Torch & Charger
User Guide

May 1, 2019

Revision 1.0

Lincoln Laboratory

Massachusetts Institute of Technology

Lexington, Massachusetts

This material is based upon work supported by the Defense Advanced Research Projects Agency under Air Force Contract No. FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LICENSE INFORMATION

TORCH AND CHARGER LICENSE

VERILOG TO ROUTING LICENSE

Torch utilizes the open source Verilog to Routing (VTR) software package, available at https://verilogtorouting.org/. License information for the VTR flow is shown below.

1. INTRODUCTION

1.1 OVERVIEW

This software package, which is based on the open source Verilog to Routing (VTR7) package, provides a platform for designing and configuring custom reconfigurable hardware.

- The Configurable Hardware Generator (***charger***) translates a high level description of hardware into a synthesizable Verilog model.
- The Toolkit for Reconfigurable Hardware (***torch***) is a platform for programming user applications onto custom reconfigurable hardware; it includes synthesis, packing, placement, routing, bitstream generation, timing analysis, and power analysis.
- Both ***torch*** and ***charger*** use XML architecture description syntax (similar to that used in VTR7) to describe target reconfigurable hardware architectures. Additional XML syntax was developed for this software package in order to enable added hardware features and software capabilities.

1.2 SYSTEM REQUIREMENTS

- Hardware: no minimum CPU requirement or hard disk size specified, 16 GB memory
- OS: RedHat Linux 6, 64-bit

1.3 INSTALLATION PROCEDURE

1. Set the TORCHPATH environment variable to the full path of the top level 'TORCH' directory.
2. Add '${TORCHPATH}/bin' to each user's PATH environment variable.
3. Type 'make' at the top-level TORCH directory to build ***torch*** and ***charger*** executables; these executables will be built in the '${TORCHPATH}/bin' subdirectory.
4. Place frequently used XML architecture description files in the '${TORCHPATH}/arch' subdirectory.

2. XML ARCHITECTURE DESCRIPTION

Torch and ***charger*** support almost all XML architecture description syntax supported by VTR7[1]. This section describes additional syntax and architectural features supported by ***torch*** and ***charger***.

2.1 BUILT-IN ARCHITECTURES

Users are encouraged to place frequently used architecture description files in the '${TORCHPATH}/arch' directory. Architecture descriptions located in this directory can be provided to ***torch*** and ***charger*** by name (without the .xml suffix) using the following syntax:

```
/home/kthurmer> torch dct example_arch --pack --place
```

All other architecture descriptions can be accessed by providing a full file path (relative or absolute) and using the '**`–arch_file`**' flag:

```
/home/kthurmer> torch dct ../arch/example2.xml --arch_file --pack --place
```

2.2 CONFIGURATION BITS FOR HARD IP BLOCKS

As in VPR7, custom hard IP blocks may be specified using the **`<models>`** section of the architecture description file. ***Charger*** instantiates these user-defined models according to the port list in the model description, and the architecture designer is expected to provide Verilog models that match this description in order to create a full simulate-able ***charger*** model of the hardware.

Any number of configuration bits may be connected to each custom hard IP block by specifying a configuration port using the syntax shown in Figure 1. In this case, ***torch*** will reserve the requested number of bits in the bitstream for each instance of the hard IP block, and ***charger*** will connect the appropriate bits in configuration memory to the proper port.

```
<models>
  <model name = "input">
    ...
    <prog_port>
      <port name="PROG_PORT" size="3" default="011">
    </prog_port>
```

Figure 1: Specifying a Configuration Port

The ***torch*** end user may specify configuration bits for each hard IP block instance in their Verilog application using a bitstream configuration file (see Section 4.1.1.1). Bits written to this file are placed in the bitstream such that they connect to the corresponding configuration port of their assigned hard IP block

[1] https://docs.verilogtorouting.org/en/latest/arch/#fpga-architecture-description

instance. A detailed description of the meaning of the configuration bits for each user-defined model may be included in the model description, as shown in Figure 2.

```
<prog_port>
  <port name="PROG_PORT" size="3">
    <field name="Pad Enable" nbits="1" start="0" default="1">
      <option bits="0" label="Disable"/>
      <option bits="1" label="Enable"/>
    </field>
    <field name="Drive Strength" nbits="2" start="1" default="00">
      <option bits="00" label=" 2 mA"/>
      <option bits="01" label=" 4 mA"/>
      <option bits="10" label=" 8 mA"/>
      <option bits="11" label="12 mA"/>
    </field>
  </port>
</prog_port>
```

Figure 2: Specifying Detailed Configuration Port Information

The end user can view this information using the '**`–list_bitcfg_opts`**' flag, as shown in Figure 3.

```
/home/kthurmer/> torch example_arch --list_bitcfg_opts
input (3 bits, default: 001)
        [0]    Pad Enable (default: 1)
                0       Disable
                1       Enable
        [2:1] Drive Strength (default: 00)
                00       2 mA
                01       4 mA
                10       8 mA
                11      12 mA
dsp_sram -- This block is not configurable.
dsp_mult -- This block is not configurable.

/home/kthurmer/>
```

Figure 3: Displaying Bitstream Configuration Options

2.3 USER DEFINED IO MODELS

By default, ***torch*** maps top-level inputs and outputs to built-in *input* and *output* models similar to those used by VPR7, and ***charger*** generates corresponding Verilog models, *input.v* and *output.v*. Each built-in model gets one GPIO (see Section 2.4) and one configuration bit; the latter is automatically programmed by ***torch*** as an enable. The architecture designer can optionally override these built-in models by describing custom versions in the **`<models>`** section of the architecture description file (to override the defaults, name the custom models *input* and *output*). In this case, ***charger*** instantiates the custom modules according to the ports listed in the model description, and the architecture designer must specify configuration bits and GPIOs, and must also provide corresponding Verilog models for simulation. Note that if a custom model is specified, ***charger*** does not generate a default model.

2.4 GPIO PORTS

In order to facilitate functional verification, ***charger*** can generate GPIO ports in the top level Verilog model of the hardware and connect them through the physical block (pb) hierarchy to GPIO ports on user-defined and built-in hard IP blocks. These ports are ignored by ***torch***.

```
<models>
  <model name = "input">
    <output_ports>
      <port name="inpad"/>
    </output_ports>
    <gpio_ports>
      <port name="gpio"/>
    </gpio_ports>
```

Figure 4: Specifying a GPIO Port in a User-Defined Model

Figure 4 shows the syntax for specifying a GPIO port in a user-defined model. The architecture designer must also specify how this GPIO connects up through the physical block (pb) hierarchy to the top level ***charger*** model. Figure 5 shows an example pb hierarchy with GPIOs, and Figure 6 shows the corresponding ***charger***-generated Verilog. (Note that the number of GPIO ports specified in the **`<models>`** section must equal the **`num_gpios`** field where that model is instantiated in the pb hierarchy.)

```
<complexblocklist>
    <pb_type name="io_perimeter" capacity="2" num_gpio="2" gpios="A B">
        ...
        <pb_type name="inpad" num_pb="1" blif_model=".input" num_gpio="1" gpios="A">
            <output name="inpad" num_pins="1"/>
        </pb_type>
        ...
        <pb_type name="outpad" num_pb="1" blif_model=".output" num_gpio="1" gpios="B">
            <input name="outpad" num_pins="1"/>
        </pb_type>
```

Figure 5: Specifying GPIOs in the Physical Block (pb) Hierarchy

```
module io_perimeter (
        outpad,
        inpad,
        GPIO_A,
        GPIO_B,
        PROG_BIT );
        ...

        inpad m_inpad_0(
                .inpad(inpad_0_inpad),
                .gpio(GPIO_A),
                .PROG_PORT(PROG_BIT[3:1])
        );

        outpad m_outpad_0(
                .outpad(outpad_0_outpad),
                .GPIO_0(GPIO_B),
                .PROG_BIT(PROG_BIT[4])
        );
```

Figure 6: GPIOs Generated by *charger*

2.5 EQUIVALENT TYPES

Torch and ***charger*** allow multiple equivalent top level physical block (pb) types with different capacities, timing information, and power information. Each type within a family of equivalent types should list all types with which it is equivalent using the following syntax:

```
<pb_type name="io_perimeter" capacity="2" num_gpio="2" gpios="A B">
    <equivalenttypes>
        <type name="io_3d"/>
        <type name="io_dense"/>
    </equivalenttypes>
```

If a cluster is assigned to one of a family of equivalent types, the user may move it to a location of equivalent type using a placement constraints file (see Section 4.1.3.1). During placement, clusters assigned to types within a family of equivalent types may be swapped to locations of any type within the family.

Note that ***torch*** and ***charger*** do not verify equivalence.

2.6 ADDITIONAL HARDWARE FLEXIBILITY

Torch and ***charger*** enable a flexible hardware layout via the following syntax enhancements:

- *Variable tile widths*: Top level physical block (tile) types can have any integer height and width:

```
<pb_type name="dsp" width="2" height="2">
```

- *Explicit gridlocations*: Single instances of top level physical block (tile) types maybe be specified at arbitrary (x,y) locations using the following syntax:

```
<gridlocations>
    <loc type="explicit" ncoords="120"
            xcoords="
               0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
            1  1  2  2  3  3  4  4  5  5  6  6  7  7  8  8  9  9  10 10 11 11 12 12 13 13 14 14 15 15 16 16
            17 17 18 18 19 19 20 20 21 21 22 22 23 23 24 24 25 25 26 26 27 27 28 28 29 29 30 30
            31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31"
            ycoords="
               1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
            0  31 0  31 0  31 0  31 0  31 0  31 0  31 0  31 0  31 0  31 0  31 0  31 0  31 0  31 0  31 0  31
            0  31 0  31 0  31 0  31 0  31 0  31 0  31 0  31 0  31 0  31 0  31 0  31 0  31 0  31
            1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30"
    />
</gridlocations>
```

- *Conditional direct connections*: Since ***torch*** and ***charger*** allow irregular heterogeneous architectures, each direct clb-to-clb connection in the **`<directlist>`** section is conditional on the presence of an instance of the correct "to" type at the specified offset from each instance of the "from" type. A pin may be listed in multiple direct connections; the first correct match will be implemented in the routing graph, and, if no correct match is found, the pin will be connected to routing tracks in the same manner as pins not involved in direct connections.

- *IO Flexibility*: ***torch*** and ***charger*** do not enforce a single IO type. Any physical block (tile) type at any (x,y) location may contain *input* and *output* blocks. If the **`perimeter`** gridlocation is not used, the default perimeter type is set to be the empty type, and the architecture designer can use explicit gridlocations to place instances of any tile types at perimeter locations.

2.7 ADDITIONAL ARCHITECTURE DESCRIPTION FILE OPTIONS

In order to simplify package distribution and user experience, torch consolidates all architecture design choices into a single XML file. Torch checks for the following settings in the architecture description file:

- *Power information*: Power information can be provided either in the architecture description file or in a separate XML file (using the '**`–tech_properties`**' option). If a separate XML file is provided, its contents override the power information in the architecture description file.

- *Routing channel width*: The number of routing channels can be provided in the **`<device>`** section of the architecture description file using the following syntax:

```
<device>
    ...
    <route_chan_width width="128"/>
</device>
```

  To override this setting on the command line, use the '**`–route_chan_width`**' option.

- *Configuration address bus widths*: Each word in the ***torch*** bitstream is addressed by the (x,y) coordinates of the tile and the word within that tile's configuration memory (wl). ***Charger*** builds corresponding address busses of the same width. These busses are sized to the minimum required widths, but may be overridden in the **`<device>`** section using the following syntax:

```
<device>
    <prog_width_overrides>
        <width_addr_x="16"/>
        <width_addr_y="16"/>
        <width_addr_wl="16"/>
    </prog_width_overrides>
    ...
</device>
```

- ***Torch*** and ***charger*** can parse the XML architecture description from a string in memory (rather than a file) by calling XmlReadArchString (rather than XmlReadArch).

2.8 UNSUPPORTED SYNTAX

Syntax that supports automatic hardware resizing is not tested and may no longer work properly.

3. CHARGER

3.1 CHARGER OVERVIEW

The Configurable Hardware Generator (***charger***) is a software platform for generating a Verilog model of a custom reconfigurable circuit from a high-level XML description. ***Charger*** generates a top level model of the configurable hardware architecture, a hierarchical model of each physical block (pb) type, and various utility models. The architecture designer must provide Verilog for all user-defined models.

3.2 USING CHARGER

The ***charger*** command line executable is '**`charger`**'. This executable should be placed in a directory on the user's execution path (see 1.1.2 for complete installation instructions). To display usage information, enter '**`charger`**' in a terminal.

```
/home/kthurmer> charger

charger: Configurable Hardware Generator
.........MIT Lincoln Laboratory.........

Usage: charger <target architecture> [Parameters...]

Optional Parameters:
        [--list_arch] List all architecture targets available with this version.
        [--arch_file] Specify full path to target xml architecture description file.
        [--route_chan_width <int>] Routing channel width (overrides width specified in the architecture description file).
        [--model_name <name>] Name of the top level Verilog module. Default: architecture name.
        [--verilog_list <name>] Name of file listing the generated verilog files. Default: <architecture name>.verilog_list.

/home/kthurmer>
```

Figure 7: Displaying *charger* Command Line Usage Information

To display the version number, enter '**`charger --version`**'.

```
/home/kthurmer> charger --version

charger: Configurable Hardware Generator
version: 2019.02.14
.........MIT Lincoln Laboratory.........

/home/kthurmer>
```

3.2.1 Required Arguments

An *architecture target* is the only required argument to ***charger***:

```
/home/kthurmer> charger example_arch
```

The architecture target name should match a target hardware architecture description located in the **`$TORCHPATH/arch/`** directory (in this case, do not use the .xml suffix). Alternatively, specify an absolute or relative path to an architecture description file (with .xml suffix) using the '**`--arch_file`**' option:

```
/home/kthurmer> charger ~/arch/example_arch2.xml --arch_file
```

3.2.2 Other *charger* Options

`--list_arch`
List all built-in architectures available in **`$TORCHPATH/arch/`**.

`--arch_file`
Interpret the target architecture argument as a file path (use the .xml extension).

`--route_chan_width <integer>`
Override the routing channel width (as specified in architecture file).

`--model_name <name>`
Name given to the top level Verilog module. The default name is the architecture name.

`--verilog_list <file name>`
Name given to the file listing the generated Verilog files. The default name is '**`<architecture_name>.verilog_list`**'.

`--version`
Display the version.

4. TORCH

4.1 TORCH OVERVIEW

The Toolkit for Reconfigurable Hardware (***torch***) is a software platform for translating an HDL application into a configuration bitstream (Figure 8), which is used to program configurable hardware. ***Torch*** is based on the open source Verilog to Routing (VTR7) flow.

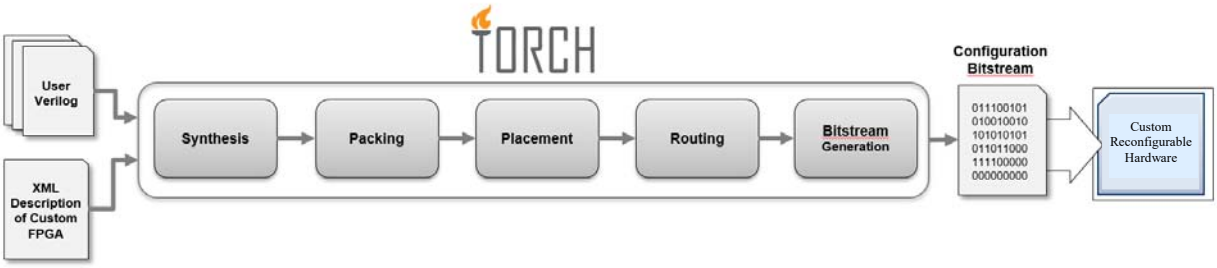

Figure 8. *Torch* Design Flow

The inputs to ***torch*** are a user Verilog application and an architecture description. The user develops a Verilog HDL model of an application, explicitly instantiating any hard IP blocks available in the target architecture. The user also specifies the target reconfigurable hardware architecture.

The output of torch is a bitstream that contains all of the information required to program the selected hardware with a user application, in the format required by the hardware.

To check the current version of torch, enter '**`torch --version`**' in a terminal.

```
/home/kthurmer> torch --version

Running torch version 2019.02.14.
```

The build phases in the torch design flow are illustrated in Figure 8 and summarized in Table 1. These build phases must be performed in the order listed (with the exception that timing analysis and power estimation may be run before bitstream generation); all build phases may be run at once, or a subset of one or more consecutive phases may be run together.

Table 1. Summary of Torch Design Flow Build Phases

Build Phase	Description	Input Files	Output Files
Synthesis	Translates complex Verilog structures into hardware primitives	• Verilog files • **Bitstream configuration (.bitcfg) file*	• Netlist (.blif) file • Activity (.act) file
Packing	Groups hardware primitives into clusters that correspond to hardware blocks	• Netlist (.blif) file	• Packed netlist (.net) file
Placement	Assigns the hardware blocks to specific locations on the hardware	• Netlist (.blif) file • Packed netlist (.net) file • **Placement constraints (.pads) file(s)* • **Disable (.disable) file*	• Placement (.place) file
Routing	Connects the placed blocks using the reconfigurable interconnect	• Netlist (.blif) file • Packed netlist (.net) file • Placement (.place) file	• Routing (.route) file
Bitstream generation	Compiles the decisions made by all previous build phases into the format required by the hardware	• Netlist (.blif) file • Packed netlist (.net) file • Placement (.place) file • Routing (.route) file	• Bitstream (.bits) file
Timing analysis	Calculates delay information and determines the circuit critical path	• Packed netlist (.net) file • Placement (.place) file • Routing (.route) file • **Timing Constraints (.sdc) file*	• Critical path (.critical_path.out) file • Net delay (.net_delay.out) file
Power estimation	Estimates power usage	• Packed netlist (.net) file • Placement (.place) file • Routing (.route) file • Activity (.act) file • **Timing Constraints (.sdc) file*	• Power (.power) file

**=optional*

4.1.1 Synthesis

Synthesis translates Verilog modules in the user application into hardware primitives available in the architecture. The output of synthesis, a BLIF *netlist* (.blif) file[2], lists the hardware primitives used in the user's applications; these primitives may include lookup tables (LUTs), as well as any other hard block available in the reconfigurable architecture (e.g. DSP blocks, memories, multipliers, etc). ***All elements except lookup tables (LUTs) should be directly instantiated in user code.*** Synthesis reports the number of each type of block used in the design.

Torch synthesis does not support the complete set of syntax defined in IEEE 1364-2005; error messages will indicate when unsupported syntax is identified. Only Verilog is supported at this time (VHDL is not currently supported). For more information on syntax limitations, see Section 5.1.

4.1.1.1 Hard IP Block Configuration

Users may specify non-default configurations for configurable hard IP block instances in their Verilog applications. Torch can generate a *bitstream configuration* file template from the user Verilog. The template file is organized according to the hierarchy of the user Verilog; it lists each configurable hard IP block instance, along with with its default configuration settings (any configurable top-level inputs and outputs will be listed at the top of the configuration template file). The user may edit this file to modify the configuration settings for some or all of the hard IP block instances. The edited template is then specified as an input to torch synthesis. After all build phases have been run, the user-specified configuration information will be embedded in the configuration bitstream and applied to the hard IP blocks when the bitstream is loaded onto the architecture. Figure 9 shows a sample bitstream configuration file.

```
input clk 1
input reset 1
input start 1
input next_k 1
output data_output_valid 1
output busy 1
input sequence_in 1
output sequence_out 1

dct_addr_gen dct_addr_gen
+--- dct_addr_multiply_wrapper dct_addr_mult
|    +--- baby_dsp_mult multiplier 010
dct_mac_wrapper dct_mac
+--- baby_dsp_mult multiplier 010
```

Figure 9. Sample Bitstream Configuration File

Torch can print a list of all configuration options for all configurable hard IP blocks available on a given architecture using the '`-list_bitcfg_opts`' flag.

[2]See http://docs.verilogtorouting.org/en/latest/vpr/file_formats for details on the BLIF format.

4.1.2 Packing

Packing clusters hardware primitives with closely related inputs and outputs. During placement (see Section 4.1.3), these clustered primitives will be assigned to logic resources within the same hardware tile and will be connected to one another using the local interconnect, thus reducing the burden of signal routing on the global interconnect. The output of the packing build phase is a *packed netlist* (.net) file.

4.1.3 Placement

Placement assigns hardware primitives and clusters of primitives (produced by packing) to specific hardware instances at specific locations on the hardware. An optional *pin placement* (.pads) file allows the user to assign physical locations to top-level ports and clusters (see Section 4.1.3.1). The output of placement is a *placement* (.place) file.

4.1.3.1 Placement Constraints: Fixed Locations

The user may specify fixed locations for top-level I/O signals and clusters by providing one or more *placement constraints* (.pads) files as an optional input to the placement build phase (using the flag '`--fix_pins <filename1> <filename2> ...`', or '`--fix_pins_list <filename>`'). This file is similar in format to the placement (.place) file; the user may list some or all of the top-level signals and clusters in the user application and specify a fixed location for each. Clusters that are not included in a placement constraints file will be automatically placed by torch during the placement phase.

Figure 10 shows an example placement constraints file specifying locations for eight multipliers in a user application. The left column lists the names of the multipliers; these names may be copied from the packed netlist (.net) file or from the placement (.place) file. The second and third columns list the x and y coordinates of the desired physical location of each multiplier. The fourth column lists the index of the desired sub-block (for locations with capacity > 1). If any member of a place macro is fixed, the entire macro will be treated as fixed.

```
#block name                                                    x    y    subblk (k)
#--------------                                                --   --   ----------
top.mult_wrapper+fir_product_real_real-0.mult+multiplier^y~0        87   49   0
top.mult_wrapper+fir_product_imag_imag-0.mult+multiplier^y~0  87    55   0
top.mult_wrapper+fir_product_real_imag-0.mult+multiplier^y~0        87   61   0
top.mult_wrapper+fir_product_imag_real-0.mult+multiplier^y~0        87   65   0
top.mult_wrapper+fir_product_real_real-1.mult+multiplier^y~0        87   71   0
top.mult_wrapper+fir_product_imag_imag-1.mult+multiplier^y~0  87    77   0
top.mult_wrapper+fir_product_real_imag-1.mult+multiplier^y~0        87   81   0
top.mult_wrapper+fir_product_imag_real-1.mult+multiplier^y~0        87   87   0
```

Figure 10. Example Placement Constraints File

The type of hardware resource at each (x,y) location may be obtained by running placement and clicking on tiles in the interactive placement graphic, as shown in Figure 11 (see Section 4.3.5 for information on displaying placement graphics).

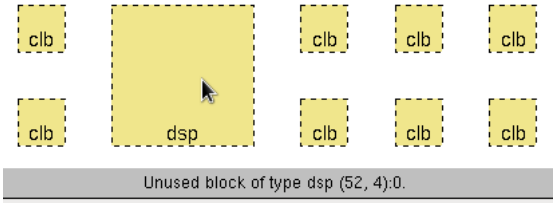

Figure 11. Finding Block Coordinates

4.1.3.2 Placement Constraints: pboxes

Torch allows the user to constrain clusters to specific regions of the hardware by specifying *placement constraint boxes* (pboxes) within a placement constraints file. The format for specifying a pbox is:

```
PBOX (Xmin,Ymin,Zmin):(Xmax,Ymax,Zmax)
<list of clusters>
!PBOX
```

Figure 12 shows a sample pbox specification within a placement constraints (.pads) file.

```
#block name                                                    x     y     subblk (k)
#--------------                                                --    --    ----------
top.mult_wrapper+fir_product_real_imag-1.mult+multiplier^y~0   87    81    0
top.mult_wrapper+fir_product_imag_real-1.mult+multiplier^y~0   87    87    0

PBOX ( 0 , 0 , 0 ) : ( 21 , 21 , 1 )
out:top^dataout_real~0
out:top^dataout_real~1
out:top^dataout_real~2
top.mult_wrapper+fir_product_real_real_0.mult+multiplier^y~0
top.mult_wrapper+fir_product_imag_imag_0.mult+multiplier^y~0
top.mult_wrapper+fir_product_real_imag_0.mult+multiplier^y~0
top.mult_wrapper+fir_product_imag_real_0.mult+multiplier^y~0
top.mult_wrapper+fir_product_real_real_1.mult+multiplier^y~0
top.mult_wrapper+fir_product_imag_imag_1.mult+multiplier^y~0
top.mult_wrapper+fir_product_real_imag_1.mult+multiplier^y~0
top.mult_wrapper+fir_product_imag_real_1.mult+multiplier^y~0
!PBOX
```

Figure 12: Example Placement Constraints File with a PBox

4.1.3.3 Disabling Grid Locations

In order to accommodate fabrication defects or other design restrictions, the user may disable (x,y) locations using a *disable* file (using the flag '`-disable_file <filename>`'). ***Torch*** will ignore logic at these locations during placement (routing resources may still be used). Figure 13 shows a sample disable file.

```
#x      y
#--     --
0       1
0       48
0       49
1       48
1       49
87      32
```

Figure 13. Example Disable File

4.1.4 Routing

Routing connects the placed blocks using routing resources. The output of routing is a *routing* (.route) file.

4.1.5 Bitstream Generation

Bitstream generation compiles the packing, placement, and routing solutions, along with user-specified hard IP block configuration information (see Section 4.1.1.1), into the format expected by the reconfigurable hardware described in the architecture description file. The output of bitstream generation is a *bitstream* (.bits) file, which may be loaded onto the configurable hardware.

4.1.6 Timing Analysis

Torch can perform timing analysis of user applications. This analysis requires that timing-based algorithms are used during the placement and routing build phases. When performing timing analysis, torch converts a placed and routed user application circuit into a graph composed of timing nodes and calculates timing information by traversing this graph. An SDC *timing constraints* file[3] may be used to constrain clocks and I/O delays. The output of torch timing analysis consists of a *critical path* (.critical_path.out) file and a *net delay* (.net_delay.out) file.

4.1.6.1 Critical Path File

The *critical path file* lists all the timing nodes in the critical path (the path with longest delay) of the mapped user circuit. Each timing node along the path is listed in sequential order along with the node's arrival time, required time, and delay value to the next node. The entire timing path, starting from the top-level input through I/O, CLB, and/or other hard IP blocks, is traversed in this file. When the path finally terminates at a flip-flop, top-level output, or hard IP block, the arrival time of the final timing node represents the critical path delay of the circuit.

[3]Only a subset of the Synopsys Design Constraint (SDC) format syntax is supported by timing analysis. See http://docs.verilogtorouting.org/en/latest/vpr/sdc_commands/ for details on the SDC format and the available commands.

The format for the critical path file is shown in Figure 14. The blocks and their names and numbers can be correlated to their corresponding instances in the packed netlist (.net) file.

```
Timing Node: <timing node number>  <TIMING NODE TYPE> Block #<block number> (<block name>)
Pin: <pb pin> pb (<block name>)
T_arr: <Arrival time>  T_req: <Required time>  Tdel: <Delay>
Internal/Routing Net: #<net number> (<net name>).  Pins on net: <#>.
```

Figure 14. The syntax for a timing node as it appears in the critical path file.

4.1.6.2 Net Delay File

The *net delay file* lists delay values calculated for each routed net in the mapped user circuit. Because the nets listed in this file directly correspond with those in the routing (.route) file, this file can be useful in determining which paths in the routing solution have the shortest or longest delays.

If there are multiple sink nodes on a single net in the routing file, a delay value is assigned to each. In this case, the delay value is the total delay time between the source node and the sink node, including any intermediate nodes. The unique net name and number allow users to relate this information to the routing file. A sample net entry in the net delay file with three different sinks is shown in Figure 15.

```
Net #         Driver_tnode  to_node        Delay
0 (n24648)    209 (256744)     0 (294868)  9.01258e-10
                             459 (270274)  6.93923e-10
                            1558 (256628)  4.87002e-10
```

Figure 15. An example net entry in the net delay file

4.1.7 Power Estimation

Torch can perform transistor-level power estimation for mapped user applications. This requires an *activity* (.act) file containing a *signal probability* and *transition density* value for each signal in the BLIF netlist.

An activity file is automatically generated during the synthesis build phase (this can be skipped with the '**-no_act_file**' option). The ***torch*** Synthesis phase is comprised of: synthesis (*Odin II*), optimization (*abc*), and activity file generation (*ace*), and outputs the following files:

```
<project name>_unopt.blif          produced by synthesis (Odin II)
<project name>_opt.blif            produced by optimization (abc)
<project name>.blif                produced by act file generation (ace)
<project name>.act                 produced by act file generation (ace)
```

The activity file generator (*ace*) slightly modifies net names in the BLIF netlist; if the user plans to run power estimation, the post-ace BLIF netlist **must** be used as the input to packing, placement, and routing so that the names match the activity file.

Alternatively, the user may specify a custom activity file (using the '**`-activity_file`**' option) containing one line for each net in the BLIF netlist file with the following format:

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

Power estimation also requires information about the circuit's clock frequency. The user may choose to run timing analysis concurrently, in which case ***torch*** will use the critical path delay to generate the maximum clock frequency. Alternatively, the user may provide an SDC file that constrains the clock frequency. Note that power values are a rough estimate only and may not correspond exactly to the actual power used by the chip.

4.2 THE TORCH API

The torch API is a single function, run_torch(t_torch_settings* torch_settings).

See $TORCHPATH/torch/include/torch.h for a description of the t_torch_settings data structure.

4.3 RUNNING TORCH ON THE COMMAND LINE

The torch command line executable is '**`torch`**'. The executable should be placed in a location on the user's execution path (see Section 1.1.2 for complete installation instructions). To display usage information, enter '**`torch`**' in a terminal (see Figure 16).

```
/home/kthurmer> torch

Running torch version 2019.02.14.

Usage: torch <project_name> <target_architecture> [Parameters...]

Parameters:
 Stage Selection Options
        [--genbitcfg] Generate bitstream configuration template.
        [--synth] Do Synthesis.
        [--pack] Do Packing.
        [--place] Do Placement.
        [--route] Do Routing.
        [--bitgen [reset] ] Generate Bitstream (Optionally all zeroes with 'reset' arg).
        [--all] Run all build phases (Synthesis, Packing, Placement, Routing, and Bitstream Generation).
 File Options
        [--arch_file] Custom target architecture (.xml format).
        [--verilog <file1.v> <file2.v> ...] List of verilog files for Synthesis.
        [--verilog_list <filename>] File containing list of verilog files for Synthesis.
        [--blif_file <filename>] Name of blif file.  Default is <project_name>.blif.
        [--net_file <filename>] Name of netlist file.  Default is <project_name>.net.
        [--place_file <filename>] Name of placement file.  Default is <project_name>.place.
        [--route_file <filename>] Name of routing file.  Default is <project_name>.route.
        [--bit_file <filename>] Name of bitstream file.  Default is <project_name>.bits.
        [--bitcfg_file <filename>] Name of bitstream configuration file.  Default (output) is <project_name>.bitcfg.
        [--fix_pins <file1.pads> <file2.pads> ...] Force placements listed in constraints file(s).
        [--fix_pins_list <filename> ...] File containing list of placement constraints file(s).
        [--disable_file <filename> ] File containing a list of (x,y) coordinates that should not be used in placement.
        [--sdc_file <filename>] Name of timing constraints file.  Default is <project_name>.sdc.
        [--log_file <filename>] Name of log file.  Default is <project_name>.log.
        [--tech_properties <filename>] Name of CMOS tech properties file.  Overrides CMOS tech info in arch file.
 Other Options
        [--timing_analysis on|off] Enable/disable timing analysis. Disabled by default.
        [--power] Enable power estimation.
        [--no_act_file] Skip activity file generation (Warning: power estimation requires an activity file).
        [--activity_file <filename>] Name of activity file for power estimation.  Default is <project_name>.act.
        [--display_route] Display existing routing solution (from <project_name>.route, or use --route_file <filename>).
        [--nodisp] Disable graphics.  Default is graphics on.
        [--list_arch] List all target architectures provided with this version.
        [--list_bitcfg_opts [ <block1_name> ... ]] List configuration bit options for each hard block.
        [--do_binary_search_route] Ignore route chan width in arch file; let the tool figure out min required routing channels.
        [--version] Display the current version of torch.

/home/kthurmer>
```

Figure 16. Displaying Torch Command Line Usage Information

To display the version number, enter '**`torch --version`**', as shown in Figure 17.

```
/home/kthurmer> torch --version

Running torch version 2018.07.13.

/home/kthurmer>
```

Figure 17. Displaying Torch Version Information

4.3.1 Required Arguments

A *project name* and *target architecture* must be specified as the first two arguments:

```
/home/kthurmer> torch framed_ladar griffin_med
```

- The project name is used to generate default names for output files. It is recommended that the project name match the name of the top-level module in the HDL design.
    - Note: the default location for output files is the current directory (the directory from which the torch command was run); the user must have write permissions for this directory.
- The architecture target name should match a target hardware architecture description located in the $TORCHPATH/arch/ directory. Alternatively, the user may specify a path to an architecture description file (.xml file) using the '***--arch_file***' option, as shown below.

```
/home/kthurmer> torch framed_ladar custom_arch.xml --arch_file
```

In addition to these two required arguments, other arguments may be required based on which torch options are invoked (e.g., when running the synthesis build phase with '**`--synth`**', the '**`--verilog`**' or '**`--verilog_list`**' options are also required). While the project name and architecture target name must be the first two arguments to torch, all other options may be entered in any order.

4.3.2 Build Phase Selection

Select one or more build phases to run using the relevant command line arguments. For example, to run synthesis, packing, and placement, add '**`--synth --pack --place`**' as shown:

```
/home/kthurmer> torch framed_ladar griffin_med --synth --pack --place
```

Note: only consecutive build phases may be run together. Below are the available build phase options:

`--synth`: run synthesis. Verilog input files must be specified (see Section 4.3.3).
`--pack`: run packing.
`--place`: run placement.
`--route`: run routing.
`--bitgen`: run bitstream generation.
`--all`: run all build phases.

4.3.3 File Selection

Output file names may be specified for each build phase; otherwise torch produces output files with default names (**`<project_name>.blif`**, **`<project_name>.net`**, etc.). Input (Verilog) files must be specified when running synthesis. For all other build phases, if input file names are not specified, Torch expects input files with default names to be present in the run directory. See Section 4.3.4 for a list of file name options.

When running synthesis, specify a list of Verilog files to be synthesized:

```
/home/kthurmer> torch framed_ladar griffin_med --synth --verilog framed_ladar_fsm.v
 ladar_macropixel_wrapper.v output_buffer.v pixel_buffer.v reimagine_ladar_top.v
```

Alternatively, specify a text file (e.g. '**`framed_ladar.txt`**') that lists the Verilog files to be synthesized:

```
/home/kthurmer> torch framed_ladar griffin_med --synth --verilog_list framed_ladar.txt
```

The above command generates a netlist (.blif) file in the run directory with the default name '**framed_ladar.blif**'. The user may optionally specify custom output file names. For example, a file name for the output netlist file may be specified using the '**--blif_file**' option as shown below:

```
/home/kthurmer> torch framed_ladar griffin_med --synth --verilog_list framed_ladar.txt
 --blif_file framed_ladar2.blif
```

In the example above, the output of synthesis is '**framed_ladar2.blif**'.

When running packing without synthesis, Torch will look for '**framed_ladar.blif**' in the run directory. Alternatively, the '**--blif_file**' option may be used to specify another .blif file as the input to packing:

```
/home/kthurmer> torch framed_ladar griffin_med --pack --place --blif_file framed_ladar2.blif
--place_file framed_ladar2.place
```

In the example above, packing reads in '**framed_ladar2.blif**' and generates '**framed_ladar.net**' (the output file is given the default name, '**<project name>.net**', since no custom packed netlist file name was specified).

Next, placement reads in '**framed_ladar2.blif**' and '**framed_ladar.net**' and produces '**framed_ladar2.place**' (the output file name is specified using the '**--place_file**' flag). An optional placement constraints file (see Section 4.1.3.1) may be specified as an input to placement using the '**--fix_pins**' flag, and an optional disable file (see Section 4.1.3.2) may be specified using the '**--disable_file**' flag, as shown below:

```
/home/kthurmer> torch framed_ladar griffin_med --pack --place --blif_file framed_ladar2.blif
--fix_pins my_fixed_pins.pads --disable_file my_disable_file
```

4.3.4 File Name Options

Below are the available file name options, which may be given in any order:

--verilog <file1.v file2.v …>
List of Verilog inputs to synthesis.

--verilog_list <filename>
File containing a list of Verilog inputs to synthesis.

--bitcfg_file <filename>
Bitstream configuration file template file generated using the **--genbitcfg** option and/or an optional input to synthesis (see Section 4.1.1.1).

--blif_file <filename>
Output generated by synthesis and/or input read by packing, placement routing, and/or bitstream generation.

--net_file <filename>
Output generated by packing and/or input read by placement, routing, and bitstream generation.

--fix_pins <filename>
OPTIONAL one or more placement constraints files may be input to placement.

--fix_pins_list <filename>
OPTIONAL file containing a list of placement constraints files may be input to placement.

`--disable_file <filename>`
OPTIONAL disable file is an input to the placement phase.

`--place_file <filename>`
Output generated by placement and/or input read by routing and bitstream generation.

`--route_file <filename>`
Output generated by routing and/or input read by bitstream generation.

`--bit_file <filename>`
Output generated by bitstream generation.

`--log_file <filename>`
Specify the name of the log file output by torch.

`--activity_file <filename>`
Specify the name of the activity file used during power analysis (required with **`--power`** option). If used without the **`--power`** flag, this specifies the output activity file generated during synthesis.

See Table 1 for a complete list of inputs and outputs for each build phase.

4.3.5 Other Options

`--list_arch`
List all architecture targets available for mapping.

`--arch_file`
Interpret the architecture target argument as a file path.

`--list_bitcfg_opts [<block_name> …]`
Display detailed information on user-configurable hard IP blocks in the architecture (see Figure 3).

`--genbitcfg`
Generate a bitstream configuration template (must specify input Verilog using **`--verilog`** or **`--verilog_list`**).

`--timing_analysis on|off`
Enable or disable timing analysis (default: disabled).

`--power`
Enable power estimation.

`--bitgen reset`
Generate an all-zero bitstream file.

`--nodisp`
Disable placement and routing graphics. If this flag is omitted, interactive graphics will pop up during placement and routing.
Note: after the interactive graphics window appears, the user must click **Proceed** to begin the build phase and **Exit** to close the interactive graphics window.

`--display_route`
Display placement and routing graphics for an already routed design. Running with this option allows users to view a placement and routing solution in the interactive graphics window without having to run any build phases.

`--version`
Display the version.

5. VERILOG APPLICATION DESIGN GUIDELINES

5.1 VERILOG SYNTAX RESTRICTIONS

Below is a list of all known partially supported and unsupported Verilog syntax.

Data Types

Integer and real variable data types are not supported.

Naming Syntax

Identifier names that begin with a '_' or contain the special character '$' are not supported.

Ports and Port Lists

- Bidirectional (inout) port declarations are not supported.
- ANSI C-style port lists are not supported.

Arrays

- Arrays are restricted to less than 4 dimensions.
- Multidimensional packed arrays are not supported, e.g.:

```
reg [3:0][1:0] my_reg      // not supported
```

- All arrays must be referenced MSB to LSB, e.g.:

```
wire [15:0] my_wire;                    // wire [0:15] my_wire is not supported
reg [7:0] my_reg [31:0];                // reg [7:0] my_reg [0:31] is not supported
assign my_wire [7:0] = my_reg [0][7:0]; // my_wire [0:7] = my_reg [0][0:7] is not supported
```

- Variable array indexing is not supported. Users are required to implement this logic explicitly. In the example below, a workaround would be to implement a multiplexer to create this indexing behavior, e.g.:

```
wire [3:0] my_array_index;
assign my_signal = my_array [my_array_index];    // not supported
```

Operators and Operations

- Variable bit shifting, division (/), exponentiation (**), and modulus (%) arithmetic operators are not supported.
- XOR and XNOR reduction operations are restricted to signals less than 4-bits wide.
- Identity operators (=== and !==) are not supported.

Continuous assignments

Implicit nets with continuous assignments are not supported (an exception to this rule is with instance port connections), e.g.:

```
assign data = data_in;     // not supported when data is an undeclared signal
```

Combinational Blocks

Implicit sensitivity lists must be contained within parentheses, e.g.:

```
always @(*)
```

Sequential Blocks

Default values in sequential blocks are not supported, e.g.:

```
always @(posedge clk) begin
    out_data <= data_default;    // not supported
    if (out_valid) begin
        out_data <= data;
           end
end
```

For-Loops

For loops are not supported outside of generate statements.

Generate Statements

- Local variables in generate for-loops are not supported.
- Case statements are not supported inside generate statements.
- If-else statements are only supported in for-loops within generate statements.

Memories

- Reads/writes to memory and virtual memory are restricted to one-dimensional values.
- Memory indices must begin at zero.

Instances

Instance arrays are not supported, e.g.:

```
shift_reg shift_reg_0[1:0] (reset, clk, data_a, data_b);    // not supported
```

Functions

Functions are not supported.

5.2 CLOCK SIGNALS IN VERILOG DESIGNS TARGETING TORCH

Torch imposes the following restrictions on the use of clock signals:

- Only single-clock designs are currently supported. Additional clocks (ref_clk, lvds_clk) are allowed in HDL design but do not participate in place and route or timing analysis, and may only be used to clock hard IP blocks.
- In each always-block sensitivity list, clock signals are expected not to drive any circuitry in the always-block, while every other signal that appears in this list must appear somewhere in the block. For example, this would be valid syntax, since the CLK signal doesn't appear anywhere in the always-block while the RST signal does:

```
always @(posedge CLK, negedge RST) begin
    if (~RST) X <= 0;
    else X <= Y;
end
```

- All signals in an always-block sensitivity list must be edge-triggered ('**posedge**' or '**negedge**') in order for the block to be considered sequential.
- Clock signals may not be used in any logic throughout a design. They must only appear in sensitivity lists and as port connections to clock ports of other module instances or hard block instances.