc_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:

```
<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_porb="en1"/>
        <port name="D" energy_per_toggle="1.0e-12" scaled_by_static_porb_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 $(1 - signal\_probability)$.

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:

$$P_{dyn} = \frac{1}{2}\alpha C V^2.$$

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

```
<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 `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:

```
<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 `pb_types`. No further recursive power estimation will be performed.

1.6.4 Global Routing

Global routing consists of switch boxes and input connection boxes.

Switch Boxes

Switch boxes are modelled as the following components (Fig. 1.4):

1. Multiplexer
2. Buffer
3. Wire capacitance

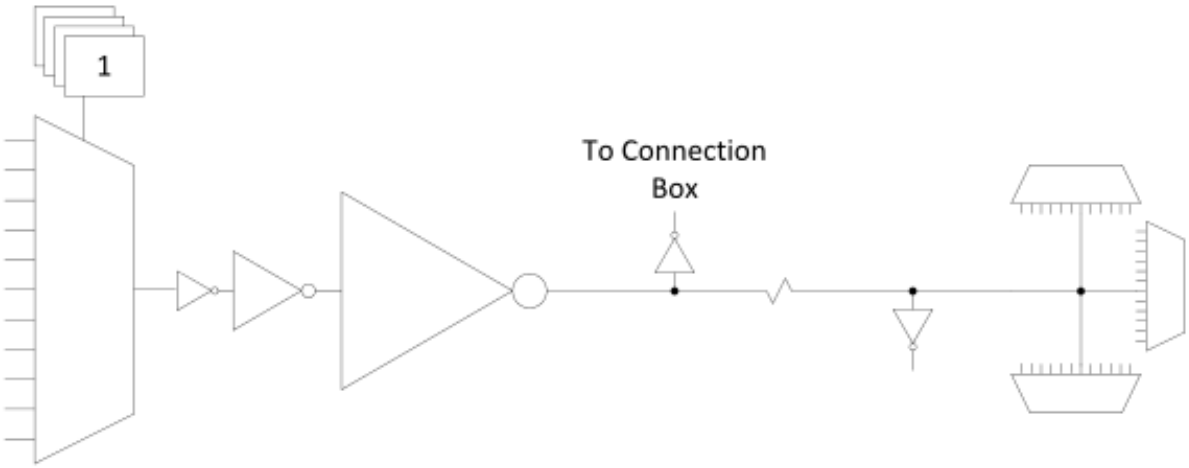


Fig. 1.4: Switch Box

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.

Buffer: The buffer is a multistage CMOS buffer. The buffer size is determined based upon output capacitance provided in the architecture file:

```
<switchlist>
    <switch type="mux" ... C_out="1.0e-16"/>
</switchlist>
```

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. 1.5):

- 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.

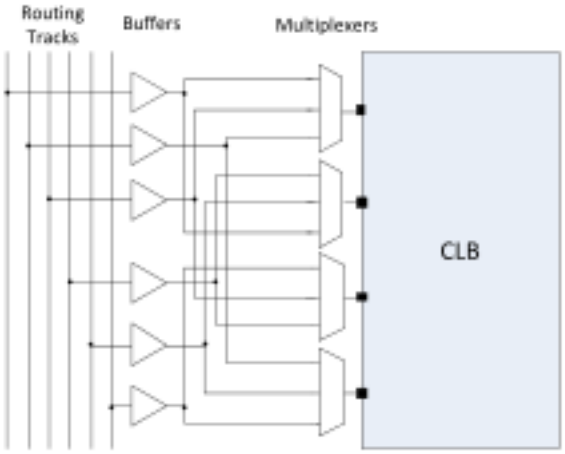


Fig. 1.5: Connection Box

Clock Network

The clock network modelled is a four quadrant spine and rib design, as illustrated in Fig. 1.6. 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.

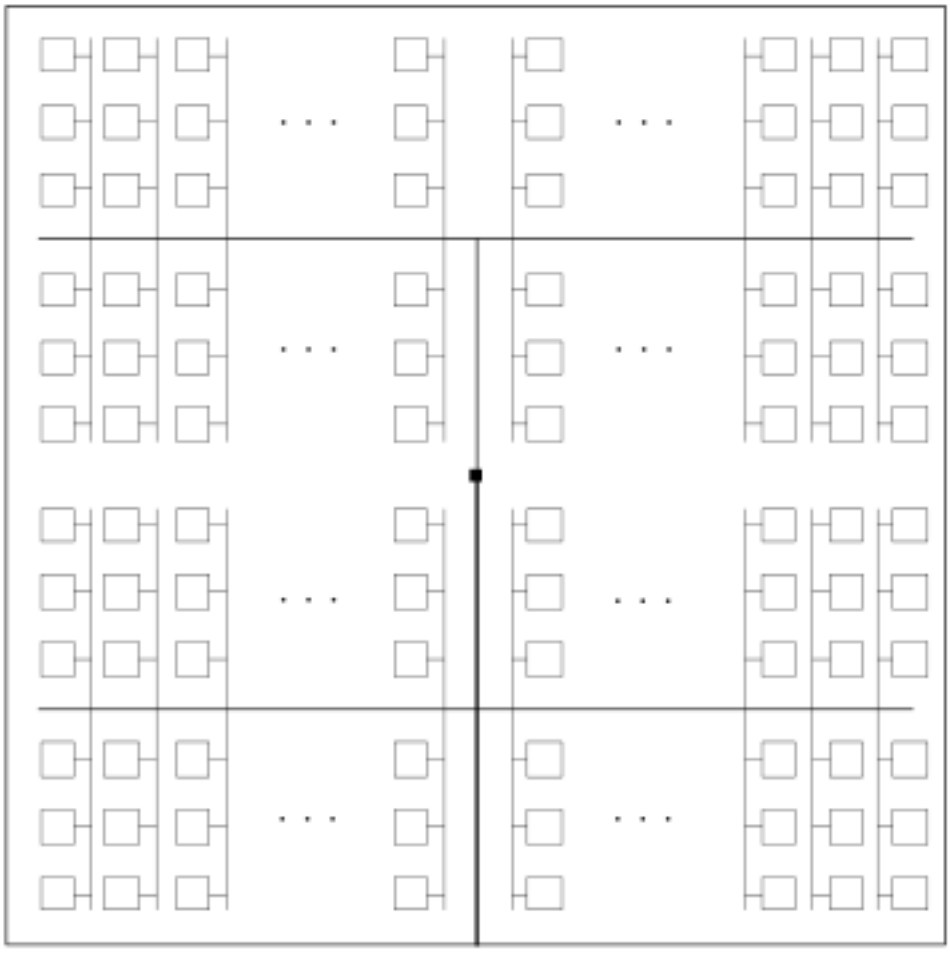


Fig. 1.6: The clock network. Squares represent CLBs, and the wires represent the clock network.

1.6.5 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. 1.7 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 me 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. Unfortuantely, a more rigorous estimation would require some information about the transistor layout.

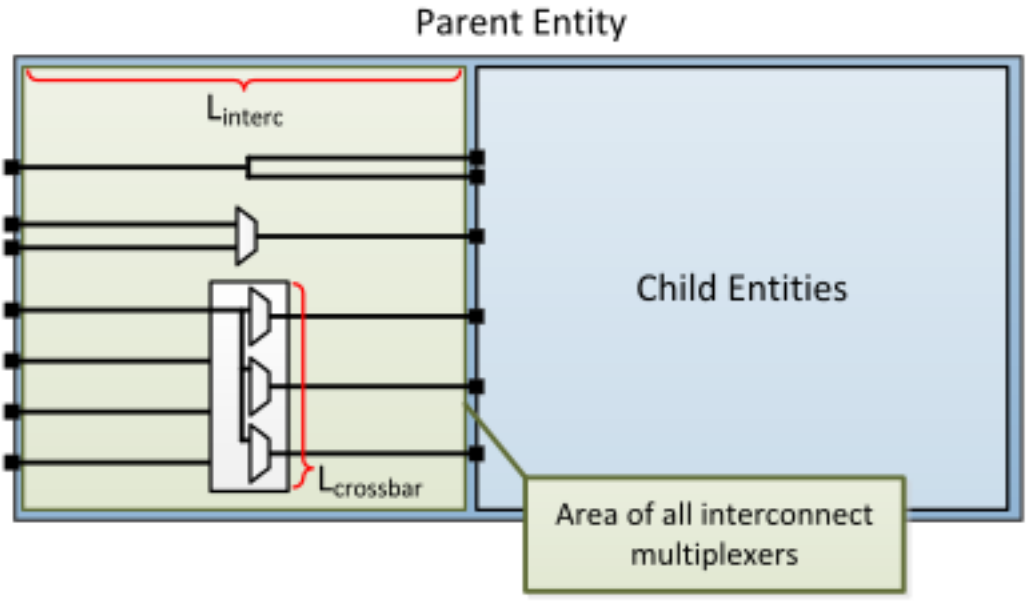


Fig. 1.7: Local interconnect wirelength.

Table 1.3: Local interconnect wirelength and capacitance. C_{inv} is the input capacitance of a minimum-sized inverter.

Connection from Entity Pin to:	Estimated Wirelength	Transistor Capacitance
Direct (Input or Output)	$0.5 \cdot L_{interc}$	0
Many-to-1 (Input or Output)	$0.5 \cdot L_{interc}$	C_{INV}
Complete *m:n* (Input)	$0.5 \cdot L_{interc} + L_{crossbar}$	$n \cdot C_{INV}$
Complete *m:n* (Output)	$0.5 \cdot L_{interc}$	C_{INV}

Table 1.3 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 `pb_type` to an interconnect structure is of length $0.5 \cdot L_{interc}$. 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:

```
<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:

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

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:

```
<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.

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

1.7 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.

1.7.1 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

1.7.2 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. 1.8 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.

1.7.3 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`

1.7.4 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
```

(continues on next page)

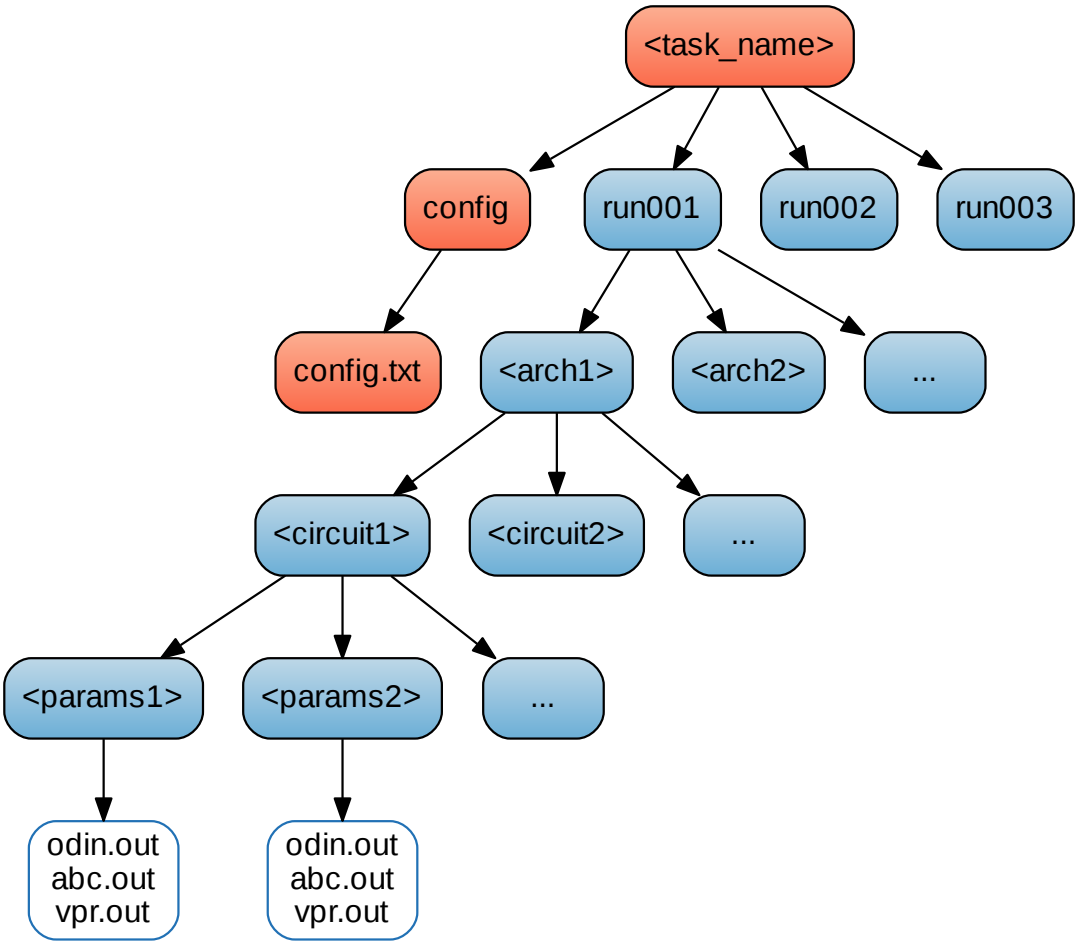


Fig. 1.8: Task directory layout.

(continued from previous page)

```
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 :ref‘run_vtr_flow‘) for the cartesian product of circuits, architectures and script parameters specified in the config file.

1.7.5 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/parse/parse_config`.

1.7.6 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 configurations to be evaluated.

- **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

1.8 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.pl
```

1.8.1 Basic Usage

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

```
run_vtr_flow.pl <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.**

1.8.2 Output

The standard out 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).

1.8.3 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`

-specific_vpr_stage `<vpr_stage>`
Perform only this stage of *VPR*.

To have any time saving effect, previous result files must be kept, as the most recent necessary ones will be moved to the current run directory (use inside tasks only).

Accepted values:

- `pack`
- `place`
- `route`

Default: empty (run all vpr stages)

Note: Specifying the routing stage requires a channel width to also be specified.

-power
Enables power estimation.

See *Power Estimation*

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

See *Technology Properties*

-keep_intermediate_files
Do not delete intermediate files.

-keep_result_files
Do not delete the 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`

1.9 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.pl
```

1.9.1 Basic Usage

Typical usage is:

```
run_vtr_task.pl <task_name1> <task_name2> ...
```

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

1.9.2 Output

Each task will execute the script specified in the configuration file for every benchmark/circuit 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 values. Any entries in the golden results with with the field names `pack_time`, `place_time`, `route_time`, `min_chan_width_route_time`, or `crit_path_route_time` will be summed to determine an estimated runtime for the benchmark. This information will be output in the following format before each circuit/benchmark combination:

```
Current time: Jan-01 01:00 AM.  Expected runtime of next benchmark: 3 minutes
```

Depending on the estimated runtime the units will automatically change between seconds, minutes and hours. This will not be output if the golden results file cannot be found, or if the `-hide_runtime` option is used, or if the underlying script is changed from the default *run_vtr_flow*.

1.9.3 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*).

-p <N>
: Perform parallel execution using N threads.

> **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.:

```
<task_name1>
<task_name2>
<task_name3>
...
```

-hide_runtime
: Do not show runtime estimates.

1.10 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/parse_vtr_flow.pl
```

1.10.1 Usage

Typical usage is:

```
parse_vtr_flow.pl <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.

1.10.2 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.

1.11 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/parse_vtr_task.pl
```

1.11.1 Usage

Typical usage is:

```
parse_vtr_task.pl <task_name1> <task_name2> ...
```

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

1.11.2 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]`.

1.11.3 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
  ```

-check_golden
: 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.

1.12 Parse Configuration

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

1.12.1 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.

1.12.2 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+)
num_io;vpr.out;Netlist inputs pins:\s*(\d+)
```

(continues on next page)

(continued from previous page)

```
num_outputs;vpr.out;Netlist output pins:\s*(\d+)
num_lut0;vpr.out;(\d+) LUTs of size 0
num_lut1;vpr.out;(\d+) LUTs of size 1
num_lut2;vpr.out;(\d+) LUTs of size 2
num_lut3;vpr.out;(\d+) LUTs of size 3
num_lut4;vpr.out;(\d+) LUTs of size 4
num_lut5;vpr.out;(\d+) LUTs of size 5
num_lut6;vpr.out;(\d+) LUTs of size 6
unabsorb_ff;vpr.out;(\d+) FFs in input netlist not absorbable
num_memories;vpr.out;Netlist memory blocks:\s*(\d+)
num_mult;vpr.out;Netlist mult_36 blocks:\s*(\d+)
equiv;abc.out;Networks are (equivalent)
error;output.txt;error=(.*)

%include "my_other_metrics.txt"     #Include metrics from the file 'my_other_metrics.
↪txt'
```

1.13 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.pl -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.

1.13.1 Task Configuration

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

1.13.2 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.

1.13.3 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 requiremnt 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:

```
%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.

1.13.4 Example File

```
vpr_status;Equal()                          #Pass if precisely equal
vpr_seconds;RangeAbs(0.80,1.40,2)           #Pass if within -20%, or +40%, or absolute␣
↪value less than 2
num_pre_packed_nets;Range(0.90,1.10)        #Pass if withing +/-10%

%include "routing_metrics.txt"              #Import all pass requirements from the file
↪'routing_metrics.txt'
```

CHAPTER 2

FPGA Architecture Description

VTR uses an XML-based architecture description language to describe the targeted FPGA architecture. This flexible description langauge 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 langauge. For a detailed reference on the supported options see the *Architecture Reference*.

2.1 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`.

2.1.1 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>`
- `<layout>`
- `<device>`
- `<switchlist>`
- `<segmentlist>`
- `<directlist>`
- `<complexblocklist>`

2.1.2 Recognized BLIF Models (<models>)

The <models> tag contains <model name="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.

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:

<port name="string" is_clock="{0 | 1} clock="string" combinational_sink_ports="string1 str

Required Attributes

- **name** – The port name.

Optional Attributes

- **is_clock** – Indicates if the port is a clock. 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 combinationally 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:

```
<models>
  <model name="single_port_ram">
    <input_ports>
      <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>
      <port name="out" clock="clk"/>
    </output_ports>
  </model>

  <model name="adder">
    <input_ports>
      <port name="a" combinational_sink_ports="cout sumout"/>
      <port name="b" combinational_sink_ports="cout sumout"/>
      <port name="cin" combinational_sink_ports="cout sumout"/>
    </input_ports>
    <output_ports>
      <port name="cout"/>
      <port name="sumout"/>
    </output_ports>
```

(continues on next page)

(continued from previous page)

```
  </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 combinationally 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 combinationally 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*

2.1.3 Global FPGA Information

<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.

<segmentlist>content</segmentlist>
Content inside this tag contains a group of tags that specify the types of wire segments and their properties.

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

2.1.4 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 `W/2 - w/2`. For a device width of 10 and a block type of width 3, this would be evaluated as $\lfloor \frac{W}{2} \rfloor - \lfloor \frac{w}{2} \rfloor = \lfloor \frac{10}{2} \rfloor - \lfloor \frac{3}{2} \rfloor = 5 - 1 = 4$.

Grid Location Tags

<fill type="string" priority="int"/>

Required Attributes

- **type** – The name of the top-level complex block type (i.e. <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"/>
```

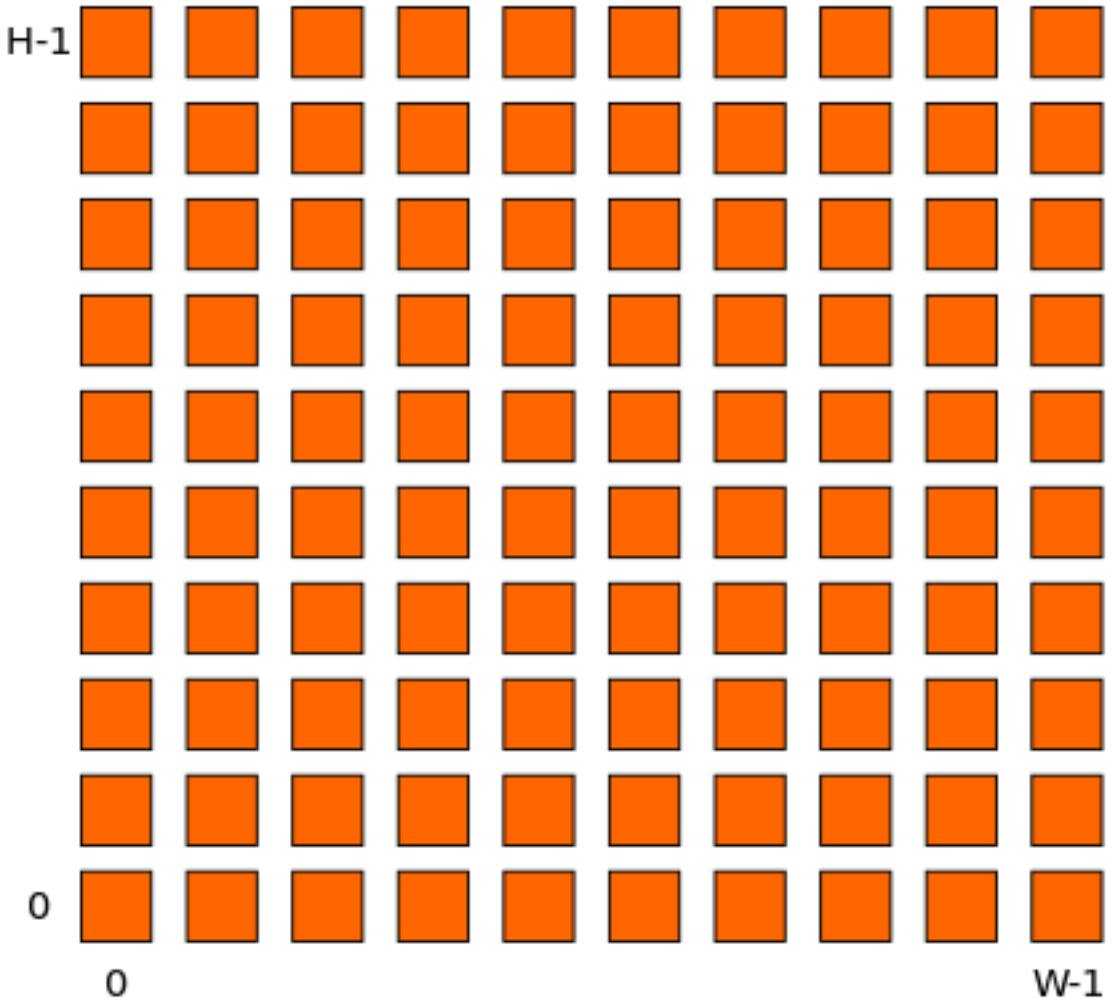


Fig. 2.1: <fill> CLB example

<perimeter 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 perimeter of the device (i.e. edges) to the specified block type.

Note: The perimeter includes the corners

Example:

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

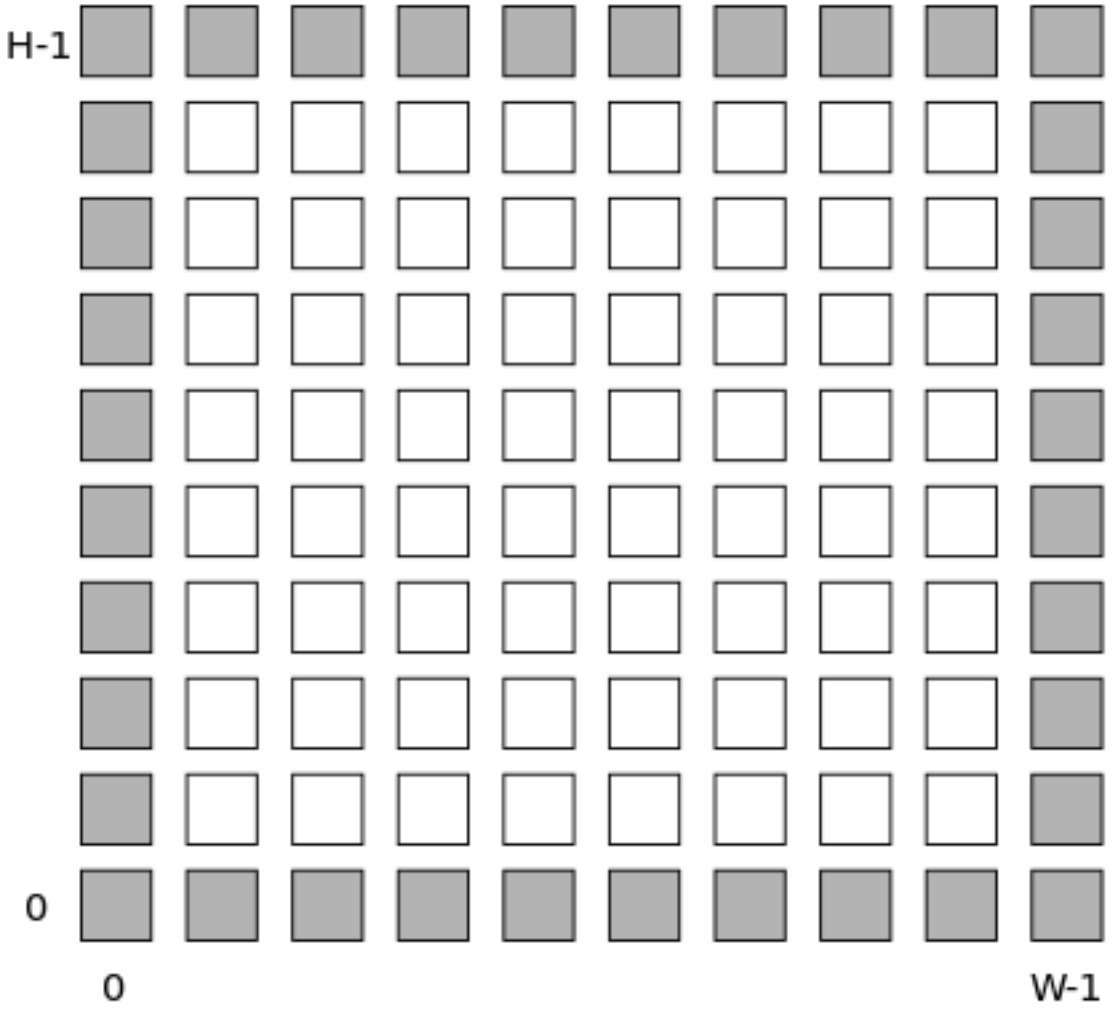


Fig. 2.2: <perimeter> io example

```
<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:

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

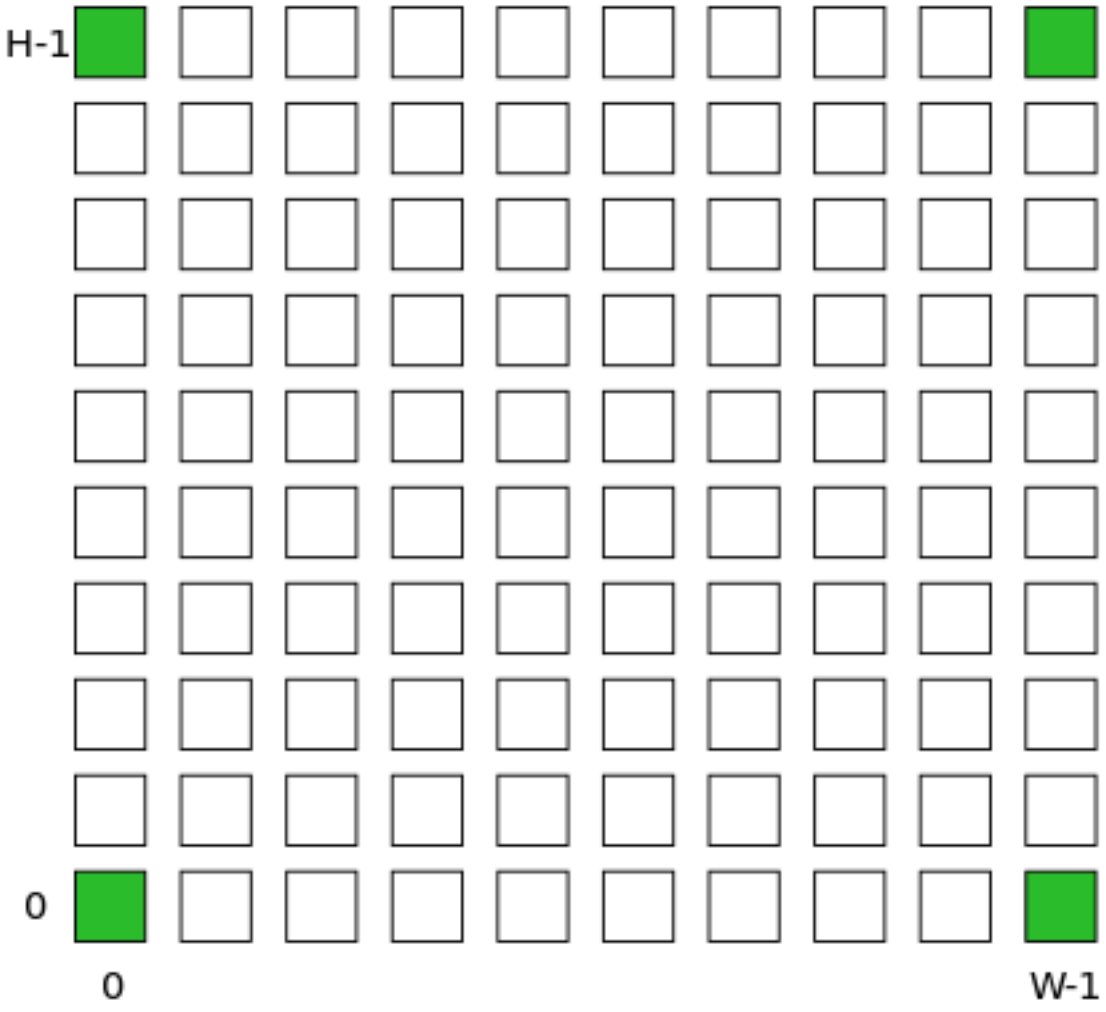


Fig. 2.3: <corners> PLL example

<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:

```
<!-- 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"/>
```

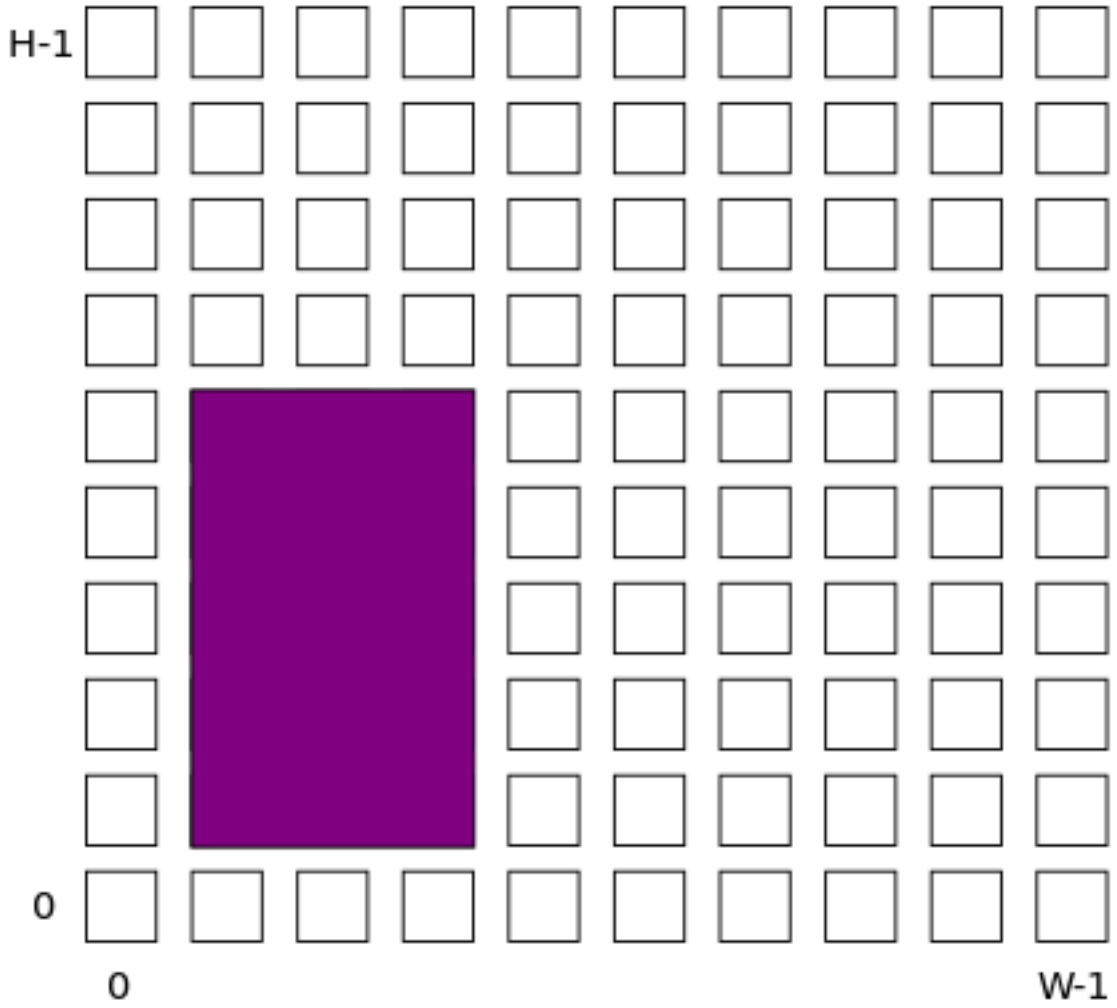


Fig. 2.4: <single> PCIE example

<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`
- **incry** – 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:

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

Example:

```
<!-- Create IO's 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`
- **incry** – An expression specifying the horizontal increment between block instantiations within the region.

  Default: `w`

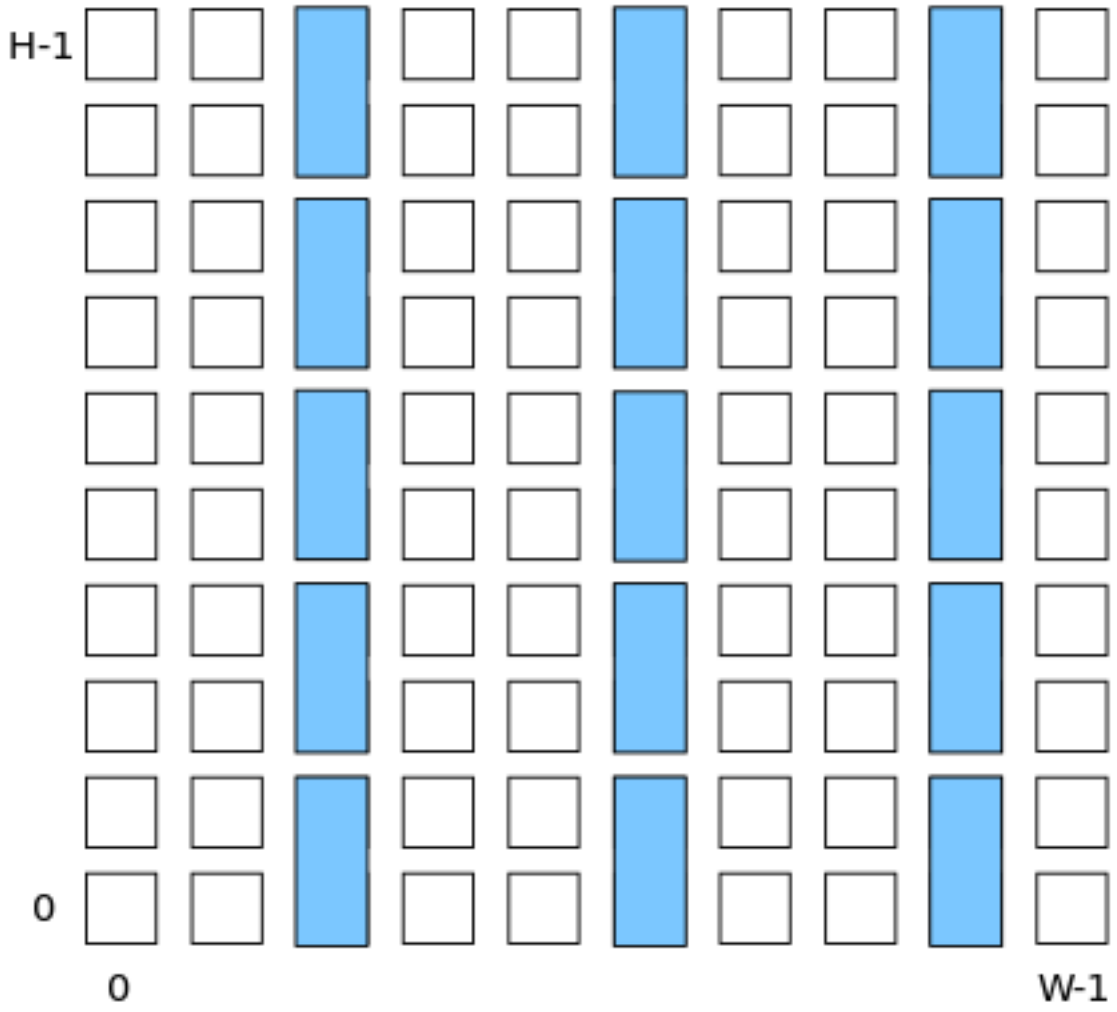


Fig. 2.5: <col> RAM example

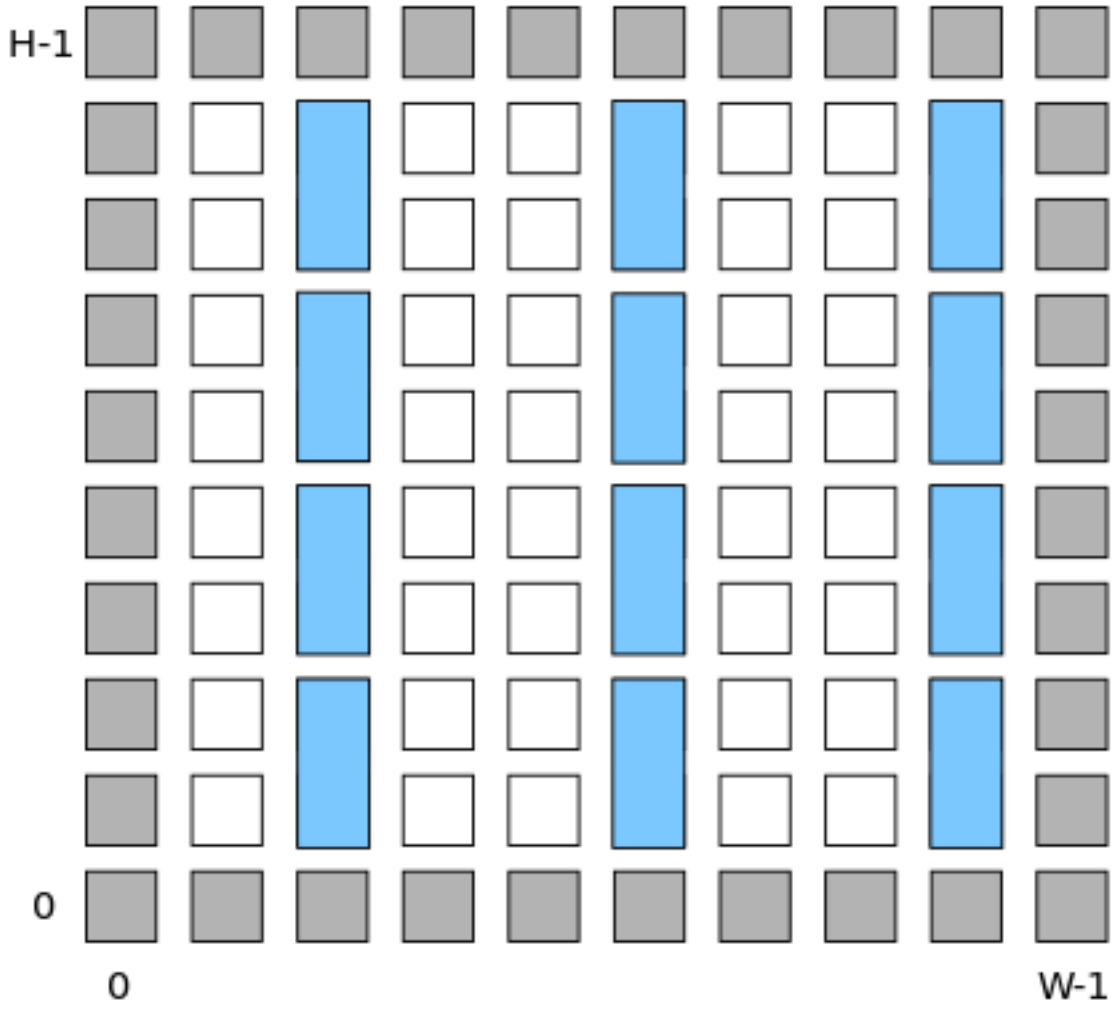


Fig. 2.6: <col> RAM and <perimeter> io example

Creates a row of the specified block type at `starty`.

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:

```
<!-- 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"/>
```

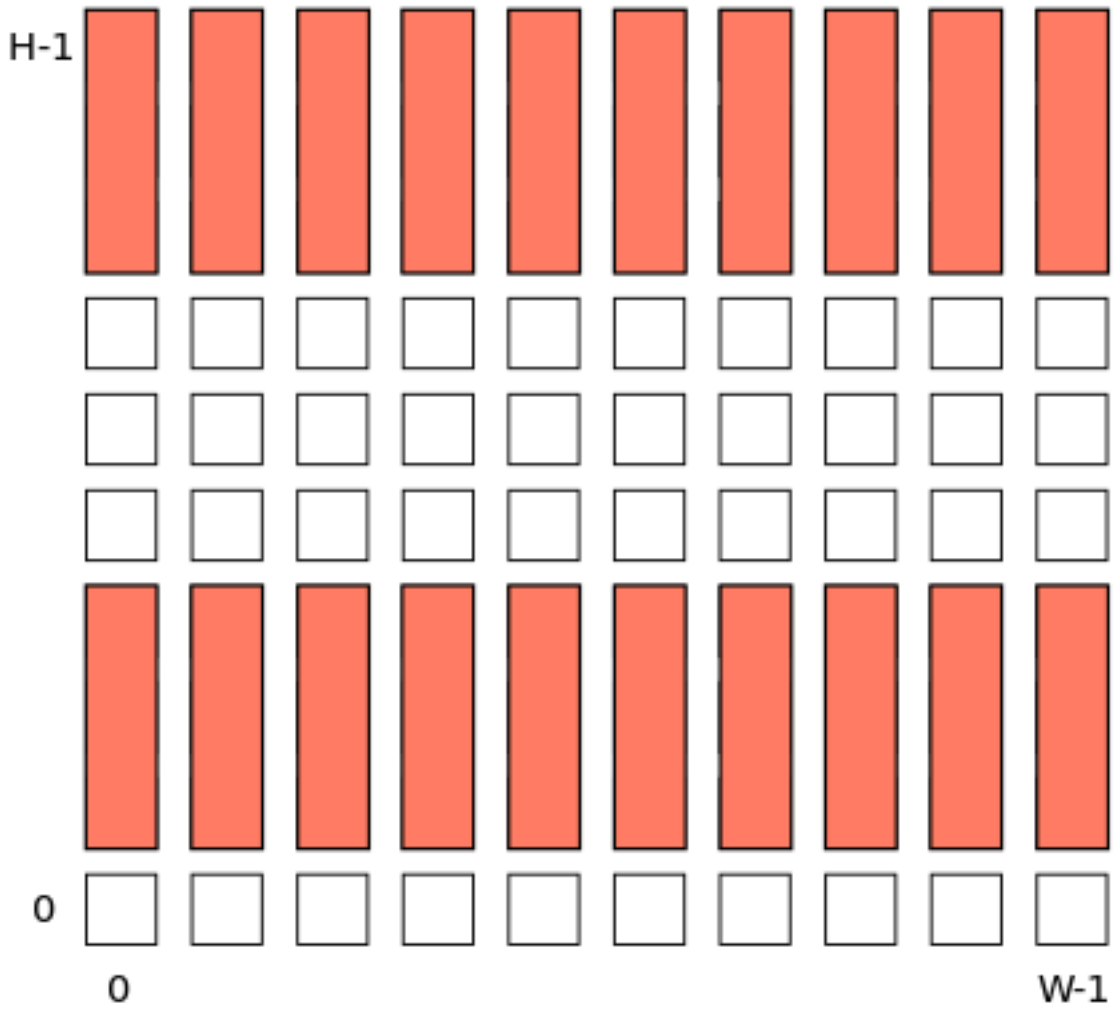


Fig. 2.7: <row> DSP example

`<region type="string" priority="int" startx="expr" endx="expr repeatx="expr" incrx="expr" s`

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 horizontal increment between block instantiations within the region.

  Default: `h`

Fills the rectangular region defined by (`startx`, `starty`) and (`endx`, `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 $x = startx + k_1 * repeatx$, is satisified for any positive integer k_1.

If `repeaty` is specified the region will be repeated wherever $y = starty + k_2 * repeaty$, is satisified for any positive integer k_2.

Example:

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

Example:

```
<!-- Create RAMs every 2nd column withing 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:

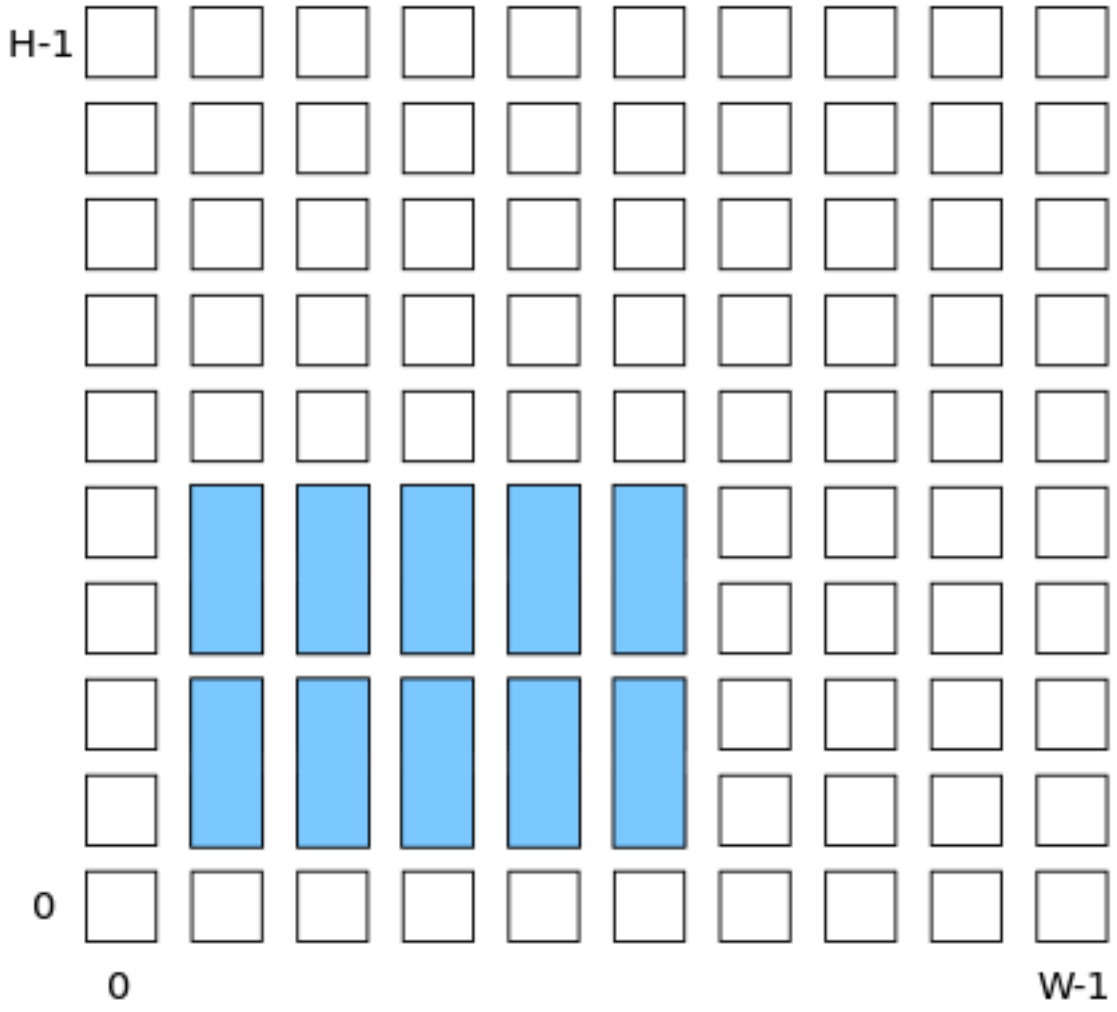


Fig. 2.8: <region> RAM example

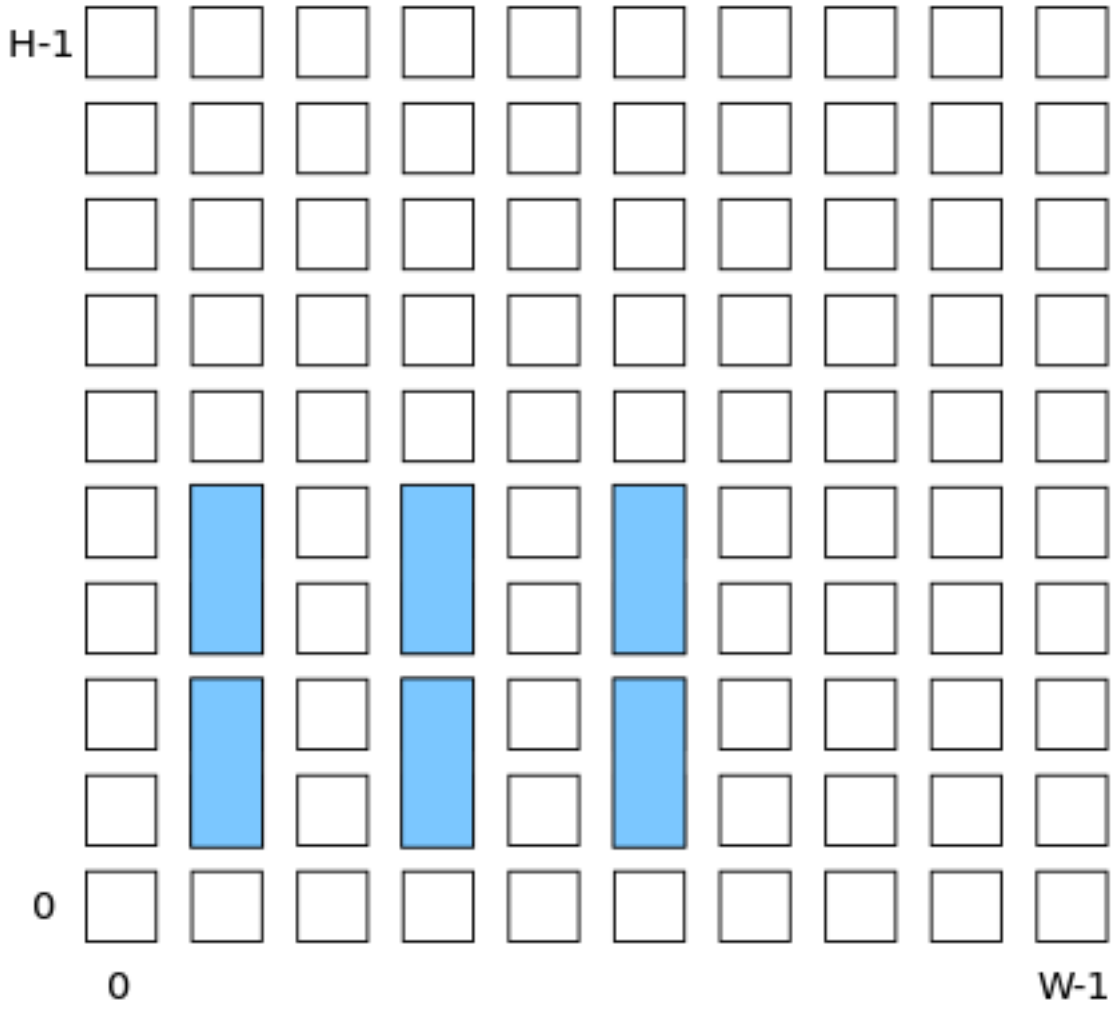


Fig. 2.9: <region> RAM increment example

```
<!-- 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"/>
```

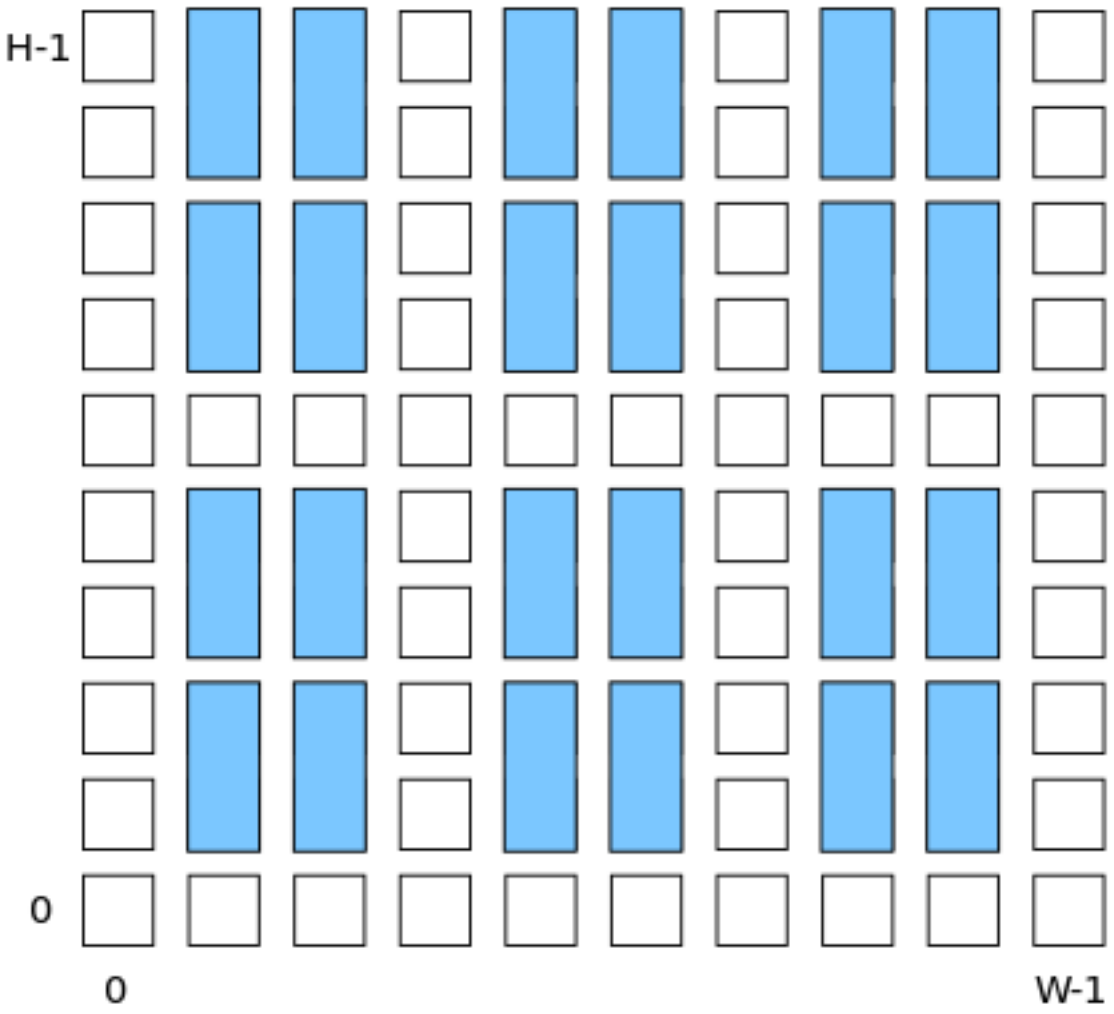


Fig. 2.10: <region> RAM repeat example

Example:

```
<!-- 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" incrx="W/4" incry="W/4"␣
↪priority="3"/>
```

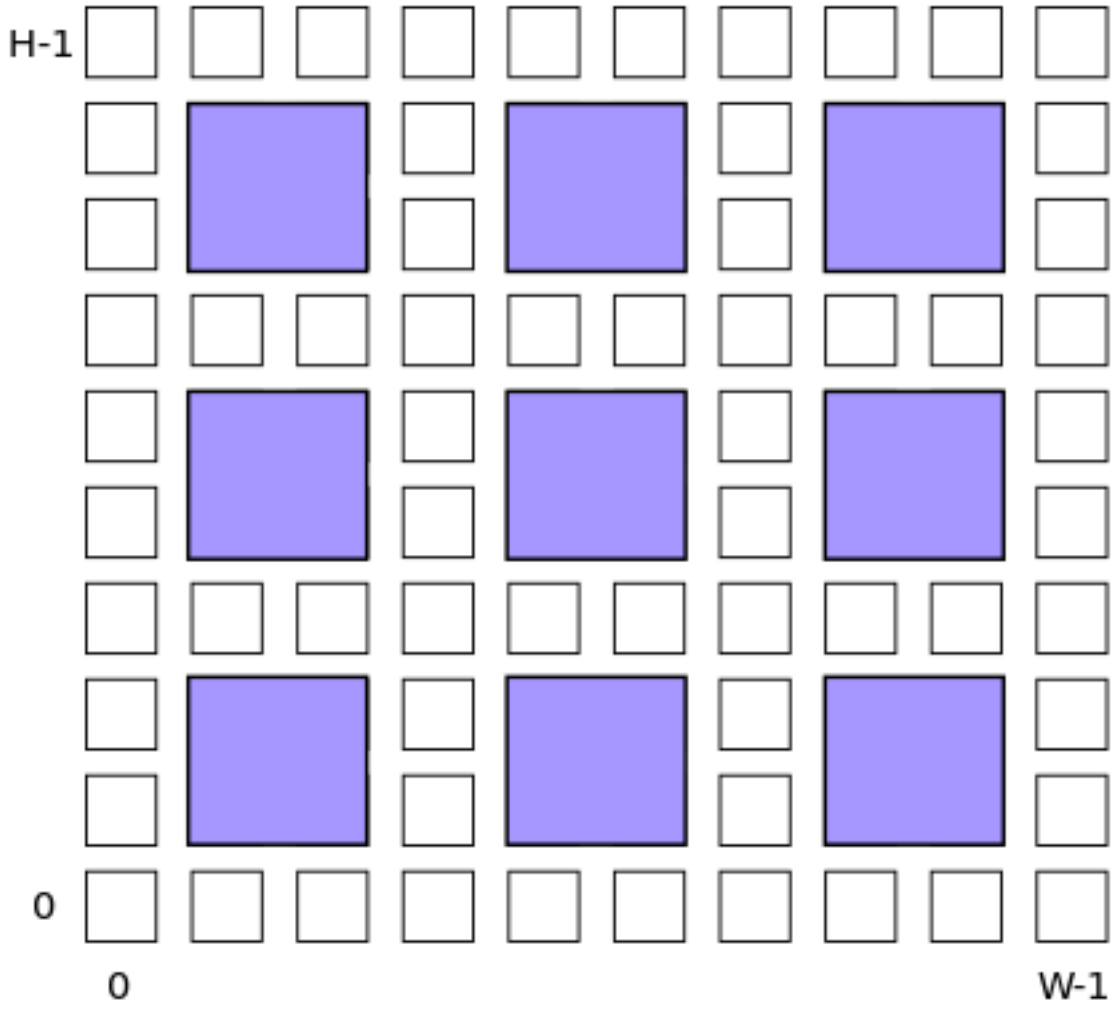


Fig. 2.11: <region> NoC mesh example

Grid Layout Example

```
<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 IOs-->
        <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>
```

2.1.5 FPGA Device Information

The tags within the <device> tag are:

<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.

<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).

Required Yes

<area grid_logic_tile_area="float"/>

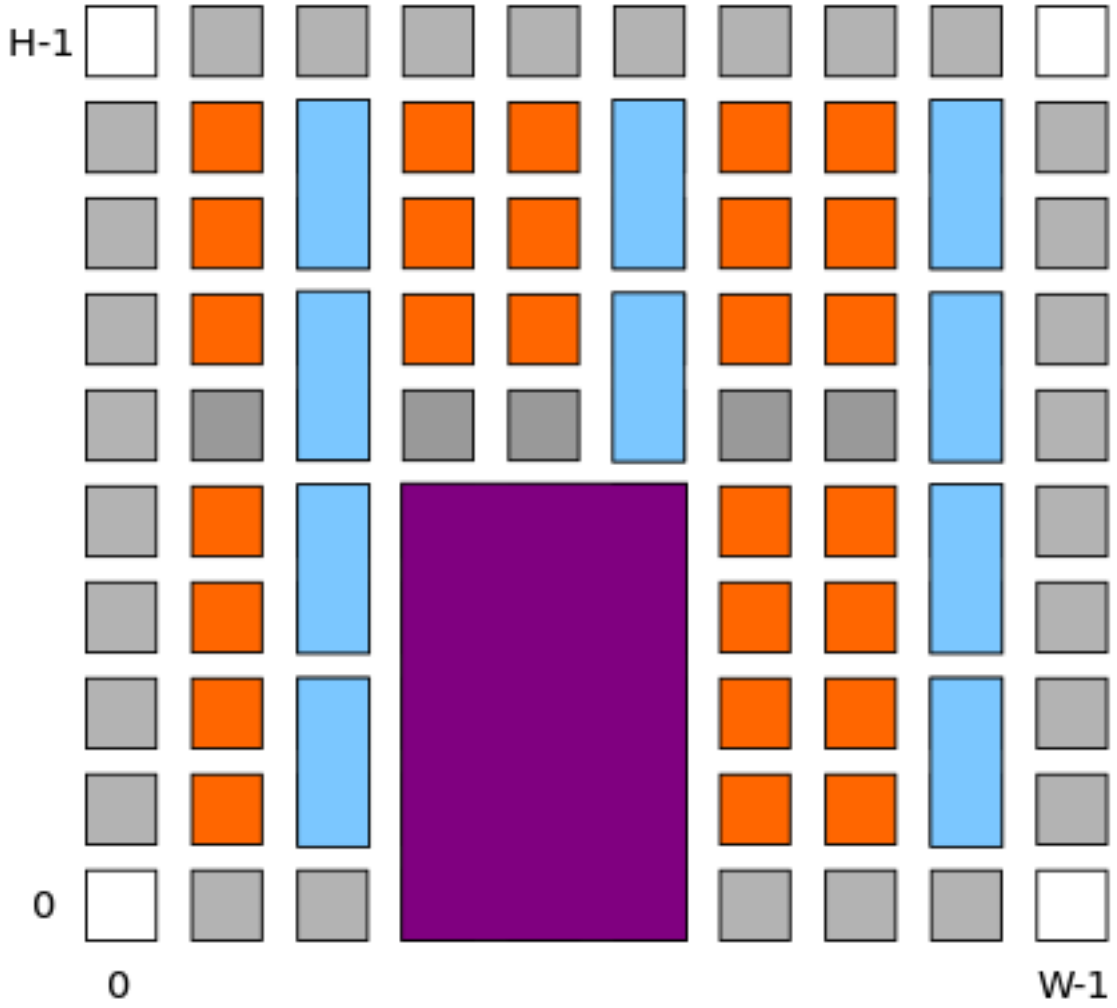


Fig. 2.12: Example FPGA grid

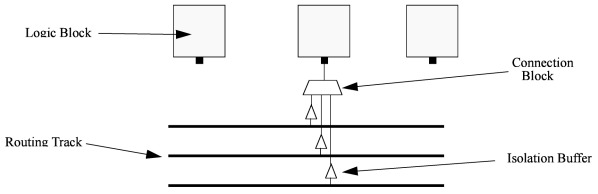


Fig. 2.13: Input Pin Diagram.

Required Yes

Used for an area estimate of the amount of area taken by all the functional blocks. This specifies the area of a 1x1 tile excluding routing.

<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*.

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

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*.

2.1.6 Switches

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

<switch type="{mux|tristate|pass_gate|short|buffer}" name="string" R="float" Cin="float" C

Describes a switch in the routing architecture.

Example:

```
<switch type="mux" name="my_awsome_mux" R="551" Cin=".77e-15" Cout="4e-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.

  Configurablity

  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

- **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: $T_{del} + R \cdot C_{out}$.

  The 'switch' includes both the mux and buffer mux type switches.

  Note: Required if no <Tdel> tags are specified

  Note: A <switch>'s resistance (R) and output capacitance (Cout) 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 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.

`<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" 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, all the channels in the FPGA must have the same width.

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

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. 2.14 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.

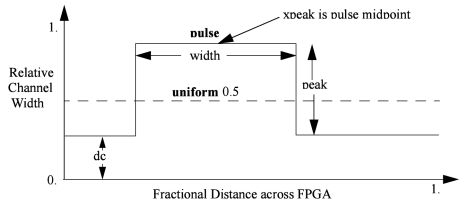


Fig. 2.14: Channel Distribution

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:

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

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

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

See also:

<x distr>

2.1.7 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" capacity="int" width="int" height="

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>
            ...
        </pb_type>
        ...
    </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 Only:

Optional Attributes

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

  Default: 1

  For example:

  ```
  <pb_type name="IO" capacity="2"/>
      ...
  </pb_type>
  ```

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

- **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 of the block type

  Default: from the <area> tag

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:

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

  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:

`<input name="string" num_pins="int" equivalent="{none|full}" is_non_clock_global="{true|fa`

Defines an input port. Multple 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.

  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 otherhand, 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 equivalance 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. Multple 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 the router must take special care to ensure it does not make use of LUT/BLE outputs which are being used to route signals strictly within the current cluster (to do so would be illegal). This is handled by special code in the router when this type of equivalence is specified.

Default: `none`

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

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

<mode name="string">

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.

For example:

```
<!--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>
```

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

The following tags are unique to the top level <pb_type> of a complex logic block. They describe how a complex block interfaces with the inter-block world.

<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 unidirection 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 Fc specification:

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

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:

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

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 (eg. carry chains), you would use the <fc_override> tag, within the <fc> tag, to specify them:

```
<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 (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.

<pinlocations pattern="{spread|perimeter|custom}">

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.

<loc side="{left|right|bottom|top}" xoffset="int" yoffset="int">name_of_complex_logic_

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.

<switchblock_locations pattern="{external_full_internal_straight|all|external|internal|none

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

- **`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 electrcially shorted to their 'straight-through' connections.

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

```
<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.

```
<switchblock_locations pattern="external_full_internal_straight" internal_switch=
↪"electrical_short"/>
```

`<sb_loc type="{full|straight|turns|none}" xoffset="int" yoffset="int", switch_override`

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 `staight` 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

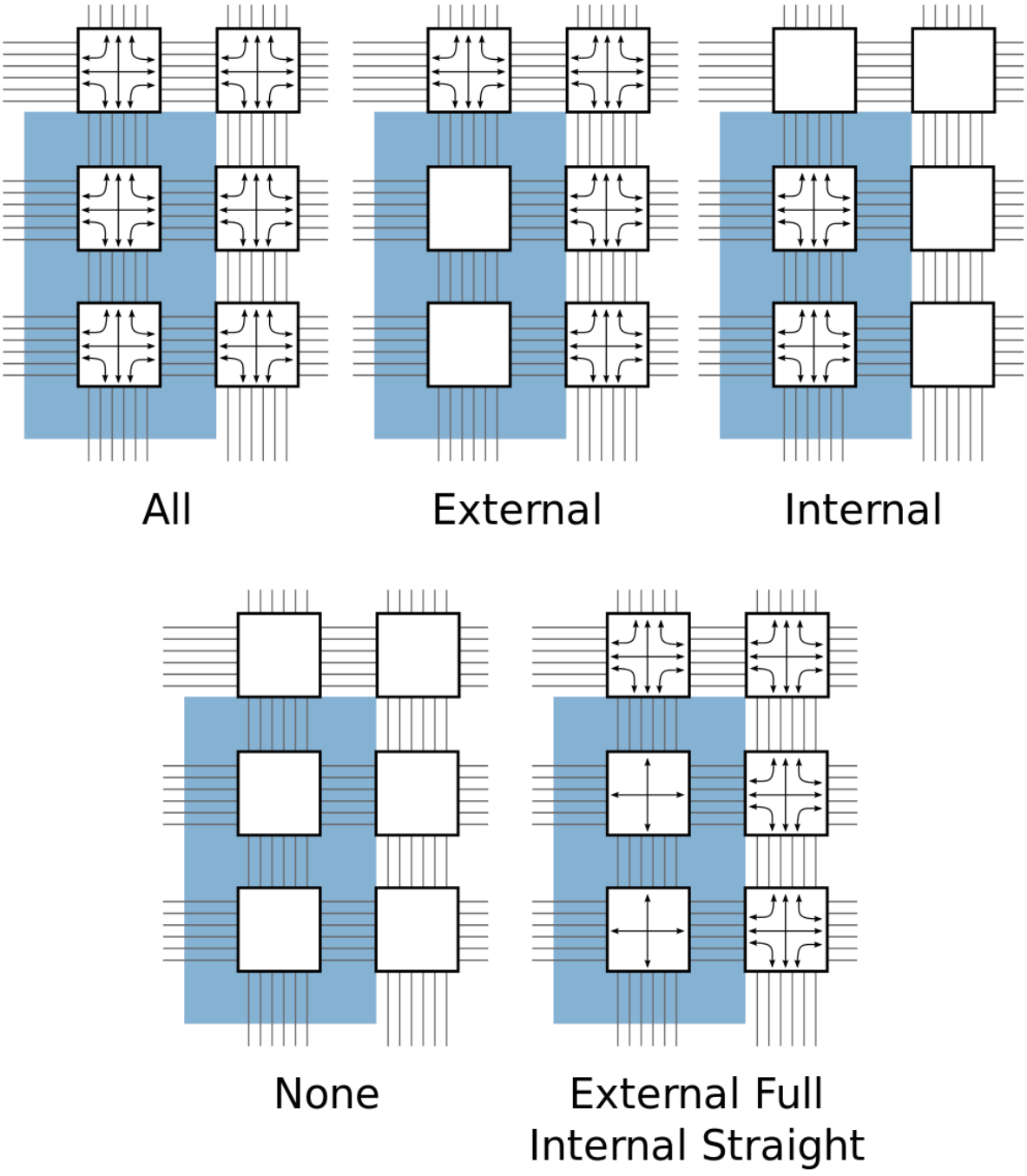


Fig. 2.15: Switchblock Location Patterns for a width = 2, height = 3 complex block

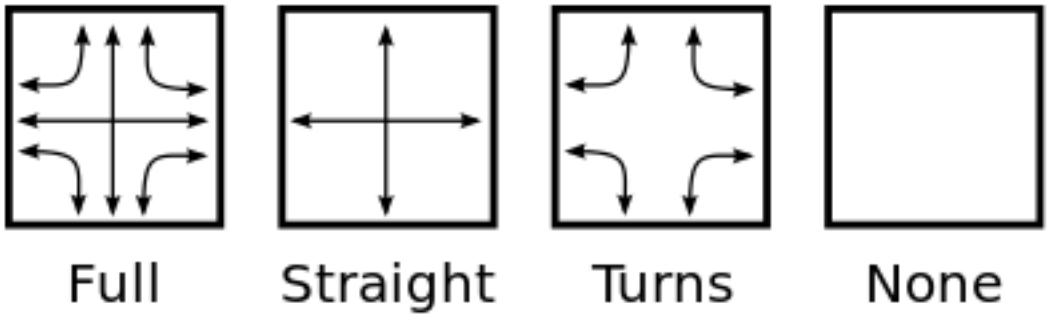


Fig. 2.16: Switchblock Types

- **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.

  Default: The default switch for the wire

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. 2.15), we can use a custom pattern to specify an architecture equivalent to the 'Electrically Shorted Internal Straight Connections' example:

```
<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/>
```

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:

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

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 (eg. 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.

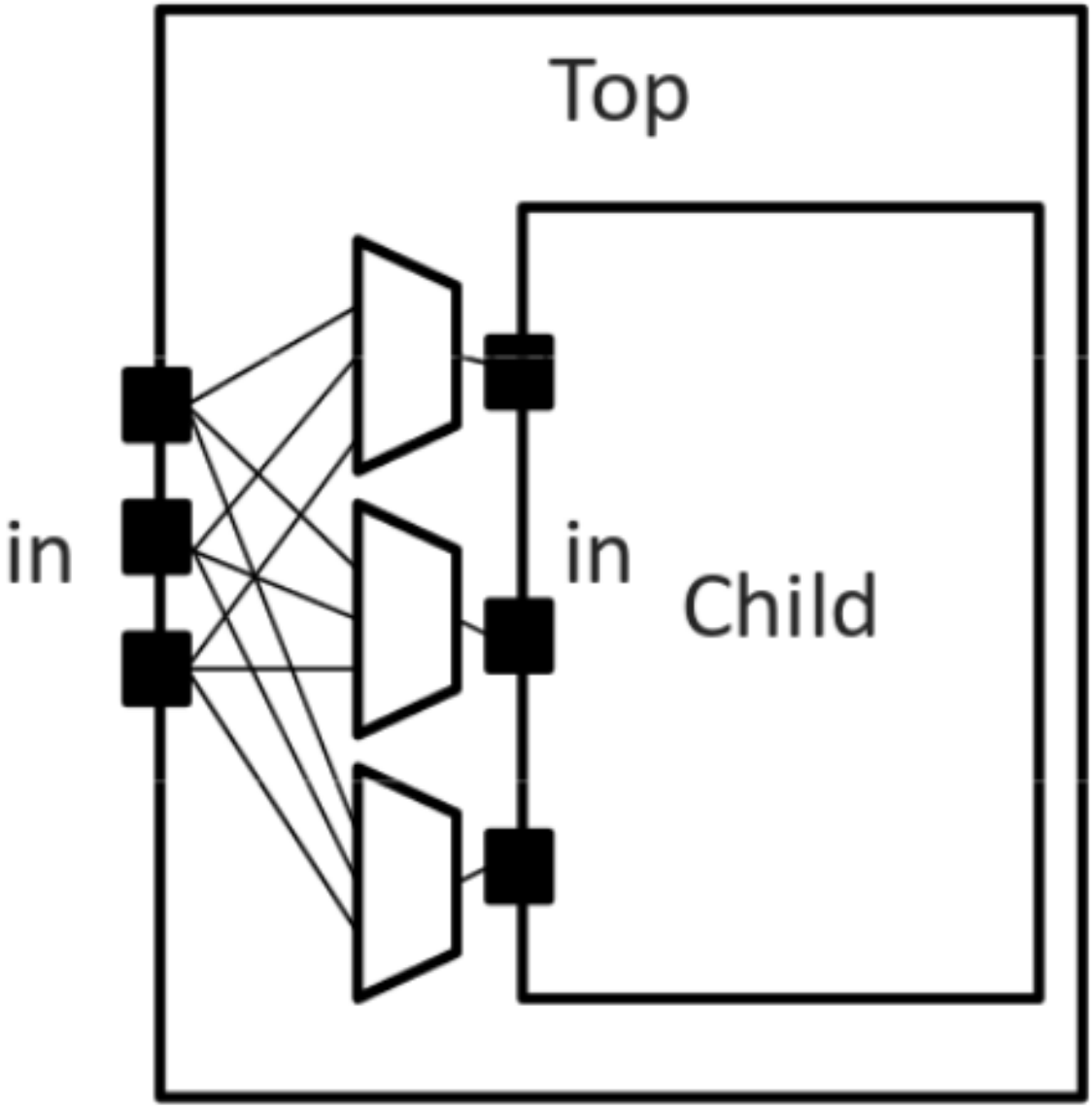


Fig. 2.17: Complete interconnect example.

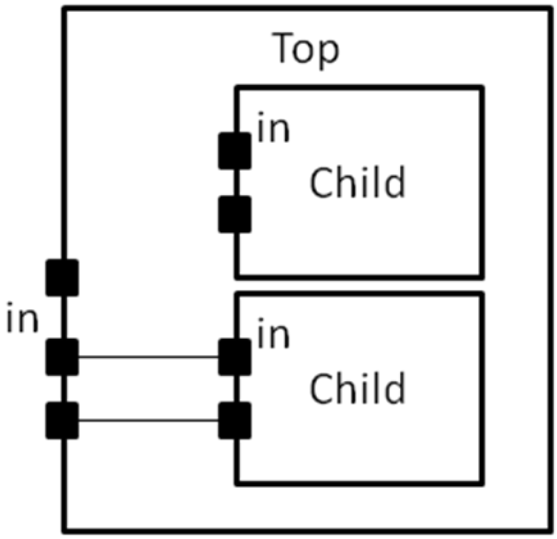


Fig. 2.18: Direct interconnect example.

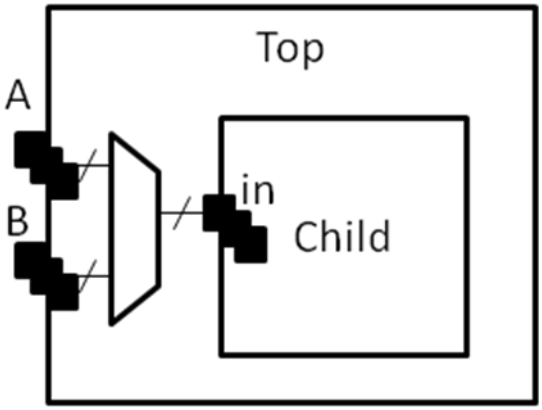


Fig. 2.19: Mux interconnect example.

<pack_pattern name="string" in_port="string" out_port="string"/>

> **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, lets 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. Lets 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:

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

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.

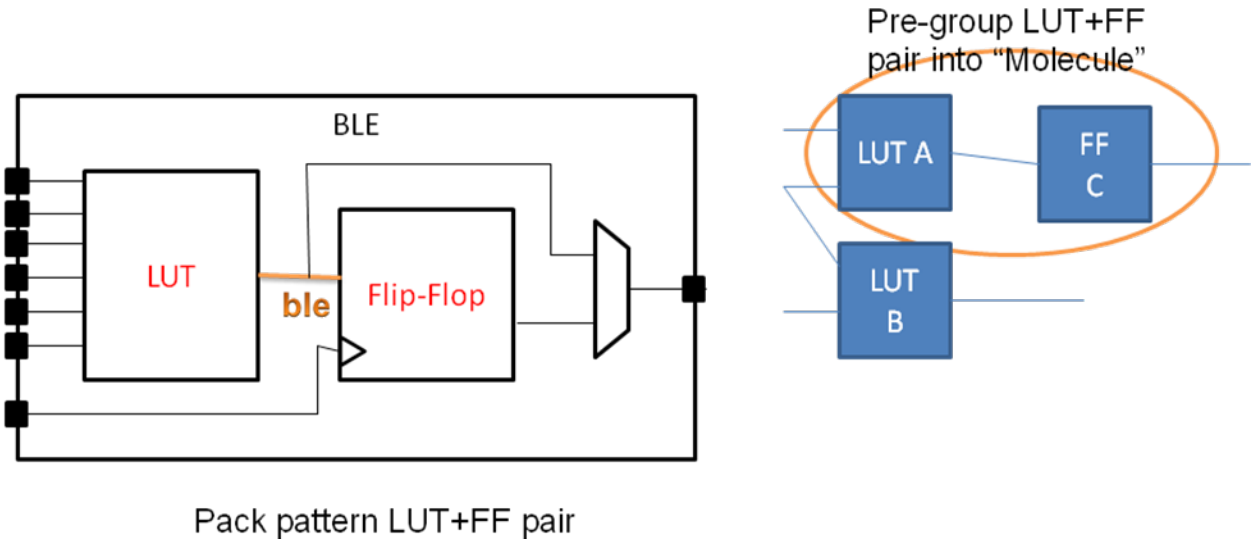


Fig. 2.20: Pack Pattern Example.

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 with in `pb_type`, `complete`, `direct`, or `mux` tags as follows:

`<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

---

`<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`:

```
<delay_matrix type="max" in_port="in" out_port="out">
    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
</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`:

```
<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>`

`<dynamic_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_stat`

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.

2.1.8 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"

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.

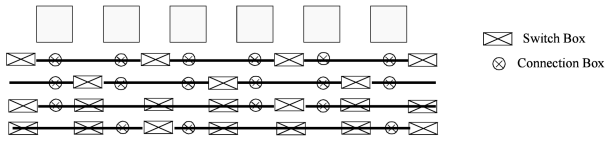


Fig. 2.21: Switch block and connection block pattern example with four tracks per channel

<sb type="pattern">int list</sb>

This tag describes the switch block depopulation (as illustrated in Fig. 2.21) 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 a 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.

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. 2.21) 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"/>

Note: For BIDIRECTIONAL only.

Required Attributes

- **name** – Name of the switch type used by block pins to drive this type of segment.

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.

2.1.9 Clocks

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.

2.1.10 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.

`<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`.

`<buffers logical_effort_factor="float"/>`

Required Attributes

- **`logical_effort_factor`** – **Default:** `4`.

2.1.11 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.

<direct name="string" from_pin="string" to_pin="string" x_offset="int" y_offset="int" z_of

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.
- **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.

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.

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:

```
<direct name="adder_carry" from_pin="clb.cout" to_pin="clb.cin" x_offset="0" y_
↪offset="-1" z_offset="0"/>
```

2.1.12 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:

```
<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">
```

(continues on next page)

(continued from previous page)

```
    ... 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).

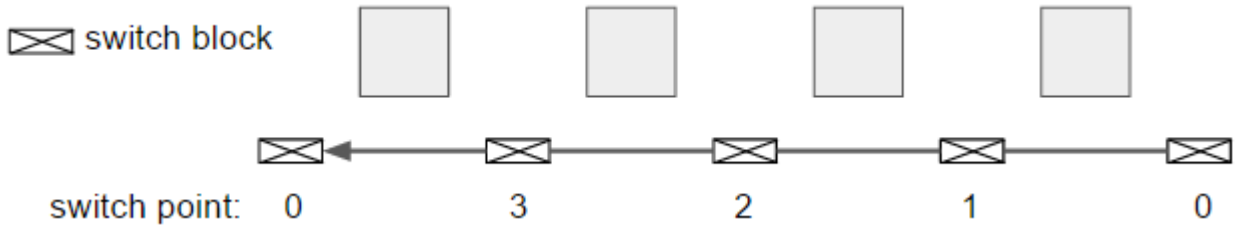


Fig. 2.22: Switch point diagram.

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.

`<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.

`<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.

<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.

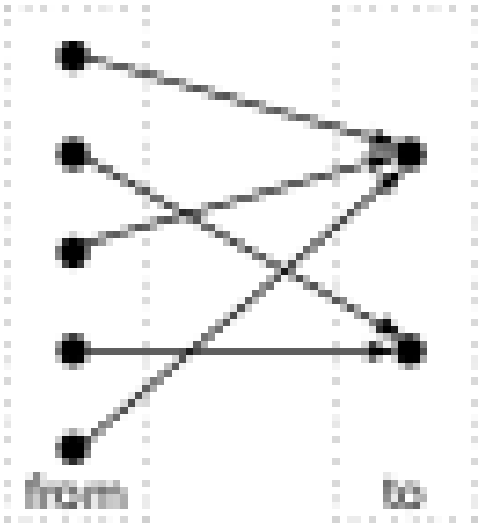
Defined under the <switchfuncs> XML node, one or more <func...> entries is used to specify permutation functions that connect different sides of a switch block.

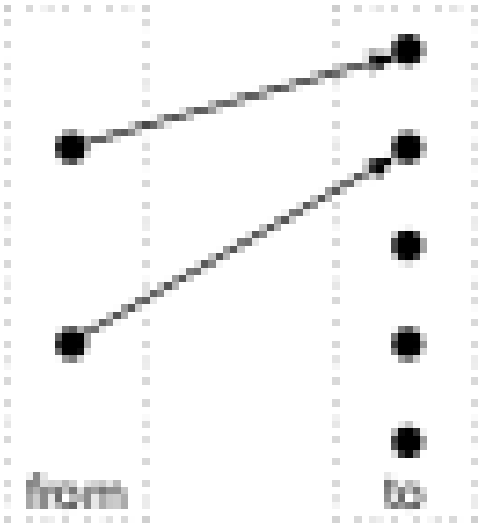
<wireconn num_conns_type="{from,to,min,max}" from_type="string, string, string, ..." to_typ

Required Attributes

- **num_conns_type** – Specifies how many connections should be created between the from_type/from_switchpoint set and the to_type/to_switchpoint set.
  - from – Creates number of switchblock edges equal to the 'from' set size.

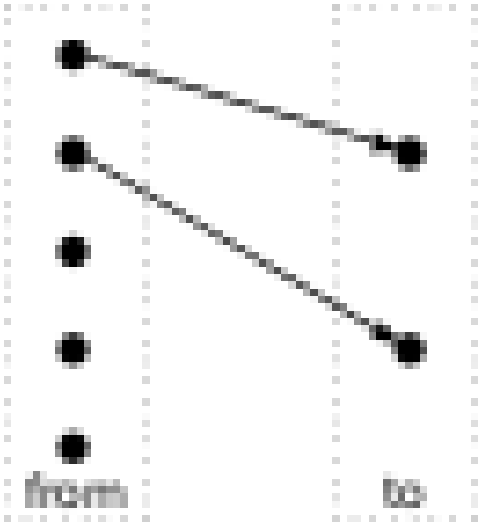
This ensures that each element in the 'from' set is connected to an element of the 'to' set. However it may leave some elements of the 'to' set either multiply-connected or disconnected.

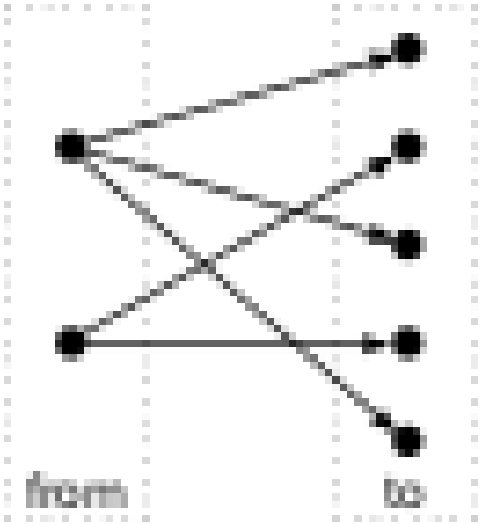




– `to` – Creates number of switchblock edges equal to the 'to' set size size.

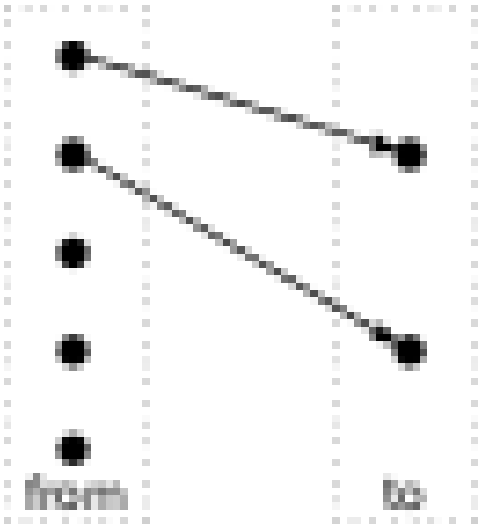
  This ensures that each element of the 'to' set is connected to precisely one element of the 'from' set. However it may leave some elements of the 'from' set either multiply-connected or disconnected.

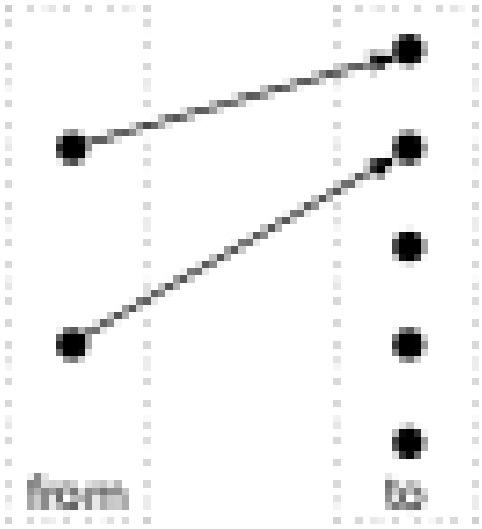




– `min` – Creates number of switchblock edges equal to the minimum of the 'from' and 'to' set sizes.

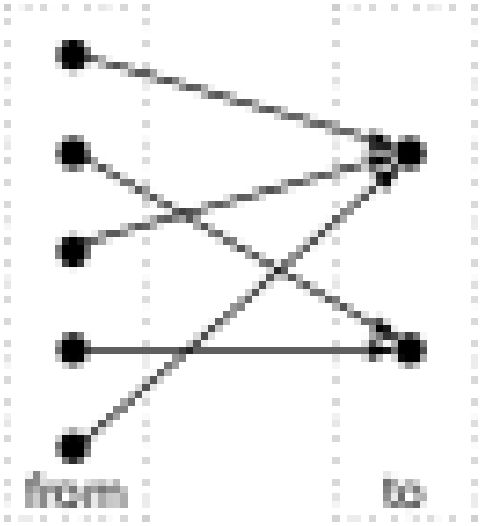
  This ensures *no* element of the 'from' or 'to' sets is connected to multiple elements in the opposing set. However it may leave some elements in the larger set disconnected.

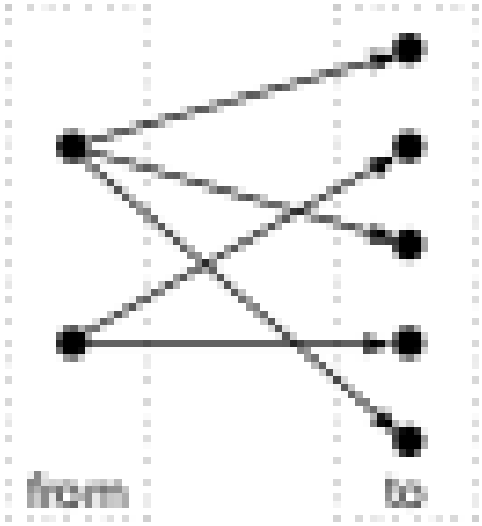




– `max` – Creates number of switchblock edges equal to the maximum of the 'from' and 'to' set sizes.

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.





Optional Attributes

- **`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.

<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.

2.2 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.

```
1  <!-- VPR Architecture Specification File --><!-- Quick XML Primer:
2      * Data is hierarchical and composed of tags (similar to HTML)
3      * 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.
4      * Inside a start tag you may specify attributes in the form key="value". Refer to␣
   ↪manual for the valid attributes for each element.
5      * Comments may be included anywhere in the document except inside a tag where it
   ↪'s attribute list is defined.
6      * Comments may contain any characters except two dashes.
7  --><!-- Architecture based off Stratix IV
8      Use closest ifar architecture: K06 N10 45nm fc 0.15 area-delay optimized, scale␣
   ↪to 40 nm using linear scaling
9      n10k06l04.fc15.area1delay1.cmos45nm.bptm.cmos45nm.xml
10     * 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
11       * delays multiplied by 40/45 to normalize for process difference between␣
   ↪stratix 4 and 45 nm technology (called full scaling)
12
13       Use delay numbers off Altera device handbook:
14
15       http://www.altera.com/literature/hb/stratix-iv/stx4_5v1.pdf
16       http://www.altera.com/literature/hb/stratix-iv/stx4_siv51004.pdf
17       http://www.altera.com/literature/hb/stratix-iv/stx4_siv51003.pdf
18       multipliers at 600 MHz, no detail on 9x9 vs 36x36
19       * datasheets unclear
20         * claims 4 18x18 independant multipliers, following test indicates that this␣
   ↪is not the case:
21           created 4 18x18 mulitpliers, logiclocked them to a single DSP block, compile
22           result - 2 18x18 multipliers got packed together, the other 2 got ejected␣
   ↪out of the logiclock region without error
23           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)
24       --><architecture>
25    <!-- ODIN II specific config -->
26    <models>
27      <model name="multiply">
28        <input_ports>
29          <port name="a" combinational_sink_ports="out"/>
30          <port name="b" combinational_sink_ports="out"/>
31        </input_ports>
32        <output_ports>
33          <port name="out"/>
34        </output_ports>
35      </model>
36      <model name="single_port_ram">
37        <input_ports>
38          <port name="we" clock="clk"/>
39          <!-- control -->
40          <port name="addr" clock="clk"/>
```

(continues on next page)

(continued from previous page)

```
        <!-- address lines -->
        <port name="data" 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="out" clock="clk"/>
        <!-- output can be broken down into smaller bit widths minimum size 1 -->
      </output_ports>
    </model>
    <model name="dual_port_ram">
      <input_ports>
        <port name="we1" clock="clk"/>
        <!-- write enable -->
        <port name="we2" clock="clk"/>
        <!-- write enable -->
        <port name="addr1" clock="clk"/>
        <!-- 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>
  <!-- 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" ipin_mux_trans_size=
↪"1.222260"/>
```

(continues on next page)

(continued from previous page)

```
    <timing C_ipin_cblock="0.000000e+00" T_ipin_cblock="7.247000e-11"/>
    <area grid_logic_tile_area="14813.392"/>
    <!--area is for soft logic only-->
    <chan_width_distr>
      <io width="1.000000"/>
      <x distr="uniform" peak="1.000000"/>
      <y distr="uniform" peak="1.000000"/>
    </chan_width_distr>
    <switch_block type="wilton" fs="3"/>
  </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"/>
  </switchlist>
  <segmentlist>
    <segment freq="1.000000" length="4" type="unidir" Rmetal="0.000000" Cmetal="0.000000e+00">
      <mux name="0"/>
      <sb type="pattern">1 1 1 1 1</sb>
      <cb type="pattern">1 1 1 1</cb>
    </segment>
  </segmentlist>
  <complexblocklist>
    <!-- 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 -->
    <pb_type name="io" capacity="8">
      <input name="outpad" num_pins="1"/>
      <output name="inpad" num_pins="1"/>
      <clock name="clock" num_pins="1"/>
      <!-- IOs can operate as either inputs or outputs -->
      <mode name="inpad">
        <pb_type name="inpad" blif_model=".input" num_pb="1">
          <output name="inpad" num_pins="1"/>
        </pb_type>
        <interconnect>
          <direct name="inpad" input="inpad.inpad" output="io.inpad">
            <delay_constant max="4.243e-11" in_port="inpad.inpad" out_port="io.inpad"/>
          </direct>
        </interconnect>
      </mode>
      <mode name="outpad">
        <pb_type name="outpad" blif_model=".output" num_pb="1">
          <input name="outpad" num_pins="1"/>
        </pb_type>
        <interconnect>
          <direct name="outpad" input="io.outpad" output="outpad.outpad">
            <delay_constant max="1.394e-11" in_port="io.outpad" out_port="outpad.outpad"/>
          </direct>
        </interconnect>
      </mode>
      <fc in_type="frac" in_val="0.15" out_type="frac" out_val="0.10"/>
      <!-- IOs 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
```

(continues on next page)

(continued from previous page)

```
147         -->
148       <pinlocations pattern="custom">
149         <loc side="left">io.outpad io.inpad io.clock</loc>
150         <loc side="top">io.outpad io.inpad io.clock</loc>
151         <loc side="right">io.outpad io.inpad io.clock</loc>
152         <loc side="bottom">io.outpad io.inpad io.clock</loc>
153       </pinlocations>
154       </pb_type>
155
156     <pb_type name="clb">
157       <input name="I" num_pins="33" equivalent="true"/> <!-- NOTE: Logically␣
    ↪Equivalent -->
158       <output name="O" num_pins="10" equivalent="true"/> <!-- NOTE: Logically␣
    ↪Equivalent -->
159       <clock name="clk" num_pins="1"/>
160       <!-- Describe basic logic element -->
161       <pb_type name="ble" num_pb="10">
162         <input name="in" num_pins="6"/>
163         <output name="out" num_pins="1"/>
164         <clock name="clk" num_pins="1"/>
165         <pb_type name="soft_logic" num_pb="1">
166           <input name="in" num_pins="6"/>
167           <output name="out" num_pins="1"/>
168           <mode name="n1_lut6">
169             <pb_type name="lut6" blif_model=".names" num_pb="1" class="lut">
170               <input name="in" num_pins="6" port_class="lut_in"/>
171               <output name="out" num_pins="1" port_class="lut_out"/>
172               <!-- LUT timing using delay matrix -->
173               <delay_matrix type="max" in_port="lut6.in" out_port="lut6.out">
174                           2.690e-10
175                           2.690e-10
176                           2.690e-10
177                           2.690e-10
178                           2.690e-10
179                           2.690e-10
180               </delay_matrix>
181             </pb_type>
182             <interconnect>
183               <direct name="direct1" input="soft_logic.in[5:0]" output="lut6[0:0].
    ↪in[5:0]"/>
184               <direct name="direct2" input="lut6[0:0].out" output="soft_logic.out[0:0]
    ↪"/>
185             </interconnect>
186           </mode>
187         </pb_type>
188         <pb_type name="ff" blif_model=".latch" num_pb="1" class="flipflop">
189           <input name="D" num_pins="1" port_class="D"/>
190           <output name="Q" num_pins="1" port_class="Q"/>
191           <clock name="clk" num_pins="1" port_class="clock"/>
192           <T_setup value="2.448e-10" port="ff.D" clock="clk"/>
193           <T_clock_to_Q max="7.732e-11" port="ff.Q" clock="clk"/>
194         </pb_type>
195         <interconnect>
196           <!-- Two ff, make ff available to only corresponding luts -->
197           <direct name="direct1" input="ble.in" output="soft_logic.in"/>
198           <direct name="direct2" input="soft_logic.out" output="ff.D"/>
199           <direct name="direct4" input="ble.clk" output="ff.clk"/>
```

(continues on next page)

(continued from previous page)

```
          <mux name="mux1" input="ff.Q soft_logic.out" output="ble.out"/>
        </interconnect>
      </pb_type>
      <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>
      <fc in_type="frac" in_val="0.15" out_type="frac" out_val="0.10"/>
      <pinlocations pattern="spread"/>
      </pb_type>

    <!-- This is the 36*36 uniform mult -->
    <pb_type name="mult_36" height="4">
      <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"/>
          <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"/>
              <output name="OUT_cfg" num_pins="18"/>
              <pb_type name="mult_9x9" blif_model=".subckt multiply" num_pb="1">
                <input name="a" num_pins="9"/>
                <input name="b" num_pins="9"/>
                <output name="out" num_pins="18"/>
                <delay_constant max="1.667e-9" in_port="mult_9x9.a" out_port="mult_
↪9x9.out"/>
                <delay_constant max="1.667e-9" in_port="mult_9x9.b" out_port="mult_
↪9x9.out"/>
              </pb_type>
              <interconnect>
                <direct name="a2a" input="mult_9x9_slice.A_cfg" output="mult_9x9.a"/>
                <direct name="b2b" input="mult_9x9_slice.B_cfg" output="mult_9x9.b"/>
                <direct name="out2out" input="mult_9x9.out" output="mult_9x9_slice.
↪OUT_cfg"/>
              </interconnect>
            </pb_type>
            <interconnect>
              <direct name="a2a" input="divisible_mult_18x18.a" output="mult_9x9_
↪slice[1:0].A_cfg"/>
              <direct name="b2b" input="divisible_mult_18x18.b" output="mult_9x9_
↪slice[1:0].B_cfg"/>
              <direct name="out2out" input="mult_9x9_slice[1:0].OUT_cfg" output=
↪"divisible_mult_18x18.out"/>
            </interconnect>
          </mode>
          <mode name="mult_18x18">
```

(continues on next page)

(continued from previous page)

```
            <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 name="mult_18x18" blif_model=".subckt multiply" num_pb="1">
                <input name="a" num_pins="18"/>
                <input name="b" num_pins="18"/>
                <output name="out" num_pins="36"/>
                <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.b" 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>
            <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>
          </mode>
        </pb_type>
        <interconnect>
          <direct name="a2a" input="mult_36.a" output="divisible_mult_18x18[1:0].a"/>
          <direct name="b2b" input="mult_36.b" output="divisible_mult_18x18[1:0].b"/>
          <direct name="out2out" input="divisible_mult_18x18[1:0].out" output="mult_
↪36.out"/>
        </interconnect>
      </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_36x36.a" out_port="mult_
↪36x36.out"/>
            <delay_constant max="1.667e-9" in_port="mult_36x36.b" out_port="mult_
↪36x36.out"/>
          </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"/>
```

(continues on next page)

(continued from previous page)

```
          </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>
      </mode>
      <fc_in type="frac">0.15</fc_in>
      <fc_out type="frac">0.10</fc_out>
      <pinlocations pattern="spread"/>
      </pb_type>

    <!-- Memory based off Stratix IV 144K memory.  Setup time set to match flip-flop␣
↪setup time at 45 nm. Clock to q based off 144K max MHz  -->
    <pb_type name="memory" height="6">
      <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"/>
      <mode name="mem_2048x72_sp">
        <pb_type name="mem_2048x72_sp" blif_model=".subckt single_port_ram" class=
↪"memory" num_pb="1">
          <input name="addr" num_pins="11" port_class="address"/>
          <input name="data" num_pins="72" port_class="data_in"/>
          <input name="we" num_pins="1" port_class="write_en"/>
          <output name="out" num_pins="72" port_class="data_out"/>
          <clock name="clk" num_pins="1" port_class="clock"/>
          <T_setup value="2.448e-10" port="mem_2048x72_sp.addr" clock="clk"/>
          <T_setup value="2.448e-10" port="mem_2048x72_sp.data" clock="clk"/>
          <T_setup value="2.448e-10" port="mem_2048x72_sp.we" clock="clk"/>
          <T_clock_to_Q max="1.852e-9" port="mem_2048x72_sp.out" clock="clk"/>
        </pb_type>
        <interconnect>
          <direct name="address1" input="memory.addr1[10:0]" output="mem_2048x72_sp.
↪addr"/>
          <direct name="data1" input="memory.data[71:0]" output="mem_2048x72_sp.data"/
↪>
          <direct name="writeen1" input="memory.we1" output="mem_2048x72_sp.we"/>
          <direct name="dataout1" input="mem_2048x72_sp.out" output="memory.out[71:0]
↪"/>
          <direct name="clk" input="memory.clk" output="mem_2048x72_sp.clk"/>
        </interconnect>
      </mode>
      <mode name="mem_4096x36_dp">
        <pb_type name="mem_4096x36_dp" blif_model=".subckt dual_port_ram" class=
↪"memory" num_pb="1">
          <input name="addr1" num_pins="12" port_class="address1"/>
          <input name="addr2" num_pins="12" port_class="address2"/>
          <input name="data1" num_pins="36" port_class="data_in1"/>
          <input name="data2" num_pins="36" port_class="data_in2"/>
          <input name="we1" num_pins="1" port_class="write_en1"/>
          <input name="we2" num_pins="1" port_class="write_en2"/>
```

(continues on next page)

(continued from previous page)

```
          <output name="out1" num_pins="36" port_class="data_out1"/>
          <output name="out2" num_pins="36" port_class="data_out2"/>
          <clock name="clk" num_pins="1" port_class="clock"/>
          <T_setup value="2.448e-10" port="mem_4096x36_dp.addr1" clock="clk"/>
          <T_setup value="2.448e-10" port="mem_4096x36_dp.data1" clock="clk"/>
          <T_setup value="2.448e-10" port="mem_4096x36_dp.we1" clock="clk"/>
          <T_setup value="2.448e-10" port="mem_4096x36_dp.addr2" clock="clk"/>
          <T_setup value="2.448e-10" port="mem_4096x36_dp.data2" clock="clk"/>
          <T_setup value="2.448e-10" port="mem_4096x36_dp.we2" clock="clk"/>
          <T_clock_to_Q max="1.852e-9" port="mem_4096x36_dp.out1" clock="clk"/>
          <T_clock_to_Q max="1.852e-9" port="mem_4096x36_dp.out2" clock="clk"/>
        </pb_type>
        <interconnect>
          <direct name="address1" input="memory.addr1[11:0]" output="mem_4096x36_dp.
↪addr1"/>
          <direct name="address2" input="memory.addr2[11:0]" output="mem_4096x36_dp.
↪addr2"/>
          <direct name="data1" input="memory.data[35:0]" output="mem_4096x36_dp.data1
↪"/>
          <direct name="data2" input="memory.data[71:36]" output="mem_4096x36_dp.data2
↪"/>
          <direct name="writeen1" input="memory.we1" output="mem_4096x36_dp.we1"/>
          <direct name="writeen2" input="memory.we2" output="mem_4096x36_dp.we2"/>
          <direct name="dataout1" input="mem_4096x36_dp.out1" output="memory.out[35:0]
↪"/>
          <direct name="dataout2" input="mem_4096x36_dp.out2" output="memory.
↪out[71:36]"/>
          <direct name="clk" input="memory.clk" output="mem_4096x36_dp.clk"/>
        </interconnect>
      </mode>
      <mode name="mem_4096x36_sp">
        <pb_type name="mem_4096x36_sp" blif_model=".subckt single_port_ram" class=
↪"memory" num_pb="1">
          <input name="addr" num_pins="12" port_class="address"/>
          <input name="data" num_pins="36" port_class="data_in"/>
          <input name="we" num_pins="1" port_class="write_en"/>
          <output name="out" num_pins="36" port_class="data_out"/>
          <clock name="clk" num_pins="1" port_class="clock"/>
          <T_setup value="2.448e-10" port="mem_4096x36_sp.addr" clock="clk"/>
          <T_setup value="2.448e-10" port="mem_4096x36_sp.data" clock="clk"/>
          <T_setup value="2.448e-10" port="mem_4096x36_sp.we" clock="clk"/>
          <T_clock_to_Q max="1.852e-9" port="mem_4096x36_sp.out" clock="clk"/>
        </pb_type>
        <interconnect>
          <direct name="address1" input="memory.addr1[11:0]" output="mem_4096x36_sp.
↪addr"/>
          <direct name="data1" input="memory.data[35:0]" output="mem_4096x36_sp.data"/
↪>
          <direct name="writeen1" input="memory.we1" output="mem_4096x36_sp.we"/>
          <direct name="dataout1" input="mem_4096x36_sp.out" output="memory.out[35:0]
↪"/>
          <direct name="clk" input="memory.clk" output="mem_4096x36_sp.clk"/>
        </interconnect>
      </mode>
      <mode name="mem_9182x18_dp">
        <pb_type name="mem_9182x18_dp" blif_model=".subckt dual_port_ram" class=
↪"memory" num_pb="1">
```

(continues on next page)

(continued from previous page)

```
392           <input name="addr1" num_pins="13" port_class="address1"/>
393           <input name="addr2" num_pins="13" port_class="address2"/>
394           <input name="data1" num_pins="18" port_class="data_in1"/>
395           <input name="data2" num_pins="18" port_class="data_in2"/>
396           <input name="we1" num_pins="1" port_class="write_en1"/>
397           <input name="we2" num_pins="1" port_class="write_en2"/>
398           <output name="out1" num_pins="18" port_class="data_out1"/>
399           <output name="out2" num_pins="18" port_class="data_out2"/>
400           <clock name="clk" num_pins="1" port_class="clock"/>
401           <T_setup value="2.448e-10" port="mem_9182x18_dp.addr1" clock="clk"/>
402           <T_setup value="2.448e-10" port="mem_9182x18_dp.data1" clock="clk"/>
403           <T_setup value="2.448e-10" port="mem_9182x18_dp.we1" clock="clk"/>
404           <T_setup value="2.448e-10" port="mem_9182x18_dp.addr2" clock="clk"/>
405           <T_setup value="2.448e-10" port="mem_9182x18_dp.data2" clock="clk"/>
406           <T_setup value="2.448e-10" port="mem_9182x18_dp.we2" clock="clk"/>
407           <T_clock_to_Q max="1.852e-9" port="mem_9182x18_dp.out1" clock="clk"/>
408           <T_clock_to_Q max="1.852e-9" port="mem_9182x18_dp.out2" clock="clk"/>
409         </pb_type>
410         <interconnect>
411           <direct name="address1" input="memory.addr1[12:0]" output="mem_9182x18_dp.
    ↪addr1"/>
412           <direct name="address2" input="memory.addr2[12:0]" output="mem_9182x18_dp.
    ↪addr2"/>
413           <direct name="data1" input="memory.data[17:0]" output="mem_9182x18_dp.data1
    ↪"/>
414           <direct name="data2" input="memory.data[35:18]" output="mem_9182x18_dp.data2
    ↪"/>
415           <direct name="writeen1" input="memory.we1" output="mem_9182x18_dp.we1"/>
416           <direct name="writeen2" input="memory.we2" output="mem_9182x18_dp.we2"/>
417           <direct name="dataout1" input="mem_9182x18_dp.out1" output="memory.out[17:0]
    ↪"/>
418           <direct name="dataout2" input="mem_9182x18_dp.out2" output="memory.
    ↪out[35:18]"/>
419           <direct name="clk" input="memory.clk" output="mem_9182x18_dp.clk"/>
420         </interconnect>
421       </mode>
422       <mode name="mem_9182x18_sp">
423         <pb_type name="mem_9182x18_sp" blif_model=".subckt single_port_ram" class=
    ↪"memory" num_pb="1">
424           <input name="addr" num_pins="13" port_class="address"/>
425           <input name="data" num_pins="18" port_class="data_in"/>
426           <input name="we" num_pins="1" port_class="write_en"/>
427           <output name="out" num_pins="18" port_class="data_out"/>
428           <clock name="clk" num_pins="1" port_class="clock"/>
429           <T_setup value="2.448e-10" port="mem_9182x18_sp.addr" clock="clk"/>
430           <T_setup value="2.448e-10" port="mem_9182x18_sp.data" clock="clk"/>
431           <T_setup value="2.448e-10" port="mem_9182x18_sp.we" clock="clk"/>
432           <T_clock_to_Q max="1.852e-9" port="mem_9182x18_sp.out" clock="clk"/>
433         </pb_type>
434         <interconnect>
435           <direct name="address1" input="memory.addr1[12:0]" output="mem_9182x18_sp.
    ↪addr"/>
436           <direct name="data1" input="memory.data[17:0]" output="mem_9182x18_sp.data"/
    ↪>
437           <direct name="writeen1" input="memory.we1" output="mem_9182x18_sp.we"/>
438           <direct name="dataout1" input="mem_9182x18_sp.out" output="memory.out[17:0]
    ↪"/>
```

(continues on next page)

(continued from previous page)

```
          <direct name="clk" input="memory.clk" output="mem_9182x18_sp.clk"/>
        </interconnect>
      </mode>
      <mode name="mem_18194x9_dp">
        <pb_type name="mem_18194x9_dp" blif_model=".subckt dual_port_ram" class=
↪"memory" num_pb="1">
          <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"/>
          <T_setup value="2.448e-10" port="mem_18194x9_dp.addr1" clock="clk"/>
          <T_setup value="2.448e-10" port="mem_18194x9_dp.data1" clock="clk"/>
          <T_setup value="2.448e-10" port="mem_18194x9_dp.we1" clock="clk"/>
          <T_setup value="2.448e-10" port="mem_18194x9_dp.addr2" clock="clk"/>
          <T_setup value="2.448e-10" port="mem_18194x9_dp.data2" clock="clk"/>
          <T_setup value="2.448e-10" port="mem_18194x9_dp.we2" clock="clk"/>
          <T_clock_to_Q max="1.852e-9" port="mem_18194x9_dp.out1" clock="clk"/>
          <T_clock_to_Q max="1.852e-9" port="mem_18194x9_dp.out2" clock="clk"/>
        </pb_type>
        <interconnect>
          <direct name="address1" input="memory.addr1[13:0]" output="mem_18194x9_dp.
↪addr1"/>
          <direct name="address2" input="memory.addr2[13:0]" output="mem_18194x9_dp.
↪addr2"/>
          <direct name="data1" input="memory.data[8:0]" output="mem_18194x9_dp.data1"/
↪>
          <direct name="data2" input="memory.data[17:9]" output="mem_18194x9_dp.data2
↪"/>
          <direct name="writeen1" input="memory.we1" output="mem_18194x9_dp.we1"/>
          <direct name="writeen2" input="memory.we2" output="mem_18194x9_dp.we2"/>
          <direct name="dataout1" input="mem_18194x9_dp.out1" output="memory.out[8:0]
↪"/>
          <direct name="dataout2" input="mem_18194x9_dp.out2" output="memory.out[17:9]
↪"/>
          <direct name="clk" input="memory.clk" output="mem_18194x9_dp.clk"/>
        </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>
          <direct name="address1" input="memory.addr1[13:0]" output="mem_18194x9_sp.
↪addr"/>
```

(continues on next page)

(continued from previous page)

```
488             <direct name="data1" input="memory.data[8:0]" output="mem_18194x9_sp.data"/>
489             <direct name="writeen1" input="memory.we1" output="mem_18194x9_sp.we"/>
490             <direct name="dataout1" input="mem_18194x9_sp.out" output="memory.out[8:0]"/
    ↪>
491             <direct name="clk" input="memory.clk" output="mem_18194x9_sp.clk"/>
492           </interconnect>
493         </mode>
494         <fc in_type="frac" in_val="0.15" out_type="frac" out_val="0.10"/>
495         <pinlocations pattern="spread"/>
496         </pb_type>
497   </complexblocklist>
498 </architecture>
```

CHAPTER 3

VPR

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 (eg. critical path delay, area, etc).

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.

3.1 Command-line Options

3.1.1 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.

3.1.2 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.

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

--place_file `<file>`
Path to final placement file

--route_file `<file>`
Path to final routing file

--sdc_file `<file>`
Path to SDC timing constraints file

--outfile_prefix `<string>`
Prefix for output files

General Options

VPR runs all three stages of pack, place, and route if none of `--pack`, `--place`, or `--route` are specified.

`--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`

`--pack`
Run packing stage

Default: off

`--place`
Run placement stage

Default: off

`--route`
Run routing stage This also implies –analysis.

Default: off

`--analysis`
Run final analysis stage (e.g. timing, power).

Default: off

`--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`

`--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`

`--slack_definition` `{ R | I | S | G | C | N }`
The slack definition used in the classic timing analyzer. This option is for experimentation only; the default is fine for ordinary usage. See path_delay.c for details.

Default: `R`

`--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`

--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_method {ideal | route}
Specifies how clock nets are handled:

- `ideal`: Treat clock pins as ideal (i.e. clock nets are not routed)
- `route`: Treat clock nets as normal nets (i.e. routed using inter-block routing)

Default: `ideal`

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`

--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

--verbose_sweep {on | off}
Controls whether sweeping describes the netlist modifications performed (i.e. what was swept).

See also:

--sweep_dangling_primary_ios

Default: off

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, mulitpliers, 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}
Controls whether or not primitives with no attraction to the current cluster can be packed into it.

Default: on

--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.

Default: blend 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

--target_ext_pin_util { <float> | <float>,<float> | <string>:<float> | <string>:<float>,<fl
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:

- <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, nad 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: `1.0`

`--debug_clustering` `{on | off}`
Controls verbose clustering output. Useful for debugging architecture packing problems.

Default: `off`

Placer Options

The placement engine in VPR places logic blocks using simulated annealing. By default, the automatic annealing schedule is used *[BRM99][BR97b]*. 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 temperature 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 1` will speed up the placer by a factor of 10 while typically reducing placement quality only by 10% or less (depends on the architecture). Hence users more concerned with CPU time than quality may find this a more appropriate value of inner_num.

Default: `10.0`

--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 {random | <file.pads>}
Do not allow the placer to move the I/O locations about during the anneal. Instead, lock each I/O pad to some location at the start of the anneal. If -fix_pins random is specified, each I/O block is locked to a random pad location to model the effect of poor board-level I/O constraints. If any word other than random is specified after -fix_pins, that string is taken to be the name of a file listing the desired location of each I/O block in the netlist (i.e. -fix_pins <file.pads>). This pad location file is in the same format as a normal placement file, but only specifies the locations of I/O pads, rather than the locations of all blocks.

Default: off (i.e. placer chooses pad locations).

--place_algorithm {bounding_box | path_timing_driven}
Controls the algorithm used by the placer.

bounding_box focuses purely on minimizing the bounding box wirelength of the circuit.

path_timing_driven focuses on minimizing both wirelength and the critical path delay.

Default: path_timing_driven

--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

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`

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`

--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`

--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`

--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}
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, but normalizes all these basic costs to be of the same magnitude as the typical delay through a routing resource.

Default: delay_normalized 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 with a fixed channel width.

Note: 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.

--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 ocassionally 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 ocassionally 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.

The breadth_first router focuses solely on routing a design successfully, while the timing_driven router 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: 64

--write_rr_graph <file>
Writes out the routing resource graph generated at the last stage of VPR into 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> in the VTR root directory and loads it into the placement and routing stage of VPR.

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.

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`

--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 }
Controls how the routing budgets are created. Routing budgets are used to guid 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 *[YLS92][FBC08]*.

`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`

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`

`--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 }`
Controls the level of detail included in generated timing reports.

- `netlist`: Timing reports show only netlist primitive pins.

  For example:

```
#Path 150
Startpoint: top^cur_state~3_FF_NODE.Q[0] (.latch clocked by top^clk)
Endpoint  : top^finish_FF_NODE.D[0] (.latch clocked by top^clk)
Path Type : setup

Point                                                                  Incr␣
↪     Path
------------------------------------------------------------------------------
↪---------
clock top^clk (rise edge)                                              0.000␣
↪     0.000
clock source latency                                                   0.000␣
↪     0.000
top^clk.inpad[0] (.input)                                              0.000␣
↪     0.000
top^cur_state~3_FF_NODE.clk[0] (.latch)                                0.042␣
↪     0.042
top^cur_state~3_FF_NODE.Q[0] (.latch) [clock-to-output]                0.124␣
↪     0.166
n1168.in[4] (.names)                                                   0.475␣
↪     0.641
n1168.out[0] (.names)                                                  0.261␣
↪     0.902
top^finish_FF_NODE.D[0] (.latch)                                       0.000␣
↪     0.902
data arrival time                                                        ␣
↪     0.902

clock top^clk (rise edge)                                              0.000␣
↪     0.000
clock source latency                                                   0.000␣
↪     0.000
top^clk.inpad[0] (.input)                                              0.000␣
↪     0.000
```

(continues on next page)

(continued from previous page)

```
top^finish_FF_NODE.clk[0] (.latch)                                    0.042␣
↪    0.042
clock uncertainty                                                     0.000␣
↪    0.042
cell setup time                                                      -0.066␣
↪    -0.024
data required time                                                        ␣
↪    -0.024
------------------------------------------------------------------------------
↪---------
data required time                                                        ␣
↪    -0.024
data arrival time                                                         ␣
↪    -0.902
------------------------------------------------------------------------------
↪---------
slack (VIOLATED)                                                          ␣
↪    -0.926
```

- `aggregated`: Timing reports show netlist pins, and an aggregated summary of intra-block and inter-block routing delays.

  For example:

```
#Path 150
Startpoint: top^cur_state~3_FF_NODE.Q[0] (.latch clocked by top^clk)
Endpoint  : top^finish_FF_NODE.D[0] (.latch clocked by top^clk)
Path Type : setup

Point                                                                 Incr␣
↪     Path
------------------------------------------------------------------------------
↪---------
clock top^clk (rise edge)                                             0.000␣
↪    0.000
clock source latency                                                  0.000␣
↪    0.000
top^clk.inpad[0] (.input)                                             0.000␣
↪    0.000
| (intra 'io' routing)                                               0.042␣
↪    0.042
| (inter-block routing)                                              0.000␣
↪    0.042
| (intra 'clb' routing)                                              0.000␣
↪    0.042
top^cur_state~3_FF_NODE.clk[0] (.latch)                               0.000␣
↪    0.042
| (primitive '.latch' Tcq_max)                                       0.124␣
↪    0.166
top^cur_state~3_FF_NODE.Q[0] (.latch) [clock-to-output]               0.000␣
↪    0.166
| (intra 'clb' routing)                                              0.045␣
↪    0.211
| (inter-block routing)                                              0.335␣
↪    0.546
| (intra 'clb' routing)                                              0.095␣
↪    0.641
```

(continues on next page)

(continued from previous page)

```
n1168.in[4] (.names)                                                    0.000␣
↪    0.641
| (primitive '.names' combinational delay)                             0.261␣
↪    0.902
n1168.out[0] (.names)                                                   0.000␣
↪    0.902
| (intra 'clb' routing)                                                0.000␣
↪    0.902
top^finish_FF_NODE.D[0] (.latch)                                        0.000␣
↪    0.902
data arrival time                                                          ␣
↪    0.902

clock top^clk (rise edge)                                               0.000␣
↪    0.000
clock source latency                                                    0.000␣
↪    0.000
top^clk.inpad[0] (.input)                                               0.000␣
↪    0.000
| (intra 'io' routing)                                                 0.042␣
↪    0.042
| (inter-block routing)                                                0.000␣
↪    0.042
| (intra 'clb' routing)                                                0.000␣
↪    0.042
top^finish_FF_NODE.clk[0] (.latch)                                      0.000␣
↪    0.042
clock uncertainty                                                       0.000␣
↪    0.042
cell setup time                                                        -0.066␣
↪   -0.024
data required time                                                         ␣
↪   -0.024
-------------------------------------------------------------------------------
↪---------
data required time                                                         ␣
↪   -0.024
data arrival time                                                          ␣
↪   -0.902
-------------------------------------------------------------------------------
↪---------
slack (VIOLATED)                                                           ␣
↪   -0.926
```

where each line prefixed with `|` (pipe character) represent a sub-delay of an edge within the timing graph.

For instance:

```
top^cur_state~3_FF_NODE.Q[0] (.latch) [clock-to-output]                 0.000␣
↪    0.166
| (intra 'clb' routing)                                                0.045␣
↪    0.211
| (inter-block routing)                                                0.335␣
↪    0.546
| (intra 'clb' routing)                                                0.095␣
↪    0.641
```

(continues on next page)

(continued from previous page)

```
n1168.in[4] (.names)                                                    0.000␣
↪     0.641
```

indicates that between the netlist pins `top^cur_state~3_FF_NODE.Q[0]` and `n1168.in[4]` there are delays of:

- `45` ps from the `.latch` output pin to an output pin of a `clb` block,
- `335` ps through the general inter-block routing fabric, and
- `95` ps from the input pin of a `clb` block to the `.names` input.

Similarly, we can observe that the connection between `n1168.out[0]` and `top^finish_FF_NODE.D[0]` is contained entirely within the same `clb` block, and does not use the general inter-block routing network:

```
n1168.out[0] (.names)                                                   0.000␣
↪     0.902
| (intra 'clb' routing)                                                 0.000␣
↪     0.902
top^finish_FF_NODE.D[0] (.latch)                                        0.000␣
↪     0.902
```

Default: `netlist`

`--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 activites 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*.

3.2 Graphics

VPR includes easy-to-use graphics for visualizing both the targetted FPGA architecture, and the circuit VPR has implementation on the architecture.

3.2.1 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:

```
-- EasyGL: graphics enabled
```

If the required libraries are not found cmake will report:

```
-- EasyGL: graphics disabled
```

and list the missing libraries:

```
-- EasyGL: Failed to find required X11 library (on debian/ubuntu try 'sudo apt-get␣
↪install libx11-dev' to install)
-- EasyGL: Failed to find required Xft library (on debian/ubuntu try 'sudo apt-get␣
↪install libxft-dev' to install)
-- EasyGL: Failed to find required fontconfig library (on debian/ubuntu try 'sudo apt-
↪get install fontconfig' to install)
-- EasyGL: 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.

A graphical window will now pop up when you run VPR.

3.2.2 Navigation

Click any mouse button on the **arrow** keys to pan the view, or click on the **Zoom-In**, **Zoom-Out** and **Zoom-Fit** buttons to zoom the view. Alternatively, click and drag the mouse wheel to pan the view, or scroll the mouse wheel to zoom in and out. Click on the **Window button**, then on the diagonally opposite corners of a box, to zoom in on a particular area.

Selecting **PostScript** creates a PostScript file (in pic1.ps, pic2.ps, etc.) of the image on screen.

Proceed tells VPR to continue with the next step in placing and routing the circuit. **Exit** aborts the program.

Note: Menu buttons will be greyed out when they are not selectable (e.g. VPR is working).

3.2.3 Visualizing Netlist Connectivity

The **Toggle Nets** button 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.

Again, you can click on Toggle Nets to make net routings visible or invisible. 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.

3.2.4 Visualizing Routing Architecture

When a routing is on-screen, clicking on **Toggle RR** will switch between various views of the routing resources available in the FPGA.

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. 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.

3.2.5 Visualizing Routing Congestion

When a routing is shown on-screen, clicking on the **Congestion** button 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.

3.2.6 Visualizing the Critical Path

During placement and routing you can click on the **Crit. Path** button 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.

3.3 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*.

3.3.1 Default Timing Constraints

If no timing constriants 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:

```
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:

```
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:

```
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 similarily extended if there are more than two netlist clocks.

3.4 SDC Commands

The following subset of SDC syntax is supported by VPR:

3.4.1 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:

```
#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.

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.

3.4.2 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:

```
#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

```
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

3.4.3 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:

```
#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.

set_false_path

-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

3.4.4 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:

```
#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.

set_max_delay/set_min_delay

<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

3.4.5 set_multicycle_path

Sets how many clock cycles elapse between the launch and capture edges for setup and hold checks.

The default the setup mutlicycle 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 3 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
```

Note: Multicycles are supported between entire clock domains, but *not* between individual 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

<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`.

3.4.6 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}]

#Set a minimum output delay of 1.0 (relative to output_clk) on
#all ports matching starting with 'out*'
set_output_delay -clock output_clk -min 1 [get_ports {out*}]

#Set both the maximum and minimum output delay to 0.3 for all I/Os
#in the design
set_output_delay -clock clk2 0.3 [get_ports {*}]
```

`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.

3.4.7 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.

3.4.8 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.

3.4.9 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 'in[0]' and 'out[0]' on
#the netlist primitive named 'blk1'
set_disable_timing -from [get_pins {blk1.in[0]}] -to [get_pins {blk1.out[0]}]
```

set_disable_timing

-from [get_pins <pin list or regexes>]
Specifies the source netlist pins.

Required: Yes

-to [get_pins <pin list or regexes>]
Specifies the sink netlist pins.

Required: Yes

3.4.10 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.

3.4.11 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 {*}]
```

E

Sample using many supported SDC commands. Inputs and outputs are constrained on separate virtual clocks.

```
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*}]
```

3.5 File Formats

VPR consumes and produces several files representing the packing, placement, and routing results.

3.5.1 FPGA Architecture (.xml)

The target FPGA architecture is specified as an architecture file. For details of this file format see *FPGA Architecture Description*.

3.5.2 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 `unconn` net (input pins only).

   VPR treats any **input pin** connected to a net named `unconn` as disconnected.

   For example:

   ```
   .names unconn 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[11]=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 \
   ```

(continues on next page)

(continued from previous page)

```
    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-cary 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 it's `cout` output left implicitly disconnected.

```
.model top
.inputs clk a[0] a[1] a[2] a[3] b[0] b[1] b[2] b[3]
.outputs sum[0] sum[1] sum[2] sum[3] cout all_sum_high_reg

.names gnd
 0

.subckt adder a=a[0] b=b[0] cin=gnd    cout=cin[1]     sumout=sum[0]
.subckt adder a=a[1] b=b[1] cin=cin[1] cout=cin[2]     sumout=sum[1]
.subckt adder a=a[2] b=b[2] cin=cin[2] cout=cin[3]     sumout=sum[2]
.subckt adder a=a[3] b=b[3] cin=cin[3]                 sumout=sum[3]
```

(continues on next page)

(continued from previous page)

```
.names sum[0] sum[1] sum[2] sum[3] all_sum_high_comb
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
```

3.5.3 Extended BLIF (.eblif)

VPR also supports several extentions to *structural BLIF* to address some of its limitations.

Note: By default VPR assumes file with `.eblif` are in extneded 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
```

(continues on next page)

(continued from previous page)

```
.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 my_dff
.param test_latch_param "test_latch_param_value"
.attr test_latch_attrib "test_latch_param_attrib"

.conn dff_q o_dff

.end
```

3.5.4 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*.

3.5.5 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 phyiscal 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 primitve (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 3.1: Example of `<port_rotation_map>` tag.

```
1  ...
2  <block name="c" instance="lut[0]">
3      <inputs>
4          <port name="in">open open lut5.in[2]->direct:lut5  open lut5.in[4]->
   ↪direct:lut5  </port>
5          <port_rotation_map name="in">open open 1 open 0 </port_rotation_map>
6      </inputs>
7      <outputs>
8          <port name="out">c </port>
9      </outputs>
10     <clocks>
11     </clocks>
12 </block>
13 ...
```

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 3.2: Example packed netlist file (trimmed for brevity).

```
1  <block name="b1.net" instance="FPGA_packed_netlist[0]">
2      <inputs>
3              pa pb pc
4      </inputs>
5
6      <outputs>
7              out:pd out:pe out:pf out:pg
```

(continues on next page)

(continued from previous page)

```
    </outputs>

    <clocks>
    </clocks>

    <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>

            <clocks>
                    <port name="clock">open </port>
            </clocks>

            <block name="pa" instance="inpad[0]">
                    <inputs>
                    </inputs>

                    <outputs>
                            <port name="inpad">pa </port>
                    </outputs>

                    <clocks>
                    </clocks>

                    <attributes>
                            <attribute name="vccio">3.3</attribute>
                    </attributes>

                    <parameters>
                            <parameter name="iostandard">LVCMOS33</parameter>
                    </parameters>
            </block>
    </block>
...
```

3.5.6 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:

```
block_name    x        y    subblock_number
```

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.

The `subblock number` is meaningful only for I/O pads. Since we can have more than one pad in a row or column when io_rat is set to be greater than 1 in the architecture file, the subblock number specifies which of the several possible pad locations in row x and column y contains this pad. Note that the first pads occupied at some (x, y) location are always those with the lowest subblock numbers – i.e. if only one pad at (x, y) is used, the subblock number of the I/O placed there will be zero. For CLBs, the subblock number is always zero.

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.

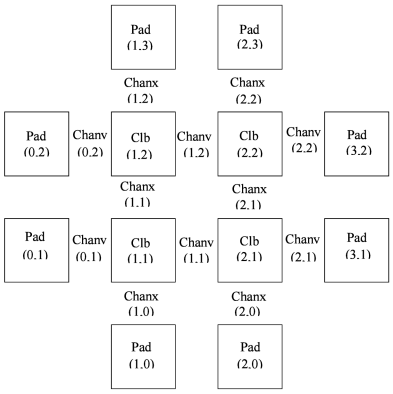


Fig. 3.1: FPGA co-ordinate system.

Fig. 3.1 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 3.3: 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
```

3.5.7 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. 3.1 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 3.4: 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.O[12]  Switch:0   #Output pin the O 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 3.5: 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
```

3.5.8 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 catagory 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>``content``</channel>
- **<switches>**
  - <switch>``content``</switch>
- **<segments>**
  - ``content``
- **<block_types>**
  - <block_type>``content``</block_type>
- **<grid>**
  - <grid_loc>``content``</grid_loc>
- **<rr_nodes>**

- <node>``content``</node>

- **<rr_edges>**
  - <edge>``content``</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:

<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.

<x_list index="int" info="int"/> <y_list index="int" info="int"/>

These are a required subtags that lists 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.

<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*.

<timing R="float" cin="float" Cout="float" Tdel="float/>

This optional subtag contains information used for timing analysis. Without it, the program assums 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.

****

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.

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:

<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 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.

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:

<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.

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.

<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 representd 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.

<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.

<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.
- **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.

****

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.

Routing Resource Graph Format Example

An example of what a generated routing resource graph file would look like is shown below:

Listing 3.6: 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"/>
        <x_list index="2" info="5"/>
        <y_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"/>
            <timing R_per_meter="201.7" C_per_meter="18.110e-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"/>
```

(continues on next page)

(continued from previous page)

```
            <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>
```

3.6 Debugging Aids

Note: This section is only relevant to developers modifying VPR

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. The `critical_path.echo` 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.

If the preprocessor flag `DEBUG` is defined in `vpr_types.h`, some additional sanity checks are performed during a run. `DEBUG` only slows execution by 1 to 2%. The major sanity checks are always enabled, regardless of the state of `DEBUG`. Finally, if `VERBOSE` is set in 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 `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 `dump_rr_graph` subroutine.

CHAPTER 4

Odin II

Odin II is used for logic synthesis and elaboration, converting a subset of the Verilog Hardware Description Language (HDL) into a BLIF netlist.

See also:

[JKGS10]

4.1 INSTALL

4.1.1 Prerequisites

1. ctags
2. bison
3. flex
4. gcc 5.x
5. cmake 2.8.12 (minimum version)
6. time
7. cairo

4.1.2 Build

To build ODIN, run "make odin_II" from the vtr root directory.

Note: ODIN uses CMake as it's build system. CMake provides a protable 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 the included verify_microbenchmarks.sh script. This will automatically compile, simulate, and verify all of the included microbenchmark circuits to ensure that Odin is working correctly on your system.

4.2 USAGE

./odin_II [args]

4.2.1 Required [args]

-c	<XML Configuration File>	fpga_architecture_file.xml format is specified from VPR
-V	<Verilog HDL File>	You may specify multiple verilog HDL files for synthesis
-b	<BLIF File>	

4.2.2 Optional [args]

-o	<output file>	full output path and file name for the blif output file
-a	<architecture file>	an FPGA architecture file in VPR format to map to
-G		Output netlist graph in GraphViz viewable .dot format. (net.dot, opens with dotty)
-A		Output AST graph in in GraphViz viewable .dot format.
-W		Print all warnings. (Can be substantial.)
-h		Print help

4.2.3 Simulation

Note: Simulation always produces files:

- input_vectors
- output_vectors
- test.do (ModelSim)

Activate Simulation with [args]

-g	<Number of random test vectors>	will simulate the generated netlist with the entered number of clock cycles using pseudo-random test vectors. These vectors and the resulting output vectors are written to "input_vectors" and "output_vectors" respectively. You can supply a predefined input vector using -t
-L	<Comma-separated list>	Comma-separated list of primary inputs to hold high at cycle 0, and low for all subsequent cycles.
-3		Generate three valued logic. (Default is binary.)
-t	<input vector file>	Supply a predefined input vector file
-U0		initial register value to 0
-U1		initial register value to 1
-UX		initial register value to X(unknown) (DEFAULT)

Simulation Optional [args]

-T	<output vector file>	The output vectors is verified against the supplied predefined output vector file
-E		Output after both edges of the clock. (Default is to output only after the falling edge.)
-R		Output after rising edge of the clock only. (Default is to output only after the falling edge.)
-p	<Comma-separated list>	Comma-separated list of additional pins/nodes to monitor during simulation. (view NOTES)

4.2.4 NOTES

Example for -p:

-p input~0,input~1	monitors pin 0 and 1 of input
-p input	monitors all pins of input as a single port
-p input~	monitors all pins of input as separate ports. (split)

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)

Examples .xml configuration file for -c

```
<config>
        <verilog_files>
                <!-- Way of specifying multiple files in a project! -->
                <verilog_file>verilog_file.v</verilog_file>
        </verilog_files>
        <output>
```

(continues on next page)

(continued from previous page)

```
                <!-- 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>
                <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 file
intput_1 input_2 output_1 output_2 output_3
# Comment
0 0XA 1 0XD 1101
```

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.

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.

Verilog HDL file Keyword Support:

Supported Keyword	NOT Sup. Keyword	Supported Operators	NOT Sup. Operators
always	automatic	**	&&&
and	buf	&&	=+:
assign	casex	||	-:
begin	casez	<=	>>>
case	disable	=>	(*
default	edge	>=	*)
‘define	endtask	<<	
defparam	macromodule	<<<	
else	scalared	>>	
end	specparam	==	
endcase	bufif0	!=	
endfunction	bufif1	===	
endmodule	cmos	!==	
endspecify	deassign	^~	

Continued on next page

Table 4.1 – continued from previous page

for	endprimitive	~^	
if	endtable	~&	
initial	event	~|	
inout	force		
input	forever		
integer	fork		
module	highz0		
function	highz1		
nand	join		
negedge	large		
nor	medium		
not	nmos		
or	notif0		
output	notif1		
parameter	pmos		
localparam	primitive		
posedge	pull0		

Continued on next page

Table 4.1 – continued from previous page

reg	pull1		
specify	pulldown		
while	pullup		
wire	rcmos		
xnor	release		
xor	repeat		
@()	rnmos		
@*	rpmos		
	rtran		
	rtranif0		
	rtranif1		
	small		
	signed		
	strong0		
	strong1		
	supply0		
	supply1		

Continued on next page

Table 4.1 – continued from previous page

	table		
	task		
	time		
	tran		
	tranif0		
	tranif1		
	tri		
	tri0		
	tri1		
	triand		
	trior		
	vectored		
	wait		
	wand		
	weak0		
	weak1		
	wor		

4.3 DOCUMENTING ODIN II

Any new command line options added to Odin II should be fully documented by the print_usage() function within odin_ii.c before checking in the changes.

4.4 TESTING ODIN II

The verify_microbenchmarks.sh and verify_regression_tests.sh scripts compile and 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 verify_microbenchmarks.sh to ensure that everything is working correctly on your system. Unlike the verify_regression_tests.sh script, verify_microbenchmarks.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. 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 these test sets. When new features are added to Odin II, new microbenchmarks should be developed which test those features for regression. Use existing circuits as a template for the addition of new circuits.

4.5 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.

Simulate the circuit in Odin II using the -E option to ensure that Odin II outputs both edges of the clock. 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.

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_input. Finally, move the output_vectors file to the test set folder and rename it test_circuit_output.

4.6 CONTACT

jamieson dot peter at gmail dot com ken at unb dot ca - We will service all requests as timely as possible, but please explain the problem with enough detail to help.

CHAPTER 5

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.

CHAPTER 6

Tutorials

6.1 Design Flow Tutorials

These tutorials describe how to run the VTR design flow.

6.1.1 Basic Design Flow Tutorial

The following steps show you 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` directory, and run:

   ```
   ../scripts/run_vtr_task.pl basic_flow
   ```

   This command will run the VTR flow on a single circuit and a single architecture. The files generated from the run are stored in `basic_flow/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_flow/run001. When the script completes, enter the following command:

   ```
   ../scripts/parse_vtr_task.pl basic_flow/
   ```

   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_flow comes with golden results that you can use to check for correctness. To do this check, enter the following command:

   ```
   ../scripts/parse_vtr_task.pl -check_golden basic_flow
   ```

   It should return: `basic_flow...[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 circuits, architecture, and CAD flow was employed by the run, look at `vtr_flow/tasks/basic_flow/config/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_release/vtr_flow/scripts/run_vtr_flow.pl` 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 pre-pended with "regression". These are regression tests included with the flow that test various combinations of flows, architectures, and benchmarks.

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*.

6.2 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:

6.2.1 Classic Soft Logic Block Tutorial

The following is an example on how to use the VPR architecture description langauge 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.

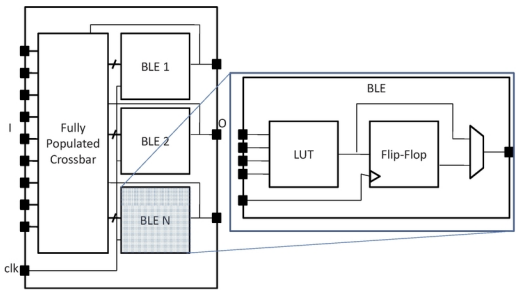


Fig. 6.1: Model of a classic FPGA soft logic cluster

Fig. 6.1 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:

```
<pb_type name="clb">
  <input name="I" num_pins="22" equivalent="true"/>
  <output name="O" num_pins="10" equivalent="true"/>
  <clock name="clk" equivalent="false"/>
```

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:

```
<pb_type name="ble" num_pb="10">
  <input name="in" num_pins="4"/>
  <output name="out" num_pins="1"/>
  <clock name="clk"/>
```

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:

```
<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. 6.2 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:

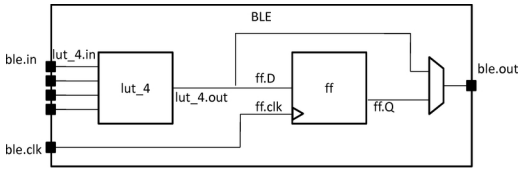


Fig. 6.2: Internal BLE names

```
  <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>
</pb_type>
```

The CLB interconnect is then modeled (see Fig. 6.1). 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. The columns occupied by complex blocks of type CLB is defined by gridlocations where fill means that all columns should be type CLB unless that column is taken up by a block with higher priority (where a larger number means a higher priority).

```
  <!-- 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"/>
  <gridlocations>
    <loc type="fill" priority="1"/>
  </gridlocations>
</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="true"/>
  <output name="O" num_pins="10" equivalent="true"/>
  <clock name="clk" equivalent="false"/>
```

(continues on next page)

(continued from previous page)

```
  <pb_type name="ble" num_pb="10">
    <input name="in" num_pins="4"/>
    <output name="out" num_pins="1"/>
    <clock name="clk"/>

    <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>
  </pb_type>

  <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>

  <!-- 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"/>
  <gridlocations>
    <loc type="fill" priority="1"/>
  </gridlocations>
</pb_type>
```

6.2.2 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.

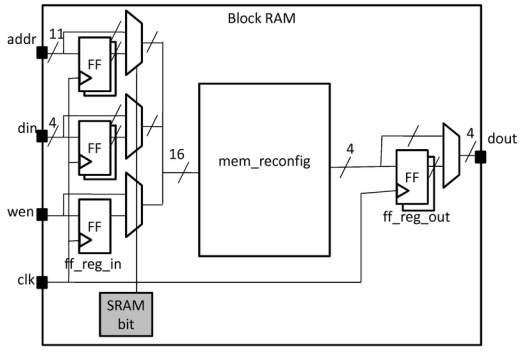


Fig. 6.3: Model of a configurable memory block

Fig. 6.3 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:

```
<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"/>
```

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 `ff_reg_in`, must be declared. There are 4 output bits that can also be registered, so 4 flip-flops (named `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 `class` attribute:

```
<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"/>
```

(continues on next page)

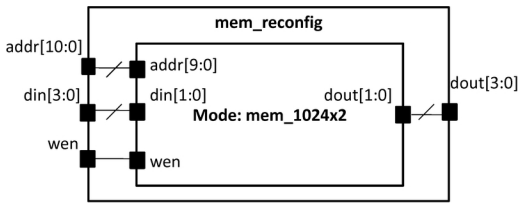


Fig. 6.4: Different modes of operation for the memory block.

(continued from previous page)

```
  <output name="dout" num_pins="4"/>

  <!-- Declare a 512x4 memory type -->
  <mode name="mem_512x4_mode">
    <!-- Follows the same pattern as the 1024x2 memory type declared below -->
  </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 2048x1 memory type -->
  <mode name="mem_2048x1_mode">
    <!-- Follows the same pattern as the 1024x2 memory type declared above -->
  </mode>

</pb_type>
```

The top-level interconnect structure of the memory SPCB is shown in Fig. 6.5. 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:

```
1  <interconnect>
2    <direct input="{block_RAM.wen block_RAM.din block_RAM.addr}" output="ff_reg_
   ↪in[15:0].D"/>
3    <direct input="mem_reconfig.dout" output="ff_reg_out[3:0].D"/>
4    <mux input="mem_reconfig.dout ff_reg_out[3:0].Q" output="block_RAM.dout"/>
5    <mux input="{block_RAM.wen block_RAM.din[3:0] block_RAM.addr[10:0]} ff_reg_
   ↪in[15:0].Q"
6         output="{mem_reconfig.wen mem_reconfig.din mem_reconfig.addr}"/>
7    <complete input="block_RAM.clk" output="ff_reg_in[15:0].clk"/>
8    <complete input="block_RAM.clk" output="ff_reg_out[3:0].clk"/>
9  </interconnect>
```

(continues on next page)

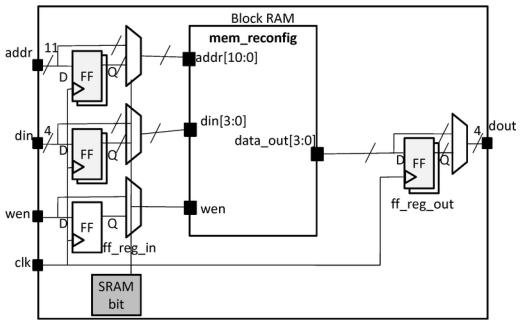


Fig. 6.5: Interconnect within the configurable memory block.

(continued from previous page)

```
10 </pb_type>
```

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 `ff_reg_in[15:0].Q` or from the SPCB inputs `{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 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. The columns occupied by complex blocks of type CLB is defined by gridlocations where `type="col" start="2" repeat="5"` means that every fifth column starting from the second column type memory CLB unless that column is taken up by a block with higher priority (where a bigger number means a higher priority).

```
<!-- 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"/>
<gridlocations>
  <loc type="col" start="2" repeat="5" priority="2"/>
</gridlocations>
```

Configurable Memory Bus-Based Complete Example

```
<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 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>
```

(continues on next page)

(continued from previous page)

```
<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>

<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"/>

  <!-- 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]"/>
```

(continues on next page)

(continued from previous page)

```
      </interconnect>
    </mode>
  </pb_type>

  <interconnect>
    <direct input="{block_RAM.wen block_RAM.din block_RAM.addr}" output="ff_reg_
↪in[15:0].D"/>
    <direct input="mem_reconfig.dout" output="ff_reg_out[3:0].D"/>
    <mux input="mem_reconfig.dout ff_reg_out[3:0].Q" output="block_RAM.dout"/>
    <mux input="{block_RAM.wen block_RAM.din[3:0] block_RAM.addr[10:0]} ff_reg_
↪in[15:0].Q"
         output="{mem_reconfig.wen mem_reconfig.din mem_reconfig.addr}"/>
    <complete input="block_RAM.clk" output="ff_reg_in[15:0].clk"/>
    <complete input="block_RAM.clk" output="ff_reg_out[3:0].clk"/>
  </interconnect>
</pb_type>

  <!-- 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"/>
  <gridlocations>
    <loc type="col" start="2" repeat="5" priority="2"/>
  </gridlocations>
```

6.2.3 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 langauge to describe a common type of configurable multiplier called a fracturable multiplier shown in Fig. 6.6. We first give a step-by-step description on how to construct the multiplier block followed by a complete example.

The large `block_mult` can implement one 36x36 multiplier cluster called a `mult_36x36_slice` or it can implement two divisble 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 mulitplier clusters called `mult_9x9_slice`. Fig. 6.7 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:

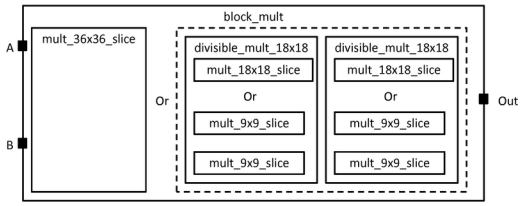


Fig. 6.6: Model of a fracturable multiplier block

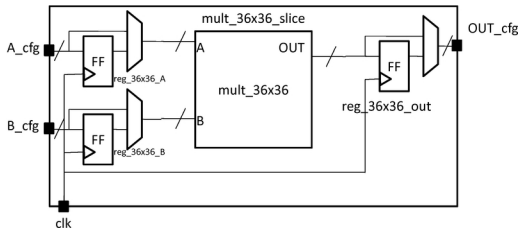


Fig. 6.7: Multiplier slice

```
<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"/>
```

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:

```
<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"/>
```

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 64 flip-flops for the output port. There is one 36x36 multiplier primitive. These primitives are described by four *pb_types* as follows:

```
<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"/>
```

(continues on next page)

(continued from previous page)

```
</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>
```

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>
</pb_type>
```

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>
</mode>
```

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"/>
```

This mode has two additional modes which are the actual 18x18 multiply block or two 9x9 mulitplier 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>
```

(continues on next page)

(continued from previous page)

```
    <interconnect>
      <!-- follows previous pattern for slice definition -->
    </interconnect>
  </mode>

  <mode name="two_mult_9x9">
    <pb_type name="mult_9x9_slice" num_pb="2">
      <!-- follows previous pattern for slice definition -->
    </pb_type>
    <interconnect>
      <!-- follows previous pattern for slice definition -->
    </interconnect>
  </mode>

</pb_type>
```

The interconnect for the divisible 18x18 mode is shown in Fig. 6.8. 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:

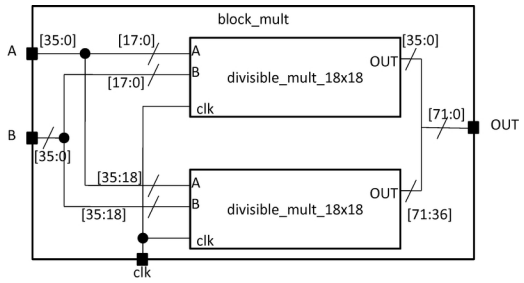


Fig. 6.8: Multiplier Cluster

```
    <interconnect>
      <direct input="block_mult.A" output="divisible_mult_18x18[1:0].A"/>
      <direct input="block_mult.B" output="divisible_mult_18x18[1:0].B"/>
      <direct input="divisible_mult_18x18[1:0].OUT" output="block_mult.OUT"/>
      <complete input="block_mult.clk" output="divisible_mult_18x18[1:0].clk"/>
    </interconnect>
  </mode>
</pb_type>
```

Fracturable Multiplier Bus-Based Complete Example

```
<!-- Example of a fracturable mutliplier 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"/>
```

(continues on next page)

(continued from previous page)

```
  <clock name="clk"/>

  <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>

      <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>
    </pb_type>
    <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>
  </mode>

  <mode  name="two_divisible_mult_18x18">
```

(continues on next page)

(continued from previous page)

```
    <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"/>

      <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 name="mult_18x18" blif_model=".subckt mult" num_pb="1">
            <input name="A" num_pins="18"/>
            <input name="B" num_pins="18"/>
            <output name="OUT" num_pins="36"/>
          </pb_type>

          <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>
        </pb_type>
        <interconnect>
          <direct input="divisible_mult_18x18.A" output="mult_18x18_slice.A_cfg"/>
```

(continues on next page)

(continued from previous page)

```
          <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 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.OUT" output="reg_9x9_out[17:0].D"/>
            <mux input="mult_9x9.OUT reg_9x9_out[17:0].Q" output="mult_9x9_slice.OUT_
↪cfg"/>

            <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"/>
          </interconnect>
        </pb_type>
        <interconnect>
          <direct input="divisible_mult_18x18.A" output="mult_9x9_slice[1:0].A_cfg"/>
          <direct input="divisible_mult_18x18.B" output="mult_9x9_slice[1:0].A_cfg"/>
          <direct input="mult_9x9_slice[1:0].OUT_cfg" output="divisible_mult_18x18.OUT
↪"/>
          <complete input="divisible_mult_18x18.clk" output="mult_9x9_slice[1:0].clk"/
↪>
```

(continues on next page)

(continued from previous page)

```
        </interconnect>
      </mode>
    </pb_type>
    <interconnect>
      <direct input="block_mult.A" output="divisible_mult_18x18[1:0].A"/>
      <direct input="block_mult.B" output="divisible_mult_18x18[1:0].B"/>
      <direct input="divisible_mult_18x18[1:0].OUT" output="block_mult.OUT"/>
      <complete input="block_mult.clk" output="divisible_mult_18x18[1:0].clk"/>
    </interconnect>
  </mode>

  <fc_in type="frac">0.15</fc_in>
  <fc_out type="frac">0.125</fc_out>

  <pinlocations pattern="custom">
    <loc side="left">a[35:0]</loc>
    <loc side="left" offset="1">b[35:0]</loc>
    <loc side="right">out[19:0]</loc>
    <loc side="right" offset="1">out[39:20]</loc>
    <loc side="right" offset="2">out[63:40]</loc>
  </pinlocations>

  <gridlocations>
    <loc type="col" start="4" repeat="5" priority="2"/>
  </gridlocations>
</pb_type>
```

Architecture Description Examples:

6.2.4 Fracturable Multiplier Example

A 36x36 multiplier fracturable into 18x18s and 9x9s

```
<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"/>

        <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"/>
            <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"/>
```

(continues on next page)

(continued from previous page)

```
            <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>
        </pb_type>
        <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>
      </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"/>
          <output name="OUT_cfg" num_pins="36"/>
```

(continues on next page)

(continued from previous page)

```
            <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>

            <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.b"/>
              </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_slice.OUT_cfg"/>
                <C_constant C="1.89e-13" in_port="mult_18x18.out" out_port="mult_
↪18x18_slice.OUT_cfg"/>
              </direct>
            </interconnect>
          </pb_type>
          <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>
            <direct name="out2out" input="mult_18x18_slice.OUT_cfg" output="divisible_
↪mult_18x18.out">
              <delay_constant max="2.03e-13" min="1.89e-13" in_port="mult_18x18_slice.
↪OUT_cfg" out_port="divisible_mult_18x18.out"/>
              <C_constant C="1.89e-13" in_port="mult_18x18_slice.OUT_cfg" out_port=
↪"divisible_mult_18x18.out"/>
            </direct>
          </interconnect>
        </mode>
      </pb_type>
      <interconnect>
```

(continues on next page)

(continued from previous page)

```
        <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"/>
        </direct>
        <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"/>
        </direct>
        <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"/>
        </direct>
      </interconnect>
    </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" area="4000
↪">
          <input name="a" num_pins="36"/>
          <input name="b" num_pins="36"/>
          <output name="out" num_pins="72"/>
          <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_36x36_slice.A_cfg" output="mult_36x36.a">
            <delay_constant max="2.03e-13" min="1.89e-13" in_port="mult_36x36_slice.A_
↪cfg" out_port="mult_36x36.a"/>
            <C_constant C="1.89e-13" in_port="mult_36x36_slice.A_cfg" out_port="mult_
↪36x36.a"/>
          </direct>
          <direct name="b2b" input="mult_36x36_slice.B_cfg" output="mult_36x36.b">
            <delay_constant max="2.03e-13" min="1.89e-13" in_port="mult_36x36_slice.B_
↪cfg" out_port="mult_36x36.b"/>
            <C_constant C="1.89e-13" in_port="mult_36x36_slice.B_cfg" out_port="mult_
↪36x36.b"/>
          </direct>
          <direct name="out2out" input="mult_36x36.out" output="mult_36x36_slice.OUT_
↪cfg">
            <delay_constant max="2.03e-13" min="1.89e-13" in_port="mult_36x36.out"␣
↪out_port="mult_36x36_slice.OUT_cfg"/>
            <C_constant C="1.89e-13" in_port="mult_36x36.out" out_port="mult_36x36_
↪slice.OUT_cfg"/>
          </direct>
```

(continues on next page)

(continued from previous page)

```
        </interconnect>
      </pb_type>
      <interconnect>
        <direct name="a2a" input="mult_36.a" output="mult_36x36_slice.A_cfg">
          <delay_constant max="2.03e-13" min="1.89e-13" in_port="mult_36.a" out_port=
↪"mult_36x36_slice.A_cfg"/>
          <C_constant C="1.89e-13" in_port="mult_36.a" out_port="mult_36x36_slice.A_
↪cfg"/>
        </direct>
        <direct name="b2b" input="mult_36.b" output="mult_36x36_slice.B_cfg">
          <delay_constant max="2.03e-13" min="1.89e-13" in_port="mult_36.b" out_port=
↪"mult_36x36_slice.B_cfg"/>
          <C_constant C="1.89e-13" in_port="mult_36.b" out_port="mult_36x36_slice.B_
↪cfg"/>
        </direct>
        <direct name="out2out" input="mult_36x36_slice.OUT_cfg" output="mult_36.out">
          <delay_constant max="2.03e-13" min="1.89e-13" in_port="mult_36x36_slice.OUT_
↪cfg" out_port="mult_36.out"/>
          <C_constant C="1.89e-13" in_port="mult_36x36_slice.OUT_cfg" out_port="mult_
↪36.out"/>
        </direct>
      </interconnect>
    </mode>



  <fc_in type="frac"> 0.15</fc_in>
  <fc_out type="frac"> 0.125</fc_out>
  <pinlocations pattern="spread"/>

  <gridlocations>
    <loc type="col" start="4" repeat="5" priority="2"/>
  </gridlocations>
</pb_type>
```

6.2.5 Configurable Memory Block Example

A memory block with a reconfigurable aspect ratio.

```
<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"/>

    <mode name="mem_1024x16_sp">
      <pb_type name="mem_1024x16_sp" blif_model=".subckt single_port_ram" class=
↪"memory" num_pb="1" area="1000">
        <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"/>
```

(continues on next page)

(continued from previous page)

```
        <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>
        <direct name="data1" input="memory.data[15:0]" output="mem_1024x16_sp.data">
        </direct>
        <direct name="writeen1" input="memory.we1" output="mem_1024x16_sp.we">
        </direct>
        <direct name="dataout1" input="mem_1024x16_sp.out" output="memory.out[15:0]">
        </direct>
        <direct name="clk" input="memory.clk" output="mem_1024x16_sp.clk">
        </direct>
      </interconnect>
    </mode>
    <mode name="mem_2048x8_dp">
      <pb_type name="mem_2048x8_dp" blif_model=".subckt dual_port_ram" class="memory"␣
↪num_pb="1" area="1000">
        <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>
        <direct name="address2" input="memory.addr2[10:0]" output="mem_2048x8_dp.addr2
↪">
        </direct>
        <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
↪" num_pb="1" area="1000">
        <input name="addr" num_pins="11" port_class="address"/>
```

(continues on next page)

(continued from previous page)

```
        <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.out" 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"␣
↪num_pb="1" area="1000">
        <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>
      <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">
```

(continues on next page)

(continued from previous page)

```
      <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.addr
↪">
        </direct>
        <direct name="data1" input="memory.data[3:0]" output="mem_4096x4_sp.data">
        </direct>
        <direct name="writeen1" input="memory.we1" output="mem_4096x4_sp.we">
        </direct>
        <direct name="dataout1" input="mem_4096x4_sp.out" 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"/>
        <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>
```

(continues on next page)

(continued from previous page)

```
    </mode>

    <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>
        <direct name="data1" input="memory.data[1:0]" output="mem_8192x2_sp.data">
        </direct>
        <direct name="writeen1" input="memory.we1" output="mem_8192x2_sp.we">
        </direct>
        <direct name="dataout1" input="mem_8192x2_sp.out" output="memory.out[1:0]">
        </direct>
        <direct name="clk" input="memory.clk" output="mem_8192x2_sp.clk">
        </direct>
      </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="addr1" num_pins="14" port_class="address1"/>
        <input name="addr2" num_pins="14" port_class="address2"/>
        <input name="data1" num_pins="1" port_class="data_in1"/>
        <input name="data2" num_pins="1" 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="1" port_class="data_out1"/>
        <output name="out2" num_pins="1" port_class="data_out2"/>
        <clock name="clk" num_pins="1" port_class="clock"/>
      </pb_type>
      <interconnect>
        <direct name="address1" input="memory.addr1[13:0]" output="mem_16384x1_dp.
↪addr1">
        </direct>
        <direct name="address2" input="memory.addr2[13:0]" output="mem_16384x1_dp.
↪addr2">
        </direct>
        <direct name="data1" input="memory.data[0:0]" output="mem_16384x1_dp.data1">
        </direct>
        <direct name="data2" input="memory.data[1:1]" output="mem_16384x1_dp.data2">
        </direct>
        <direct name="writeen1" input="memory.we1" output="mem_16384x1_dp.we1">
        </direct>
        <direct name="writeen2" input="memory.we2" output="mem_16384x1_dp.we2">
        </direct>
        <direct name="dataout1" input="mem_16384x1_dp.out1" output="memory.out[0:0]">
        </direct>
        <direct name="dataout2" input="mem_16384x1_dp.out2" output="memory.out[1:1]">
        </direct>
```

(continues on next page)

(continued from previous page)

```
        <direct name="clk" input="memory.clk" output="mem_16384x1_dp.clk">
        </direct>
      </interconnect>
    </mode>

    <mode name="mem_16384x1_sp">
      <pb_type name="mem_16384x1_sp" blif_model=".subckt single_port_ram" class=
↪"memory" num_pb="1" area="1000">
        <input name="addr" num_pins="14" port_class="address"/>
        <input name="data" num_pins="1" port_class="data_in"/>
        <input name="we" num_pins="1" port_class="write_en"/>
        <output name="out" num_pins="1" port_class="data_out"/>
        <clock name="clk" num_pins="1" port_class="clock"/>
      </pb_type>
      <interconnect>
        <direct name="address1" input="memory.addr1[13:0]" output="mem_16384x1_sp.addr
↪">
        </direct>
        <direct name="data1" input="memory.data[0:0]" output="mem_16384x1_sp.data">
        </direct>
        <direct name="writeen1" input="memory.we1" output="mem_16384x1_sp.we">
        </direct>
        <direct name="dataout1" input="mem_16384x1_sp.out" output="memory.out[0:0]">
        </direct>
        <direct name="clk" input="memory.clk" output="mem_16384x1_sp.clk">
        </direct>
      </interconnect>
    </mode>



  <fc_in type="frac"> 0.15</fc_in>
  <fc_out type="frac"> 0.125</fc_out>
  <pinlocations pattern="spread"/>
  <gridlocations>
    <loc type="col" start="2" repeat="5" priority="2"/>
  </gridlocations>
</pb_type>
```

6.2.6 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. 6.9, as follows:

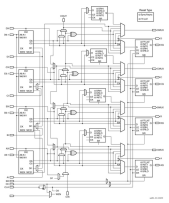


Fig. 6.9: Commercial FPGA logic block slice (Xilinx Virtex-6)

```
<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"/>
  <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"/>
```

(continues on next page)

(continued from previous page)

```
    <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"/>
    <output name="Q" num_pins="1"/>
    <clock name="CK" num_pins="1"/>
  </pb_type>
  <!-- TODO: Add in ability to specify constants such as gnd/vcc -->

  <interconnect>
    <direct name="fraclutA" input="{v6_lslice.A v6_lslice.B v6_lslice.C v6_lslice.D}"␣
↪output="fraclut.A"/>
    <direct name="fraclutW" input="{v6_lslice.A v6_lslice.B v6_lslice.C v6_lslice.D}"␣
↪output="fraclut.W"/>
    <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="carryO6" input="fraclut.O6" output="carry.xor"/>
    <direct name="carrymuxxor" input="carry[2:0].cmux_out" output="carry[3:1].cmuxxor
↪"/>
    <direct name="carrymmux" input="{fraclut[3].O6 fraclut[2].O6 fraclut[2].O6␣
↪fraclut[1].O6 fraclut[1].O6 fraclut[0].O6}" output="carry[2:0].mmux"/>
    <direct name="carrymmux_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="ABCD" input="fraclut[3:0].O6" output="{v6_lslice.Dout v6_lslice.
↪Cout v6_lslice.Bout v6_lslice.Aout}"/>
    <direct name="Q" input="ff_big.Q" output="{DQ CQ BQ AQ}"/>

    <mux name="ff_smallA" input="v6_lslice.AX fraclut[0].O5" output="ff_small[0].D"/>
    <mux name="ff_smallB" input="v6_lslice.BX fraclut[1].O5" output="ff_small[1].D"/>
    <mux name="ff_smallC" input="v6_lslice.CX fraclut[2].O5" output="ff_small[2].D"/>
    <mux name="ff_smallD" input="v6_lslice.DX fraclut[3].O5" output="ff_small[3].D"/>

    <mux name="ff_bigA" input="fraclut[0].O5 fraclut[0].O6 carry[0].cmux_out carry[0].
↪mmux_out carry[0].xor_out" output="ff_big[0].D"/>
    <mux name="ff_bigB" input="fraclut[1].O5 fraclut[1].O6 carry[1].cmux_out carry[1].
↪mmux_out carry[1].xor_out" output="ff_big[1].D"/>
    <mux name="ff_bigC" input="fraclut[2].O5 fraclut[2].O6 carry[2].cmux_out carry[2].
↪mmux_out carry[2].xor_out" output="ff_big[2].D"/>
```

(continues on next page)

(continued from previous page)

```
    <mux name="ff_bigD" input="fraclut[3].O5 fraclut[3].O6 carry[3].cmux_out carry[3].
↪mmux_out carry[3].xor_out" output="ff_big[3].D"/>

    <mux name="AMUX" input="fraclut[0].O5 fraclut[0].O6 carry[0].cmux_out carry[0].
↪mmux_out carry[0].xor_out ff_small[0].Q" output="AMUX"/>
    <mux name="BMUX" input="fraclut[1].O5 fraclut[1].O6 carry[1].cmux_out carry[1].
↪mmux_out carry[1].xor_out ff_small[1].Q" output="BMUX"/>
    <mux name="CMUX" input="fraclut[2].O5 fraclut[2].O6 carry[2].cmux_out carry[2].
↪mmux_out carry[2].xor_out ff_small[2].Q" output="CMUX"/>
    <mux name="DMUX" input="fraclut[3].O5 fraclut[3].O6 carry[3].cmux_out carry[3].
↪mmux_out carry[3].xor_out ff_small[3].Q" output="DMUX"/>

    <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"/>
    <mux name="AfraclutDI1" input="v6_lslice.AI v6_lslice.BI v6_lslice.DI fraclut[2].
↪MC31 fraclut[1].MC31" output="fraclut[0].DI1"/>

    <mux name="carrymuxxorA" input="v6_lslice.AX v6_lslice.CIN" output="carry[0].
↪muxxor"/>
    <mux name="carrymuxA" input="v6_lslice.AX fraclut[0].O5" output="carry[0].cmux"/>
    <mux name="carrymuxB" input="v6_lslice.BX fraclut[1].O5" output="carry[1].cmux"/>
    <mux name="carrymuxC" input="v6_lslice.CX fraclut[2].O5" output="carry[2].cmux"/>
    <mux name="carrymuxD" input="v6_lslice.DX fraclut[3].O5" output="carry[3].cmux"/>


    <complete name="clock" input="v6_lslice.CLK" output="{ff_small.CK ff_big.CK}"/>
    <complete name="ce" input="v6_lslice.CE" output="{ff_small.CE ff_big.CE}"/>
    <complete name="SR" input="v6_lslice.SR" output="{ff_small.SR ff_big.SR}"/>
  </interconnect>
</pb_type>
```

Modeling Guides:

6.2.7 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 dependancies are between the pins.
2. Specifying the physical delay values

These two steps separatate 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,

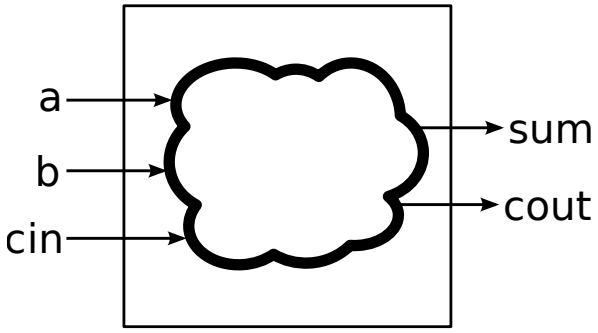


Fig. 6.10: Full Adder

where `a`, `b` and `cin` are combinational inputs, and `sum` and `cout` are combinational outputs.

We can model these timing dependancies on the model with the `combinational_sink_ports`, which specifies the output ports which are dependant on an input port:

```
<model name="adder">
  <input_ports>
    <port name="a" combinational_sink_ports="sum cout"/>
    <port name="b" combinational_sink_ports="sum cout"/>
    <port name="cin" combinational_sink_ports="sum cout"/>
  </input_ports>
  <output_ports>
    <port name="sum"/>
    <port name="cout"/>
  </output_ports>
</model>
```

The physical timing delays are specified on any `pb_type` instances of the adder model. For example:

```
<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.

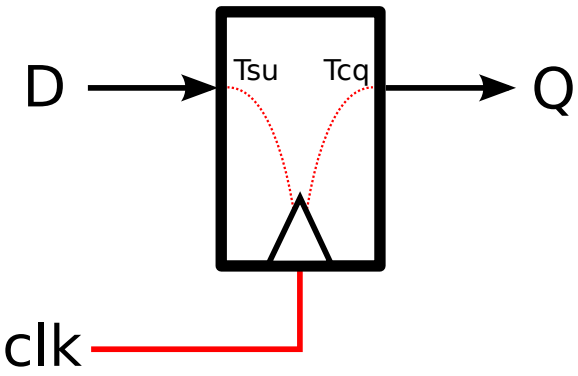


Fig. 6.11: DFF

Sequential model ports are specified by providing the `clock="<name>"` attribute, where `<name>` is the name of the associated clock ports. The assoicated clock port must have `is_clock="1"` specified to indicate it is a clock.

```
<model name="dff">
  <input_ports>
    <port name="d" clock="clk"/>
    <port name="clk" is_clock="1"/>
  </input_ports>
  <output_ports>
    <port name="q" clock="clk/>
  </output_ports>
</model>
```

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.

```
<pb_type 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"/>
</pb_type>
```

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`).

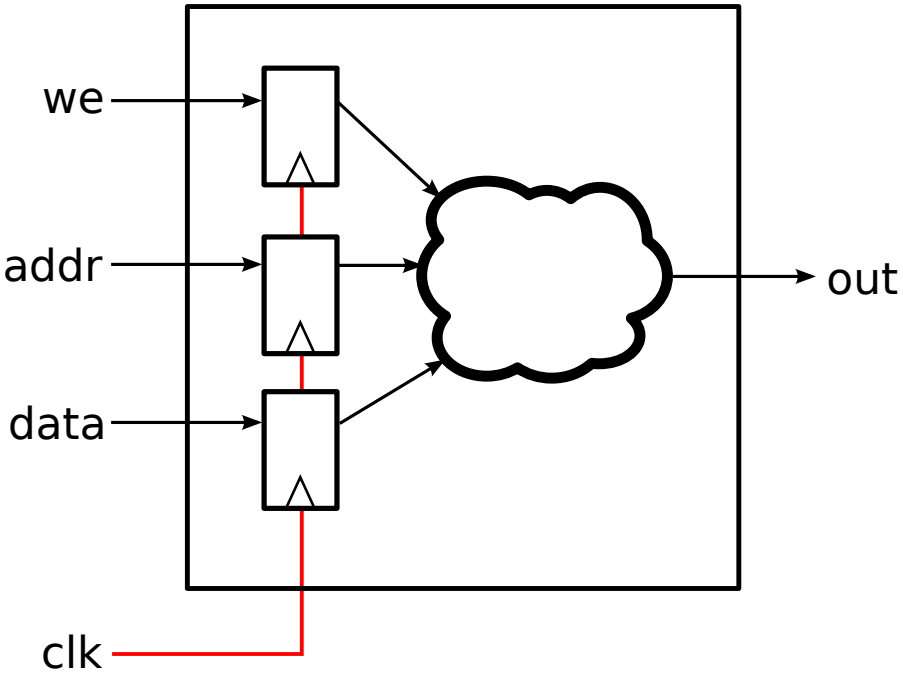


Fig. 6.12: Mixed sequential/combinational single port ram

However the output port (`out`) is a combinational output, connected internally to the `we`, `addr` and `data` input registers.

```
<model name="single_port_ram_mixed">
  <input_ports>
    <port name="we" combinational_sink_ports="out"/>
    <port name="addr" combinational_sink_ports="out"/>
    <port name="data" combinational_sink_ports="out"/>
    <port name="clk" is_clock="1"/>
  </input_ports>
  <output_ports>
    <port name="out"/>
  </output_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)

```
<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"/>
```

(continues on next page)

(continued from previous page)

```
  <!-- 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="800e-12" in_port="mem_sp.addr" out_port="mem_sp.out"/>
  <delay_constant max="800e-12" in_port="mem_sp.data" out_port="mem_sp.out"/>
  <delay_constant max="800e-12" in_port="mem_sp.we" out_port="mem_sp.out"/>
</pb_type>
```

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.

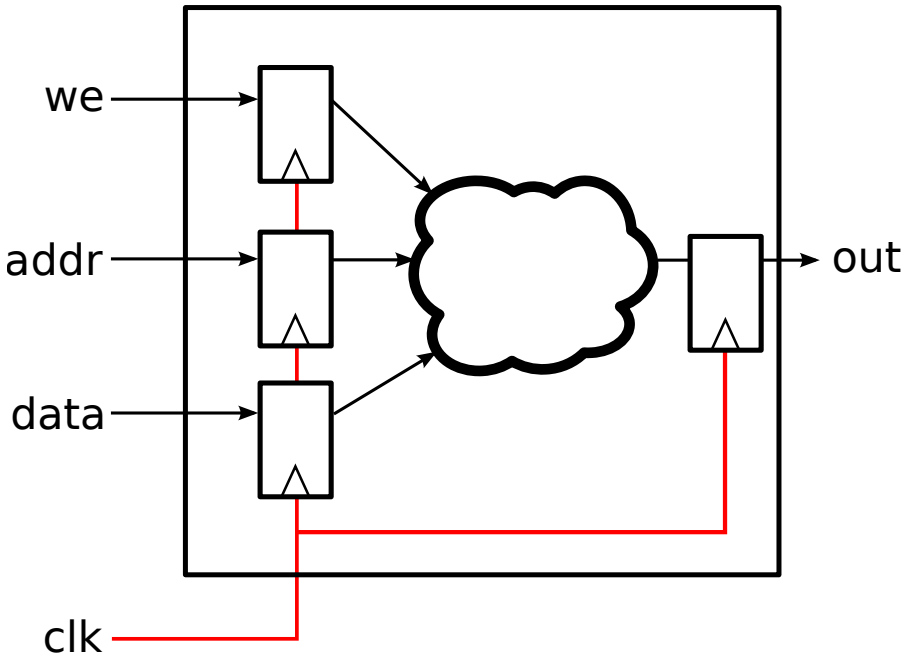


Fig. 6.13: Sequential single port ram

```
<model name="single_port_ram_seq">
  <input_ports>
    <port name="we" clock="clk" combinational_sink_ports="out"/>
```

(continues on next page)

(continued from previous page)

```
    <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>
```

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.

```
<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.

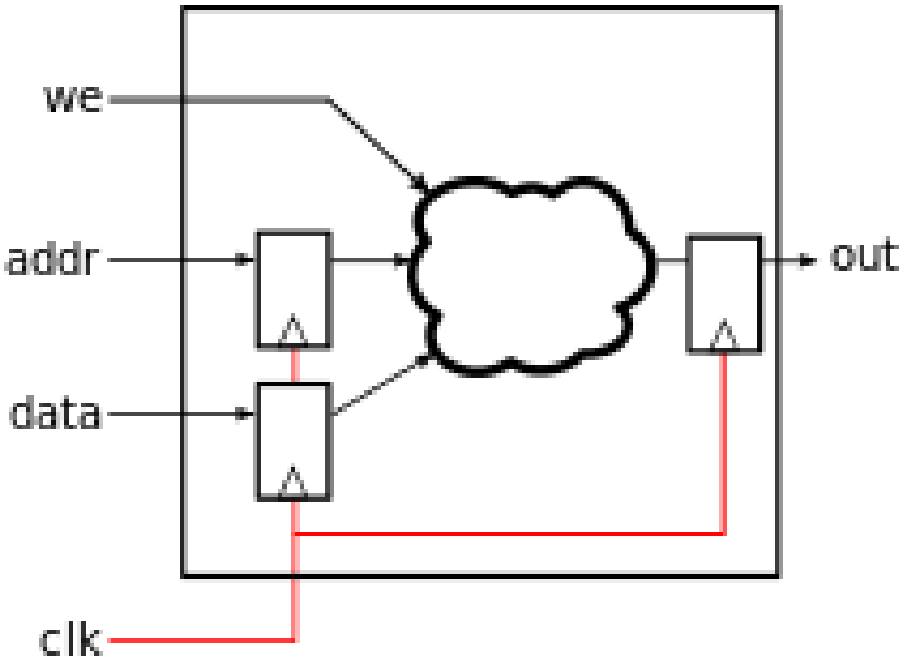


Fig. 6.14: Sequential single port ram with a combinational input

```
<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>
```

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 effects the `we` to `out` path for an effective delay of 860ps.

```
<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"/>

  <!-- 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"/>
```

(continues on next page)

(continued from previous page)

```
  <!-- 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"/>

  <!-- External combinational delay -->
  <delay_constant max="800e-12" in_port="mem_sp.we" out_port="mem_sp.out"/>

  <!-- 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"/>

  <!-- 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>
```

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`.

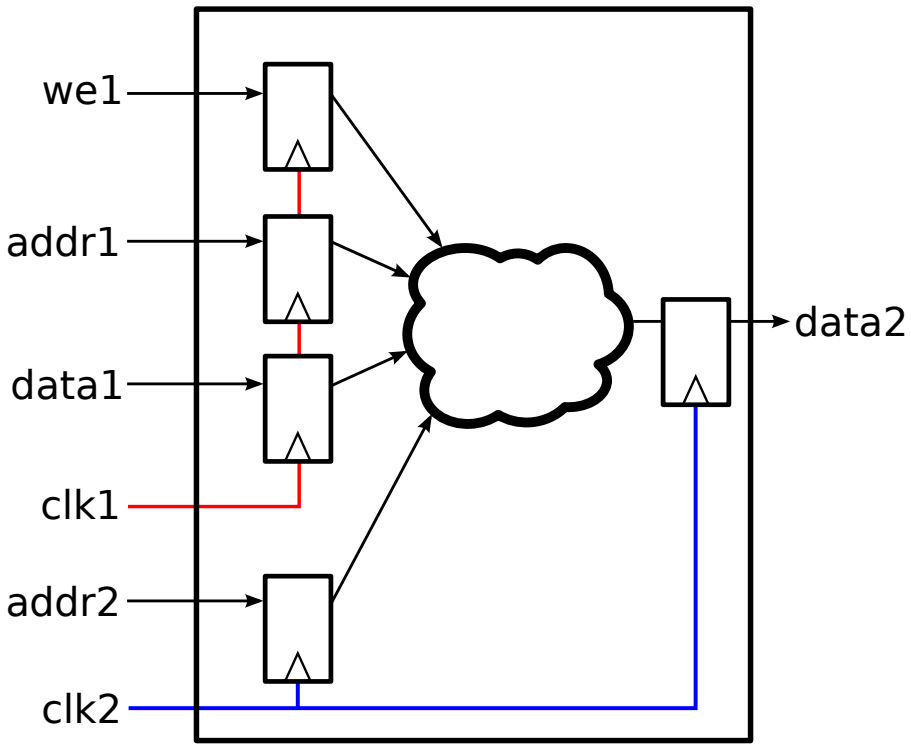


Fig. 6.15: Multi-clock sequential simple dual port ram

```
<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.

```
<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"/>
  <delay_constant max="740e-12" in_port="mem_dp.we1" out_port="mem_dp.data2"/>
  <delay_constant max="740e-12" in_port="mem_dp.addr2" out_port="mem_dp.data2"/>

  <!-- Internal output register timing -->
  <T_setup value="60e-12" port="mem_dp.data2" clock="clk2"/>

  <!-- External output register timing -->
  <T_clock_to_Q max="300e-12" port="mem_dp.data2" clock="clk2"/>
</pb_type>
```

Clock Generators

Some blocks (such as PLLs) generate clocks on-chip. To ensure that these generated clocks are identified as clocks, the associated model output port should be marked with `is_clock="1"`.

As an example consider the following simple PLL model:

```
<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.

6.3 Running the Titan Benchmarks

This tutorial describes how to run the *Titan benchmarks* with VTR.

6.3.1 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:

```
$ 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
..................................................................................
↪..............
Downloading 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
```

(continues on next page)

(continued from previous page)

```
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/benchmarks/titan23/cholesky_mc/netlists/cholesky_mc_
↪stratixiv_arch_timing.blif to ./vtr_flow/benchmarks/titan_blif/cholesky_mc_
↪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
```

(continues on next page)

(continued from previous page)

```
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.

6.3.2 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
```

6.4 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)

6.4.1 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:

```
$ cp $VTR_ROOT/vtr_flow/benchmarks/vtr_benchmarks_blif/stereovision3.blif .
$ cp $VTR_ROOT/vtr_flow/arch/timing/k6_N10_40nm.xml .
```

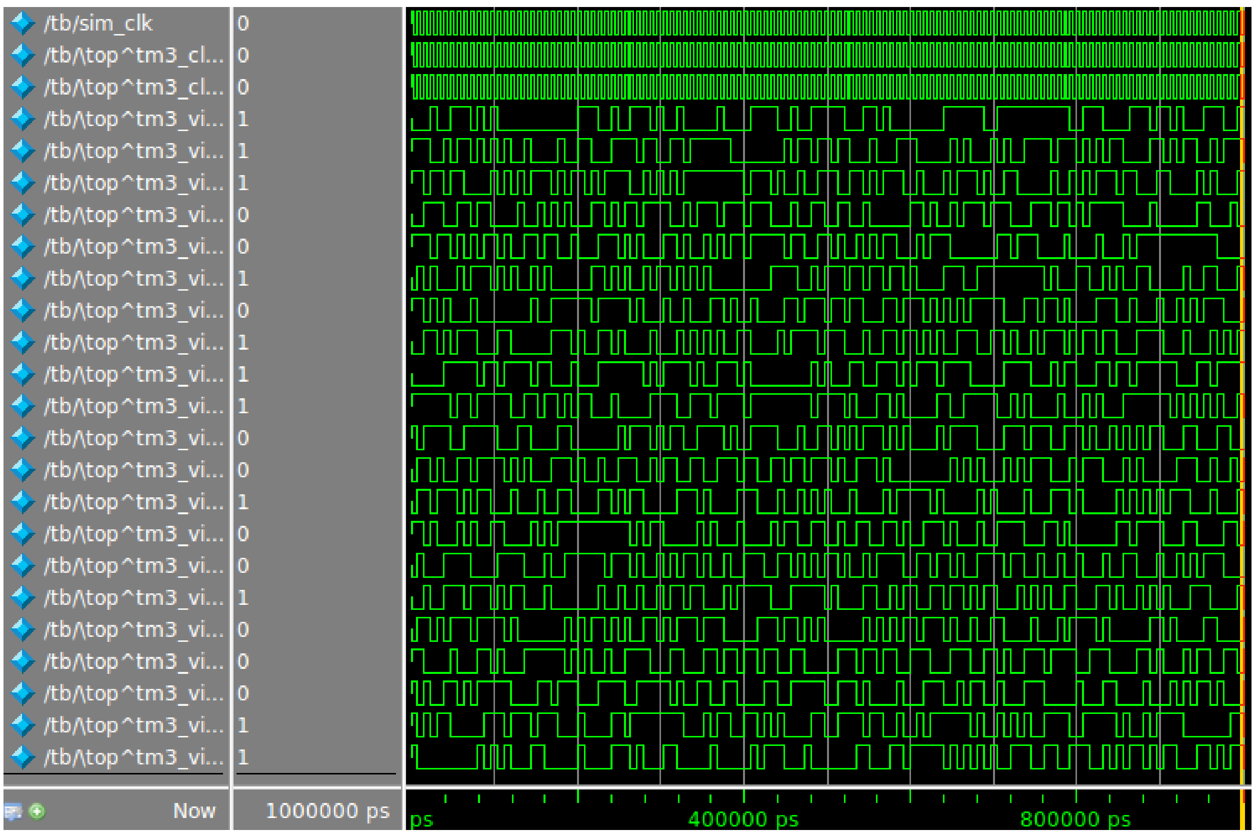


Fig. 6.16: Timing simulation waveform for `stereovision3`

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
```

6.4.2 Inspecting the Post-Implementation Netlist

Lets take a quick look at the generated files.

First is a snippet of the verilog netlist:

Listing 6.1: Verilog netlist snippet

```
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'b0000000000000000000000000000000000010000100100010000000100000010)
) \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 ),
    .clock(\latch_top^FF_NODE~387_clock_0_0 )
);
```

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`

Lets now take a look at the Standard Delay Fromat (SDF) file:

Listing 6.2: SDF snippet

```
(CELL
    (CELLTYPE "fpga_interconnect")
    (INSTANCE routing_segment_lut_n616_output_0_0_to_lut_n497_input_0_4)
    (DELAY
        (ABSOLUTE
            (IOPATH datain dataout (312.648:312.648:312.648) (312.648:312.648:312.
↪648))
        )
    )
)

(CELL
    (CELLTYPE "LUT_K")
    (INSTANCE lut_n452)
    (DELAY
        (ABSOLUTE
            (IOPATH in[0] out (261:261:261) (261:261:261))
            (IOPATH in[2] out (261:261:261) (261:261:261))
            (IOPATH in[3] out (261:261:261) (261:261:261))
            (IOPATH in[4] out (261:261:261) (261:261:261))
        )
    )
)

(CELL
    (CELLTYPE "DFF")
    (INSTANCE latch_top\^FF_NODE\~387)
    (DELAY
        (ABSOLUTE
            (IOPATH (posedge clock) Q (124:124:124) (124:124:124))
        )
    )
    (TIMINGCHECK
        (SETUP D (posedge clock) (66:66:66))
    )
)
```

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 6.1.

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.

6.4.3 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 6.3: The test bench `tb.sv`.

```
`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_vidin_llc ;
logic \top^tm3_vidin_vs ;
logic \top^tm3_vidin_href ;
logic \top^tm3_vidin_cref ;
logic \top^tm3_vidin_rts0 ;
logic \top^tm3_vidin_vpo~0 ;
logic \top^tm3_vidin_vpo~1 ;
logic \top^tm3_vidin_vpo~2 ;
logic \top^tm3_vidin_vpo~3 ;
logic \top^tm3_vidin_vpo~4 ;
logic \top^tm3_vidin_vpo~5 ;
logic \top^tm3_vidin_vpo~6 ;
logic \top^tm3_vidin_vpo~7 ;
logic \top^tm3_vidin_vpo~8 ;
logic \top^tm3_vidin_vpo~9 ;
logic \top^tm3_vidin_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 ;
```

(continues on next page)

(continued from previous page)

```
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);
```

(continues on next page)

(continued from previous page)

```
100     \top^tm3_vidin_vpo~11 <= $urandom_range(1,0);
101     \top^tm3_vidin_vpo~12 <= $urandom_range(1,0);
102     \top^tm3_vidin_vpo~13 <= $urandom_range(1,0);
103     \top^tm3_vidin_vpo~14 <= $urandom_range(1,0);
104     \top^tm3_vidin_vpo~15 <= $urandom_range(1,0);
105 end
106
107 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.

6.4.4 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 6.4: Modelsim do file `tb.do`. Note that *$VTR_ROOT* should be replaced with the relevant path.

```
1  #Enable command logging
2  transcript on
3
4  #Setup working directories
5  if {[file exists gate_work]} {
6      vdel -lib gate_work -all
7  }
8  vlib gate_work
9  vmap work gate_work
10
11 #Load the verilog files
12 vlog -sv -work work {sv_chip3_hierarchy_no_mem_post_synthesis.v}
13 vlog -sv -work work {tb.sv}
14 vlog -sv -work work {$VTR_ROOT/vtr_flow/primitives.v}
15
16 #Setup the simulation
17 vsim -t 1ps -L gate_work -L work -voptargs="+acc" +sdf_verbose +bitblast tb
18
19 #Log signal changes to a VCD file
20 vcd file sim.vcd
21 vcd add /tb/dut/*
22 vcd add /tb/dut/*
23
24 #Setup the waveform viewer
25 log -r /tb/*
26 add wave /tb/*
27 view structure
28 view signals
29
```

(continues on next page)

(continued from previous page)

```
30 #Run the simulation for 1 microsecond
31 run 1us -all
```

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, lets 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`.

CHAPTER 7

Developer Guide

7.1 Building VTR

7.1.1 Overview

VTR uses CMake as it's build system.

CMake provides a portable cross-platform build systems with many useful features.

7.1.2 Unix-like

For unix-like systems we provide a wrapper Makefile which supports the traditional `make` and `make clean` commands, but calls CMake behind the scenes.

Dependencies

For the basic tools you need:

- Bison & Flex
- cmake, make
- A modern C++ compiler supporting C++14 (such as GCC >= 4.9 or clang >= 3.6)

For the VPR GUI you need:

- Cairo
- FreeType
- Xft (libXft + libX11)
- fontconfig

For the regression testing and benchmarking you will need:

- Perl + List::MoreUtils
- Python
- time

It is also recommended you install the following development tools:

- git
- ctags
- gdb
- valgrind

For Docs generation you will need:

- Doxygen
- python-sphinx
- python-sphinx-rtd-theme
- python-recommonmark

Debian & Ubuntu

The following should be enough to get the tools, VPR GUI and tests going on a modern Debian or Ubuntu system:

```
apt-get install \
    build-essential \
    flex \
    bison \
    cmake \
    fontconfig \
    libcairo2-dev \
    libfontconfig1-dev \
    libx11-dev \
    libxft-dev \
    perl \
    liblist-moreutils-perl \
    python \
    time
```

For documentation generation these additional packages are required:

```
apt-get install \
    doxygen \
    python-sphinx \
    python-sphinx-rtd-theme \
    python-recommonmark
```

For development the following additional packages are useful:

```
apt-get install \
    git \
    valgrind \
    gdb \
    ctags
```

Building using the Makefile wrapper

Run `make` from the root of the VTR source tree

```
#In the VTR root
$ make
...
[100%] Built target vpr
```

Specifying the 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):

```
#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 to modify the build configuration.

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

#Build
$ make
```

7.1.3 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 milage 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).

Microsoft Windows

NOTE: VTR support on Microsoft Windows is considered experimental

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 `libgcc` 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 Microsft 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 buid 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.

7.2 Contribution Guidelines

Thanks for considering contributing to VTR! Here are some helpful guidelines to follow.

7.2.1 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 guidlines for *submitting code*.

7.2.2 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/enahcement 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.

7.3 Commit Procedures

7.3.1 For external developers

See Submitting Code to VTR.

7.3.2 For developers with commit rights

The guiding principle in internal development is to submit your work into the repository without breaking other people's work. When you commit, make sure that the repository compiles, that the flow runs, and that you did not clobber someone else's work. In the event that you are responsible for "breaking the build", fix the build at top priority.

We have some guidelines in place to help catch most of these problems:

1. Before you push code to the central repository, your code MUST pass the check-in regression test. The check-in regression test is a quick way to test if any part of the VTR flow is broken.

   At a minimum you must run:

   ```
   #From the VTR root directory
   $ ./run_reg_test.pl vtr_reg_basic
   ```

   You may push if all the tests return `All tests passed`.

   However you are strongly encouraged to run both the *basic* and *strong* regression tests:

   ```
   #From the VTR root directory
   $ ./run_reg_test.pl vtr_reg_basic vtr_reg_strong
   ```

   since it performs much more thorough testing.

   It is typically a good idea to run tests regularily as you make changes. If you have failures see *how to debugging failed tests*.

2. The automated BuildBot will perform more extensive regressions tests and mark which revisions are stable.

3. Everyone who is doing development must write regression tests for major feature they create. This ensures regression testing will detect if a feature is broken by someone (or yourself). See *Adding Tests* for details.

4. In the event a regression test is broken, the one responsible for having the test pass is in charge of determining:

   - If there is a bug in the source code, in which case the source code needs to be updated to fix the bug, or
   - If there is a problem with the test (perhaps the quality of the tool did in fact get better or perhaps there is a bug with the test itself), in which case the test needs to be updated to reflect the new changes.

   If the golden results need to be updated and you are sure that the new golden results are better, use the command `../scripts/parse_vtr_task.pl -create_golden your_regression_test_name_here`

5. Keep in sync with the master branch as regularly as you can (i.e. `git pull` or `git pull --rebase`). The longer code deviates from the trunk, the more painful it is to integrate back into the trunk.

Whatever system that we come up with will not be foolproof so be conscientious about how your changes will effect other developers.

7.4 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 tests:

- `vtr_reg_basic`: ~3 minutes serial

  Goal: Quickly check basic VTR flow correctness

  Feature Coverage: Low

  Benchmarks: A few small and simple circuits

  Architectures: A few simple architectures

  Not suitable for evaluating QoR or performance.

- `vtr_reg_strong`: ~30 minutes serial, ~15 minutes with `-j4`

  Goal: Exercise most of VTR's features, moderately fast.

  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

  Not suitable for evaluating QoR or performance.

- `vtr_reg_nightly`: ~15 hours with `-j2`

  Goal: Basic QoR and Performance evaluation.

  Feature Coverage: Medium

  Benchmarks: Small-medium size, diverse. Includes:

  - MCNC20 benchmarks
  - VTR benchmarks
  - Titan 'other' benchmarks (smaller than Titan23)

  Architectures: A wider variety of architectures

- `vtr_reg_weekly`: ~30 hours with `-j2`

  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

These can be run with `run_reg_test.pl`:

```
#From the VTR root directory
$ ./run_reg_test.pl vtr_reg_basic
$ ./run_reg_test.pl vtr_reg_strong
```

The *nightly* and *weekly* regressions require the Titan benchmarks which can be integrated into your VTR tree with:

```
make get_titan_benchmarks
```

They can then be run using `run_reg_test.pl`:

```
$ ./run_reg_test.pl vtr_reg_nightly
$ ./run_reg_test.pl 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.pl 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.pl vtr_reg_basic vtr_reg_strong -j4
```

7.4.1 Odin Functionality Tests

Odin has its own set of tests to verify the correctness of its synthesis results:

- `odin_reg_micro`: ~2 minutes serial
- `odin_reg_full`: ~6 minutes serial

These can be run with:

```
#From the VTR root directory
$ ./run_reg_test.pl odin_reg_micro
$ ./run_reg_test.pl odin_reg_full
```

and should be used when makeing changes to Odin.

7.4.2 Unit Tests

VTR also has a limited set of unit tests, which can be run with:

```
#From the VTR root directory
$ make && make test
```

7.5 Debugging Failed Tests

If a test fails you probably want to look at the log files to determine the cause.

Lets assume we have a failure in `vtr_reg_basic`:

```
#In the VTR root directory
$ ./run_reg_test.pl vtr_reg_strong
#Output trimmed...
regression_tests/vtr_reg_basic/basic_no_timing
----------------------------------------
k4_N10_memSize16384_memData64/ch_intrinsics...failed: vpr
k4_N10_memSize16384_memData64/diffeq1...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`. Lets move to that directory:

```
#From the VTR root directory
$ cd vtr_flow/tasks/regression_tests/vtr_reg_basic/basic_no_timing
$ ls
config  run002  run004
run001  run003  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`. From the output of `run_reg_test.pl` we know that one of the failing architecture/circuit combinations was `k4_N10_memSize16384_memData64/ch_intrinsics`. Each architecture/circuit combination is run in its own sub-folder. Lets move to that directory:

```
$ cd run005/k4_N10_memSize16384_memData64/ch_intrinsics
$ ls
abc.out                      k4_N10_memSize16384_memData64.xml  qor_results.txt
ch_intrinsics.net            odin.out                           thread_1.out
ch_intrinsics.place          output.log                         vpr.out
ch_intrinsics.pre-vpr.blif   output.txt                         vpr_stdout.log
ch_intrinsics.route          parse_results.txt
```

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.

7.6 Adding Tests

Any time you add a feature to VTR you **must** add a test which exercies 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 ~2 mintues it should probably go in a longer running regression test (but see first if you can create a smaller testcase first).

7.6.1 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:

```
#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_fracturable_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_annealing     strong_titan
strong_custom_switch_block   strong_mcnc                 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`

```
$ 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:

```
#From the VTR root
$ cd vtr_flow/tasks
$ ../scripts/run_vtr_task.pl 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.pl` with the `-create_golden` option:

```
$ ../scripts/parse_vtr_task.pl regression_tests/vtr_reg_strong/strong_mytest -
↪create_golden
```

   And check that everything matches with `-check_golden`:

```
$ ../scripts/parse_vtr_task.pl 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`:

```
#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`:

```
#From the VTR root directory
$ ./run_reg_test.pl 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:

```
#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
#Commit the changes, when pushed the test will automatically be picked up by␣
↪BuildBot
$ git commit
```

7.7 Debugging Aids

VTR has support for several additional tools/features to aid debugging.

7.7.1 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
```

7.7.2 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:

```
#From the VTR root directory
$ cmake -D VTR_ASSERT_LEVEL=3 build
$ make
```

this turns on more extensive assertion checking and re-builds VTR.

7.8 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/update_external_subtrees.py` script.

For instance, running `./dev/update_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.

7.8.1 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
   ```

7.8.2 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:

   ```
   <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: `def/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/update_external_subtrees.py $SUBTREE_NAME` to add the subtree.

   For the `libfoo` example above this would be:

   ```
   ./dev/update_external_subtrees.py 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.

7.8.3 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.

7.9 Finding Bugs with Coverity

Coverity Scan is a static code analysis service which can be used to detect bugs.

7.9.1 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

7.9.2 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 gaurded 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++).

1. Run the coverity build tool

```
#From the build directory where we ran cmake
cov-build --dir cov-int make -j8
```

2. Archive the output directory

```
tar -czvf vtr_coverity.tar.gz cov-int
```

3. Submit the archive through the coverity web interface

Once the build has been analyzed you can browse the latest results throught the coverity web interface

7.9.3 No files emitted

If you get the following warning from cov-build:

```
[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`

7.10 Debugging with clang static analyser

First make sure you have clang installed. define clang as the default compiler: `export CC=clang export CXX=clang++`

set the build type to `debug` in makefile

CHAPTER 8

Contact

8.1 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.

8.2 Issue Tracker

Please file bugs on our issue tracker.

Patches are welcome!

CHAPTER 9

Glossary

$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`.

CHAPTER 10

Publications & References

CHAPTER 11

Indices and tables

- genindex
- search

Bibliography

[Xilinx Inc12] *Virtex-6 FPGA Configurable Logic Block User Guide*. Xilinx Inc, ug364 edition, feb 2012. URL: http://www.xilinx.com/support/documentation/user_guides/ug364.pdf.

[BR97a] V. Betz and J. Rose. Cluster-based logic blocks for fpgas: area-efficiency vs. input sharing and size. In *Custom Integrated Circuits Conference*, 551–554. 1997. doi:10.1109/CICC.1997.606687.

[Bet98] Vaughn Betz. *Architecture and CAD for the Speed and Area Optimization of FPGAs*. PhD thesis, University of Toronto, 1998.

[BR96a] Vaughn Betz and Jonathan Rose. Directional bias and non-uniformity in fpga global routing architectures. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '96, 652–659. Washington, DC, USA, 1996. IEEE Computer Society. doi:10.1109/ICCAD.1996.571342.

[BR96b] Vaughn Betz and Jonathan Rose. On biased and non-uniform global routing architectures and cad tools for fpgas. CSRI Technical Report 358, University of Toronto, 1996. URL: http://www.eecg.toronto.edu/~vaughn/papers/techrep.ps.Z.

[BR97b] Vaughn Betz and Jonathan Rose. Vpr: a new packing, placement and routing tool for fpga research. In *Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications*, FPL '97, 213–222. London, UK, 1997. Springer-Verlag. doi:10.1007/3-540-63465-7_226.

[BR00] Vaughn Betz and Jonathan Rose. Automatic generation of fpga routing architectures from high-level descriptions. In *Int. Symp. on Field Programmable Gate Arrays*, FPGA, 175–184. New York, NY, USA, 2000. ACM. doi:10.1145/329166.329203.

[BRM99] Vaughn Betz, Jonathan Rose, and Alexander Marquardt. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, mar 1999. ISBN 0792384601.

[BFRV92] S. Brown, R. Francis, J. Rose, and Z. Vranesic. *Field-Programmable Gate Arrays*. Kluwer Academic Publishers, 1992. ISBN 978-0-7923-9248-4.

[CWW96] Yao-Wen Chang, D. F. Wong, and C. K. Wong. Universal switch modules for fpga design. *ACM Trans. Des. Autom. Electron. Syst.*, 1(1):80–101, January 1996. doi:10.1145/225871.225886.

[CCMB07] S. Cho, S. Chatterjee, A. Mishcenko, and R. Brayton. Efficient fpga mapping using priority cuts. In *FPGA*. 2007.

[CD94] J. Cong and Y. Ding. Flowmap: an optimal technology mapping algorithm for delay optimization in lookup-table based fpga designs. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 13(1):1–12, Jan 1994. doi:10.1109/43.273754.

[FBC08] R. Fung, V. Betz, and W. Chow. Slack allocation and routing to improve fpga timing while repairing short-path violations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(4):686–697, April 2008. doi:10.1109/TCAD.2008.917585.

[HYL+09] Chun Hok Ho, Chi Wai Yu, Philip Leong, Wayne Luk, and Steven J. E. Wilton. Floating-point fpga: architecture and modeling. *IEEE Trans. Very Large Scale Integr. Syst.*, 17(12):1709–1718, December 2009. doi:10.1109/TVLSI.2008.2006616.

[JKGS10] P. Jamieson, K. Kent, F. Gharibian, and L. Shannon. Odin ii-an open-source verilog hdl synthesis tool for cad research. In *International Symposium on Field-Programmable Custom Computing Machines*, 149–156. 2010. doi:10.1109/FCCM.2010.31.

[LW06] Julien Lamoureux and Steven J. E. Wilton. Activity estimation for field-programmable gate arrays. In *International Conference on Field Programmable Logic and Applications*, 1–8. 2006. doi:10.1109/FPL.2006.311199.

[LLTY04] G. Lemieux, E. Lee, M. Tom, and A. Yu. Direction and single-driver wires in fpga interconnect. In *International Conference on Field-Programmable Technology*, 41–48. 2004. doi:10.1109/FPT.2004.1393249.

[LAK+14] Jason Luu, Nooruddin Ahmed, Kenneth B. Kent, Jason Anderson, Jonathan Rose, Vaughn Betz, Jeffrey Goeders, Michael Wainberg, Andrew Somerville, Thien Yu, Konstantin Nasartschuk, Miad Nasr, Sen Wang, and Tim Liu. VTR 7.0: Next Generation Architecture and CAD System for FPGAs. *ACM Transactions on Reconfigurable Technology and Systems*, 7(2):1–30, jun 2014. doi:10.1145/2617593.

[LAR11] Jason Luu, Jason Anderson, and Jonathan Rose. Architecture description and packing for logic blocks with hierarchy, modes and complex interconnect. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, 227–236. New York, NY, USA, 2011. ACM. doi:10.1145/1950413.1950457.

[LKJ+09] Jason Luu, Ian Kuon, Peter Jamieson, Ted Campbell, Andy Ye, Wei Mark Fang, and Jonathan Rose. Vpr 5.0: fpga cad and architecture exploration tools with single-driver routing, heterogeneity and process scaling. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '09, 133–142. New York, NY, USA, 2009. ACM. doi:10.1145/1508128.1508150.

[MBR99] A Marquardt, V. Betz, and J. Rose. Using cluster-based logic blocks and timing-driven packing to improve fpga speed and density. In *FPGA*, 37–46. 1999. doi:10.1145/296399.296426.

[MBR00] Alexander Marquardt, Vaughn Betz, and Jonathan Rose. Timing-driven placement for fpgas. In *Proceedings of the 2000 ACM/SIGDA Eighth International Symposium on Field Programmable Gate Arrays*, FPGA '00, 203–213. New York, NY, USA, 2000. ACM. doi:10.1145/329166.329208.

[MWL+13] K.E. Murray, S. Whitty, S. Liu, J. Luu, and V. Betz. Titan: enabling large and complex benchmarks in academic cad. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, 1–8. Sept 2013. doi:10.1109/FPL.2013.6645503.

[MWL+15] Kevin E. Murray, Scott Whitty, Suya Liu, Jason Luu, and Vaughn Betz. Timing-driven titan: enabling large benchmarks and exploring the gap between academic and commercial cad. *ACM Trans. Reconfigurable Technol. Syst.*, 8(2):10:1–10:18, March 2015. doi:10.1145/2629579.

[Pet16] Oleg Petelin. Cad tools and architectures for improved fpga interconnect. Master's thesis, University of Toronto, 2016. URL: http://hdl.handle.net/1807/75854.

[PHMB07] J. Pistorius, M. Hutton, A. Mishcenko, and R. Brayton. Benchmarking method and designs targeting logic synthesis for fpgas. In *IWLS*, 230–237. 2007.

[RLY+12] Jonathan Rose, Jason Luu, Chi Wai Yu, Opal Densmore, Jeffrey Goeders, Andrew Somerville, Kenneth B. Kent, Peter Jamieson, and Jason Anderson. The vtr project: architecture and cad for fpgas from verilog to routing. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '12, 77–86. New York, NY, USA, 2012. ACM. doi:10.1145/2145694.2145708.

[SG] Berkeley Logic Synthesis and Verification Group. Abc: a system for sequential synthesis and verification. URL: http://www.eecs.berkeley.edu/~alanmi/abc/.

[Wil97] S. Wilton. *Architectures and Algorithms for Field-Programmable Gate Arrays with Embedded Memories*. PhD thesis, University of Toronto, 1997. URL: http://www.ece.ubc.ca/~stevew/publications.html.

[Yan91] S. Yang. Logic Synthesis and Optimization Benchmarks User Guide 3.0. Technical Report, MCNC, 1991.

[YLS92] H. Youssef, R. B. Lin, and E. Shragowitz. Bounds on net delays for vlsi circuits. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 39(11):815–824, Nov 1992. doi:10.1109/82.204129.

Index

Symbols

A

B

C

D

T

V

W

X

Y