TAU 2015 Contest
Incremental Timing Analysis and
Incremental Common Path Pessimism Removal (CPPR)

Contest File formats

v1.15 – February 9^{th}, 2015

https://sites.google.com/site/taucontest2015

Contents

1 Introduction

This document explains the input and output file formats used in the TAU 2015 Contest. For further details regarding binary input syntax, deliverable instructions, and other usage details, please refer to `contest_rules.pdf`. For reference within this document, the binary syntax is:

`%>myTimer <.tau2015> <.timing> <.ops> <output file>`

2 Input Files

The TAU 2015 Contest will use a subset of keywords within the standard *Verilog* (**.v**), *(***S***tandard* **P***arasitic* **E***xchange* **F***ormat* (**.spef**), and *Liberty* (**.lib**) syntaxes to describe a benchmark, along with a *timing file* (**.timing**) from the TAU 2014 Contest. The **.v** file describes the netlist and structure of the design. The **.spef** file describes the parasitics (e.g., *RC*) of the nets of the design. The **.lib** describes the set of gates or cells used in the design. These files will be encapsulated in a wrapper file for streamlined usage. The **.timing** file describes the initial operating conditions of the design, such as the arrival times of the primary inputs. The **.ops** file describes the set of operations that modifies the design and queries timing information.

2.1 .tau2015 (wrapper)

This file is the main benchmark wrapper file from the input

`%>myTimer <.tau2015> <.timing> <.ops> <output file>`

A benchmark is comprised of three groups of files: (*i*) **.v** (Section 2.2), (*ii*) **.spef** (Section 2.3) and (*iii*) two **.lib** (Section 2.4). The **.tau2015** file contains the list or location of these four files needed for benchmark parsing. The ordering of the files is fixed. In order, they will appear as <**Early**>**.lib**, <**Late**>**.lib**, **.spef** and **.v**. For example, `simple.tau2015` would contain:

```
simple_Early.lib simple_Late.lib simple.spef simple.v
```

The wrapper file is used as the *only* input file when loading a benchmark. Each contained file name will have the same directory path location as the `.tau2015` file. For example, if the call to load the `simple.tau2015` file is

%> myTimer ../simple/simple.tau2015 ...

then the four files in `simple.tau2015` are expected to be in the ../simple/ directory, e.g.,

```
../simple/simple_Early.lib
../simple/simple_Late.lib
../simple/simple.spef
../simple/simple.v
```

The **.timing** file (Section 2.5), which controls the initial conditions of the benchmark, is not part of the wrapper file.

2.2 .v (Verilog)

The Verilog file specifies the top-level hierarchy of the design. For this contest, we will be using a small set of keywords with the Verilog language. If you are implementing your own Verilog parser, you will be expected to support the set of keywords (in **bold**) found within the `simple.v` file (reproduced below for clarity). You will also be expected to support comments that start with '//'. The expected syntax is:

```
module <circuit name> (
  <input 1>,
  ...,
  <input n>,
  <output 1>,
  ...
  <output m> );

  input <input 1>;
  ...
  input <input n>;
  output <output 1>;
  ...
  output <output m>;

  // begin wire definitions
  wire <wire 1> ;
  //  end  wire definitions

  // begin cell definitions
  <cell type> <cell instance name> ( .<pin name> (<net name) );
  //  end  cell definitions
end module
```

The expected structure of the Verilog file is to start with a module declaration, defining the interface of the module with name <`circuit name`>. The inputs and output pins are explicitly declared; the internal wires are optionally declared with the keyword `wire`. For each cell definition, every <`cell type`> (.<`pin name`>) should be a specified cell type (pin) in the library file (e.g., `simple.lib`), and every <`cell instance name`> and <`net name`> should be found in the design specification (e.g., `simple.v`). Each field is considered a string. The following example is from `simple.v`; its corresponding implementation is shown in Figure 1.

```
01. module simple (
02. inp1,
03. inp2,
04. tau2015_clk,
05. out
06. );
07.
08. // Start PIs
09. input inp1;
10. input inp2;
11. input tau2015_clk;
12.
13. // Start POs
14. output out;
15.
16. // Start wires
17. wire n1;
18. wire n2;
19. wire n3;
20. wire n4;
21. wire inp1;
22. wire inp2;
23. wire tau2015_clk;
24. wire out;
25.
26. // Start cells
27. NAND2_X1 u1 ( .a(inp1), .b(inp2), .o(n1) );
28. DFF_X80 f1 ( .d(n2), .ck(tau2015_clk), .q(n3) );
29. INV_X1 u2 ( .a(n3), .o(n4) );
30. INV_X2 u3 ( .a(n4), .o(out) );
31. NOR2_X1 u4 ( .a(n1), .b(n3), .o(n2) );
32.
33. endmodule
```

Lines 01 and 33 define the start and end of the specified design with the keywords `module` and `endmodule`. Lines 01-06 specify the input and output connection names of the module (note that the direction is not specified here). Lines 09-11 specify the primary inputs (PIs) of the module with the keyword `input`. These names must match the ones stated with `module` (lines 01-06). Line 14 specifies the primary output (PO) of the module with the keyword `output`. This name must match the one stated with `module` (lines 01-06). Lines 17-24 specify the connections or nets within the module with the keyword `wire`. These connections specify both the external PIs and POs as well as the internal connections between gates (explained further

after lines 27-31). Lines 27-31 specify the cells used in the design, as well as how the cells are connected. For example, on line 27, a `NAND2_X1`-type cell instance of `u1` is specified, its `a` pin is fed by primary input `inp1`, its `b` pin is fed by primary input `inp2`, and its `o` pin feeds the internal net `n1`. On line 31, `n1` feeds the `a` pin of the `NOR2_X1`-type cell instance `u4`. Line 33 terminates the module definition.

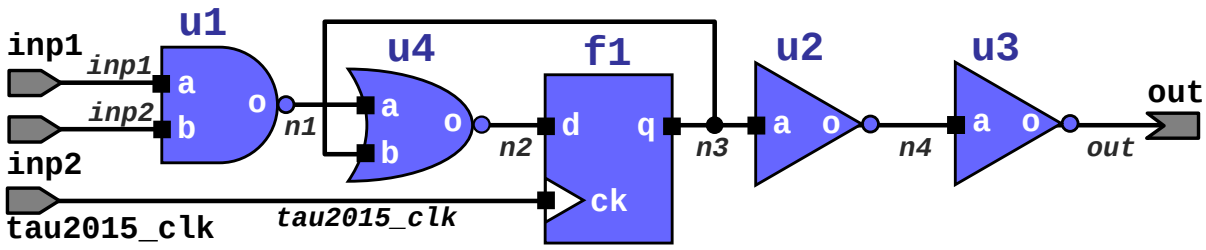

Figure 1: Implementation of `simple.v`.

2.3 .spef (SPEF)

This file contains the parasitics of a set of nets as a resistive-capacitive (RC) network. If a (e.g., gate-to-gate) connection does not have parasitics, then that the connection has 0 delay and the output slew is equivalent to the input slew. If you plan on writing your own **.spef** parser, you will be expected to support the format specified in `simple.spef` (portions reproduced for clarity). You will also be expected to support comments beginning with '//'. The format is:

```
// begin header
*SPEF <string>
*DESIGN <string>
*DATE <string>
*VENDOR <string>
*PROGRAM <string>
*VERSION <string>
*DESIGN_FLOW <string>
*DIVIDER <string>
*DELIMITER <string>
*BUS_DELIMITER <string>
*T_UNIT <int> <string>
*C_UNIT <int> <string>
*R_UNIT <int> <string>
*L_UNIT <int> <string>
//  end  header
// begin nets
// ...
//  end  nets
```

The header describes the general set of units for the file. In this contest, the `DELIMITER` field will be set to ':', the `T_UNIT` field will be set to one picosecond (`1 PS`), the `C_UNIT` field will be set to one femptoFarad (`1 FF`), and the `R_UNIT` field will be set to one kiloOhm (`1 KOHM`). All other fields in the header will not be used. Below shows an example header.

```
01. *SPEF "IEEE 1481-1998"
02. *DESIGN "simple"
03. *DATE "Tue Sep 30 11:51:50 2014"
04. *VENDOR "TAU 2015 Contest"
05. *PROGRAM "TAU 2015 Contest Generator"
06. *VERSION "0.0"
07. *DESIGN_FLOW "NETLIST_TYPE_VERILOG"
08. *DIVIDER /
09. *DELIMITER :
10. *BUS_DELIMITER [ ]
11. *T_UNIT 1 PS
12. *C_UNIT 1 FF
13. *R_UNIT 1 KOHM
14. *L_UNIT 1 UH
```

Line 01 specifies the SPEF format date. Line 02 specifies the design name. Line 03 specifies the date at which the file was generated. Line 04 specifies the consumer of this file. Line 05 specifies the tool used to generate the file. Line 06 specifies the version of this file. Line 07 specifies the format in which this file is used. Line 08 specifies the hierarchy divider character. Line 09 specifies the pin divider character. Line 10 specifies the bus delimiter characters. Line 11 specifies the time units for the design. Line 12 specifies the capacitance units for the design. Line 13 specifies the resistance units for the design. Line 14 specifies the inductance units for the design. After the header, each net's parasitics will be defined by the following format:

```
*D_NET <net name> <total net capacitance>
*CONN
<pin type> <pin name> <pin direction>
// more pin definitions
*CAP
<integer label> <pin or node name> <pin or node capacitance>
// more capacitor definitions
*RES
<integer label> <pin or node name> <pin or node name> <pin or node resistance>
// more resistor definitions
*END
```

Each net's definition begins with the keyword *D_NET followed by its name and the sum of all the capacitors of the net. The <net name> will be unique for each net, and each net referenced in the **.ops** file will reference nets by these names. The <total net capacitance> will be a decimal value, and is the sum of all the capacitors defined in the *CAP section. The *CONN keyword describes the set of pins attached to the net. The <pin type> field will either be of type port (*P), which is a primary input or output pin, or internal (*I), which is an internal pin in the design. In this section, only design pins will be referenced – no intermediate SPEF-specific node will be listed. The <pin name> field will be either a primary input, a primary output, have the syntax <cell name>:<cell pin name>, e.g., NAND2_X2:A, or have the syntax <net name>:<int>, e.g., n1:1. The <pin direction> field refers to the *pin* directional type (not the net), and will be either input (I) or output (O).

The *CAP keyword describes the set of *grounded* capacitors that are in the net. Namely, each capacitor will be connected to a specified node and GND. For each capacitor, the <integer label> is a unique integer that identifies the capacitor *for this net*. The <pin or node name> is a string, and can be a primary input, primary output, a design pin with the syntax <cell name>:<cell pin name>, or an intermediate SPEF-specific node with the syntax <net name>:<integer>. The <pin or node capacitance> will be a decimal value specifying the capacitance attached to the node. The actual capacitance will be this value multiplied by the C_UNIT value specified in the header. For example, if C_UNIT is 1 FF and <pin or node capacitance> is 1.2, then the capacitance is 1.2 fF.

The *RES keyword describes the set of resistors in the net. Each resistor connects two pins or nodes (whose format is identical to the *CAP field), and similarly has a unique <integer label>. The <pin or node resistance> is a decimal value; the actual resistance value is this field multiplied by the R_UNIT value specified in the header. For example, if R_UNIT is 1 KOHM and <pin or node resistance> is 3.4, then the resistance is 3.4 $K\Omega$. The *END keyword indicates the end of the net parasitics. An example net definition is shown below:

```
01. *D_NET inp2 2.0
02. *CONN
03. *P inp2 I
04. *I u1:b I
05. *CAP
06. 1 inp2 0.2
07. 2 inp2:1 0.5
08. 3 inp2:2 0.4
09. 4 u1:b 0.9
10. *RES
11. 1 inp2 inp2:1 1.4
12. 2 inp2:1 inp2:2 1.5
13. 3 inp2:2 u1:b 1.6
14. *END
```

Let `*R_UNIT` and `*C_UNIT` be the same values as in the header above, i.e., `*R_UNIT` is `1 KOHM` and `*C_UNIT` is `1 FF`. Line 01 defines the net `inp2` with a total lumped capacitance of 2 fF. Lines 02-04 defines the connectivity of the net `inp2`. Line 03 specifies the primary (or port) pin `inp2` is an input type. Line 04 specifies the internal design pin `u1:b` is an input type. Lines 05-09 defines the set of capacitors for net `inp2`. Line 06 defines capacitor `1` that is between primary input pin `inp2` and GND with a value of 0.2 fF. Line 07 specifies capacitor `2` between the SPEF-specific intermediate node `inp2:1` and GND with a value of 0.5 fF. Line 08 specifies capacitor `3` that is connected between the SPEF-specific intermediate node `inp2:2` and GND with a value of 0.4 fF. Line 09 specifies capacitor `4` that is between the design pin `u1:b` and GND with a value of 0.9 fF. Lines 10-13 defines the set of resistors of net `inp2`. Line 11 specifies resistor `1` between primary input pin `inp2` and the SPEF-specific intermediate node `inp2:1` with a value of 1.4 $K\Omega$. Line 12 specifies resistor `2` between the SPEF-specific intermediates nodes `inp2:1` and `inp2:2` with a value of 1.5 $K\Omega$. Line 13 specifies resistor `3` between the SPEF-specific node `inp2:2` and design pin `u1:b` with a value of 1.6 $K\Omega$. Line 14 ends the net definition. Figure 2 illustrates the parasitics described above for net `inp2`.

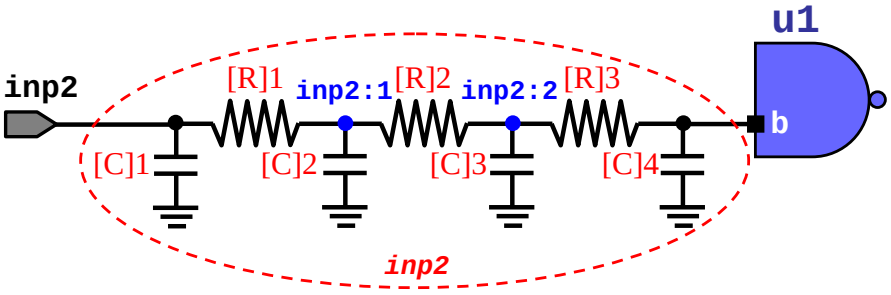

Figure 2: Parasitics of net `inp2`. The `[R]` (`[C]`) labels refer to the resistors (capacitors).

2.4 .lib (Liberty)

This file contains the set of all cells or gates that are available to the design. All cell instances found in the **.v** file will have corresponding cell type that is located in this file. Gate-level delay and output slew calculations will use the relevant timing information found for the appropriate cell type. For this contest, we will be using a limited set of keywords in the **.lib** syntax. If you are writing your **.lib** parser, you will be expected to support all keywords found `simple.lib`. You will also be expected to support comments beginning with '/*' and ending with '*/'. For this contest, you can expect that the early lib will contain all the early delays and slews, and the hold guard times of the design; the late lib will contain all the late delays and slews, and the setup guard times of the design.

The relevant portions of the **.lib** file are explained below. The library consists of (*i*) a header, (*ii*) a set of lookup-table definitions, and (*iii*) a set of cell definitions, where a cell will be a combinational element (e.g., NAND2) or a sequential element (e.g., flip-flop DFF). While there are many keywords available, this contest will only use the following set. If other keywords are present, your parser is expected to ignore them. For readability, each syntax set is discussed in separate subsections below.

```
library (<library name>) {
  /* begin header */
  /*      ...     */
  /*  end header  */
  /* begin lookup table definitions */
  /*               ...               */
  /*  end  lookup table definitions */
  /* begin cell definitions */
  /*          ...           */
  /*  end  cell definitions */
}
```

Header. The header sets the general information about the library, and is defined with the following format:

```
  delay_model : table_lookup ;
  time unit    : <time unit> ;
  voltage_unit : <voltage unit> ;
  current_unit : <current unit> ;
  capacitive_load_unit(<int>, <capacitance unit>);
  leakage_power_unit     : <power unit> ;
  pulling_resistive_unit : <resistance unit> ;
  input_threshold_pct_rise : <int> ;
  input_threshold_pct_fall : <int> ;
  output_threshold_pct_rise : <int> ;
  output_threshold_pct_fall : <int> ;
  slew_upper_threshold_pct_rise : <int> ;
  slew_lower_threshold_pct_rise : <int> ;
  slew_upper_threshold_pct_fall : <int> ;
  slew_lower_threshold_pct_fall : <int> ;
  nom_process     :  <double> ;
  nom_temperature :  <double> ;
  nom_voltage     :  <double> ;
  operating_conditions(<instance name>) {
    process     :  <double> ;
    temperature :  <double> ;
    voltage     :  <double> ;
  }
  default_operating_conditions : <instance name> ;
```

The <unit> fields and <instance name> are considered strings, and may or may be enclosed in ’“’ and ‘”’. A sample header section looks like:

```
01. delay_model : table_lookup ;
02. time unit    : "1ps" ;
03. voltage_unit : "1V" ;
04. current_unit : "1mA" ;
05. capacitive_load_unit(1, ff);
06. leakage_power_unit     : "1uW" ;
07. pulling_resistive_unit : "1kohm" ;
08. input_threshold_pct_rise : 50 ;
09. input_threshold_pct_fall : 50 ;
10. output_threshold_pct_rise : 50 ;
11. output_threshold_pct_fall : 50 ;
12. slew_upper_threshold_pct_rise : 90 ;
13. slew_lower_threshold_pct_rise : 10 ;
14. slew_upper_threshold_pct_fall : 90 ;
15. slew_lower_threshold_pct_fall : 10 ;
16. nom_process     :  1.0 ;
17. nom_temperature :  85.0 ;
18. nom_voltage     :  0.600 ;
19. operating_conditions(typical) {
20.   process     :  1.0 ;
21.   temperature :  85.0 ;
22.   voltage     :  0.600 ;
23. }
24. default_operating_conditions : typical ;
```

Line 01 specifies the delay model used. Lines 02-07 specify the units in which the values in the **.lib** file are referenced. Lines 08-11 specify the reference voltage point at which the slew is measured. Lines 12-15 specify the reference upper and lower bounds at which the slew is measured. Lines 16-18 specify the nominal process, temperature, and voltage at which the library is characterized at. Lines 19-23 specify a set of operating conditions for the "typical" profile. Line 24 sets the default operating conditions of the library.

Lookup tables. The lookup table templates are defined as follows:

```
lu_table_template (<table label>) {
  variable_1 : <variable name> ;
  index_1 (<string of data points for variable_1>);
  variable_2 : <variable name> ;
  index_2 (<string of data points for variable_2>);
  ...
}
```

The <table label > and <variable name> fields are considered to be strings, and may or may not be enclosed in "" and "". The string of data points will be a set of integer or double values indicating the index values of the table. The variable and index definition lines can be in any order, e.g., all variable definitions can come before all index definitions. Each <table label> can be referenced in the cell definitions. An example combinational table template looks like:

```
01. lu_table_template (delay_slew_load_5x1) {
02.   variable_1 : input_net_transition;
03.   index_1 ("1, 2, 3, 4, 5");
04.   variable_2 : total_output_net_capacitance;
05.   index_2 ("1");
06. }
```

Lines 01 and 06 define the table template with label "delay_slew_load_5x1". Line 02 defines variable_1 to be input_net_transition. In this example, this is considered to be the input slew. Line 03 defines that this variable has five sampled data indices. Line 04 defines variable_2 to be output_net_capacitance. In this example, this is considered to be the output load. Line 05 defines that this variable has only one sample data index. This lookup table has size 5x1, e.g., a single vertical column of values.

An example sequential table template that is used for setup and hold tests looks like:

```
01. lu_table_template (hold_slew_slew_5x4) {
02.   variable_1 : constrained_pin_transition;
03.   index_1 ("1, 2, 3, 4, 5");
04.   variable_2 : related_pin_transition;
05.   index_2 ("1, 2, 3, 4");
06. }
```

Lines 01 and 06 define the table template with label "hold_slew_slew_5x4". Line 02 defines variable_1 to be constrained_pin_transition. In this example, this is considered to be the input slew of the *data*. Line 03 defines that this variable has five sampled data-side indices. Line 04 defines variable_2 to be related_pin_transition. In this example, this is considered to be the input slew of the clock. Line 05 defines that this variable has four sampled clock-side indices. This lookup table has size 5x1, e.g., a single vertical column of values.

An example of an instance of a lookup-table for "delay_slew_load_5x1" is shown below:

```
01. index_1 ("1.05, 2.00, 5.00, 5.50, 9.00");
02. index_2 ("1.00");
03. values ( \
04.   "2.10", \
05.   "4.00", \
06.   "8.00", \
07.   "11.00", \
08.   "18.00", \
09. );
```

Line 01 specifies the <`variable 1`> indices. Line 02 specifies the <`variable 2`> index. Lines 03-08 specify the values that correspond to the indices. For instance, at indices $(2.00, 1.00)$, the corresponding value is 4.00.

Cell definitions. A cell specifies a gate that could be used as part of a design, e.g., combinational gate `NAND2` and flip-flop `DFF`. Its relevant specified syntax in the **.lib** format is:

```
cell (<cell type>) {
  pin(<pin name>) {
    clock           : <Boolean> ;
    direction       : <direction> ;
    capacitance     : <double> ;
    max_capacitance : <double> ;
    min_capacitance : <double> ;
    timing() {
      related_pin     : <pin name> ;
      /* combinational or sequential definitions */
    }
    /* other timing() definitions */
  }
  /* other pin definitions */
}
```

In a cell, multiple pins can be defined, e.g., a standard NAND2 will have 3 pins – two inputs and one output. For each pin, the `direction` field indicates the type of pin: (*i*) `input`, (*ii*) `output`, or (*iii*) `internal`. The `capacitance`, `max_capacitance`, and `min_capacitance` fields specify the respective pin capacitance, maximum and minimum expected pin loads. In the case where the pin a clock input of a sequential element, the `clock` field will be present on that pin with **true**. By default, if the `clock` keyword is not present, the pin is not considered a clock pin. A `timing()` definition creates a *timing arc* (directed pin-to-pin) inside a cell. The specific syntax is different for a combinational and sequential connection (discussed below).

Combinational timing arcs. Combinational arcs propagate delay and output slew from a source pin to a sink pin. They are found in common combinational logic gates, e.g., `NAND2` or as a clock-trigger segment in flip-flops. A propagate segment's `timing()` syntax is:

```
timing() {
  related_pin  : <pin name> ;
  timing_sense : <timing sense> ;
  timing_type  : <timing type> ;
  cell_<transition> (<table label>) {
    <table instance> /* omitted for space */
  }
  <transition>_transition(<table label>) {
  <table instance> /* omitted for space */
  }
  /* other cell transition table definitions */
}
```

The `related_pin` is the *source* of the segment, and the `pin` (from the pin definition) is the *sink* of the segment. The `timing_sense` field specifies the transition mode: (*i*) `positive_unate`, where the source and sink transitions are the same (e.g., rise-to-rise), (*ii*) `negative_unate`, where the source and sink transitions are opposite (e.g., rise-to-fall), and (*iii*) `non_unate`, where the source transition has no relation to the sink transition. The `timing_type` field specifies if the arc is `combinational`, where the unateness is be defined as either `positive_unate` or `negative_unate`, or <`timing type edge`>`_edge`, where the unateness is defined as `non_unate` and <`timing_type_edge`> is either `rising` or `falling`, and refers to the source.

The `cell_`<`transition`> table refers to *delay*; the <`transition`>`_transition` table refers to *output slew*. In both tables, the <`transition`> refers to the *sink* of the arc, and is either `rise` or `fall`. Note that in the case of (*i*) `positive_unate` and (*ii*) `negative_unate`, the direction of the source-to-sink transition is implicitly defined by knowing the unateness and the <`transition`>`_transition`. For instance, if the arc is `negative_unate` and there exists a table with `fall_transition`, the arc described is a rise-to-fall transition. In the case of `non_unate`, both <`timing_sense`> and <`transition`>`_transition` must be used, where the former describes the source edge, and the latter describes the sink edge. For example, if <`timing_sense`> is `rising_edge` and there exists a table with `fall_transition`, the arc described is a rise-to-fall transition. The <`table label`> will be a string that corresponds either (*i*) to a previously-declared lookup-table template or (*ii*) be the keyword `scalar`, indicating that the value stored is a single element (i.e., a 1x1 table). A sample gate is shown below:

```
01. cell(OR2_X2) {
02.   pin ("o") {
03.     direction : output ;
04.     capacitance : 2.00 ;
05.     timing() {
06.       related_pin  : "a";
07.       timing_sense : positive_unate;
08.       timing_type  : combinational;
09.       cell_fall (scalar) {
10.         values ("40.00");
11.       }
12.       fall_transition (delay_slew_load_6x1) {
13.         index_1 ("1.050, 2.000, 5.000, 5.500, 9.000, 20.00");
14.         index_2 ("1.0000");
15.         values ( \
16.           "1.050000", \
17.           "2.000000", \
18.           "5.000000", \
19.           "5.500000", \
20.           "9.000000", \
21.           "20.000000" \
22.         );
23.       }
24.     }
25.   }
26. }
```

Lines 01-26 define the cell `OR2_X2`. Lines 02-25 define the pin `o` inside cell `OR_X2`. Line 03 specifies that `o` is an output pin. Line 04 specifies that the pin capacitance of the cell (for both rise and fall) is $2fF$. Lines 05-24 specify a timing arc between source pin `a` (line 06) and sink pin `o`. Line 07 specifies that this timing arc is of type `positive_unate`, which propagates the incoming transition to the output transition (i.e., rise-to-rise and fall-to-fall). Lines 09-11 specify that the arc contains a fall transition at the output with a fixed (scalar) delay value of $40ps$. Due to the `cell_fall` definition and the `positive_unate` type, this arc is implicitly a fall-to-fall transition. Lines 12-23 specify the output slew table using lookup-table template `delay_slew_load_6x1`, with lines 13-22 matching the corresponding table syntax.

Sequential timing checks. These timing tests or checks ensure that timing is consistent between two pins. For example, a *hold test* ensures that the data signal is held long enough after the clock launches, and a *setup test* ensures that the data signal arrives safely before the clock signal. They are found between the data pin and clock pin in flip-flops. A test's `timing()` syntax is:

```
timing() {
  related_pin  : <pin name> ;
  timing_type  : <timing type>;
  <transition>_constraint (<table label>) {
    <table instance> /* omitted for space */
  }
  /* other transition table definitions */
}
```

The `related_pin` is the *clock* side of the test, and the `pin` (from the pin definition) is the *data* side of the segment. The `timing_type` field specifies the type of test and its corresponding *clock-side transition*: (*i*) `hold_rising`, where a hold test is checking against the clock-rise transition, (*ii*) `hold_falling`, where a hold test is checking against the clock-fall transition, (*iii*) `setup_rising`, where a setup test is checking against the clock-rise transition, and (*iv*) `setup_falling`, where a setup test is checking against the clock-fall transition. The <`transition`>`_constraint` tables refer to the *test time* (hold or setup time) tables of the test. The <`transition`> refers to the *data* side of the test, and is either `rise` or `fall`. The <`table label`> will be a string that corresponds either (*i*) to a previously-declared lookup-table template or (*ii*) be the keyword `scalar`, indicating that the value stored is a single element (i.e., a 1x1 table). A sample cell is shown below:

```
01. cell(DFF_X80) {
02.   pin(d) {
03.     direction : input ;
04.     capacitance : 2.00 ;
05.     timing() {
06.       related_pin : "ck";
07.       timing_type : hold_rising;
08.       fall_constraint (scalar) {
09.         values ("44.3");
10.       }
11.       rise_constraint (scalar) {
12.         values ("29.9");
13.       }
14.     }
15.   }
16. }
```

Lines 01-16 define a flip-flop with type `DFF_X80`. Lines 02-15 define the `d` pin in the `DFF_X80` cell. Line 03 specifies that `d` is an input pin. Line 04 specifies its pin capacitance (both rise and fall) is $2fF$. Lines 05-14 specify a hold test between the data-side `d` pin and the clock-side `ck`. Line 07 specifies the check is against the rise transition of `ck`. Lines 08-10 specify the required hold time between the clock-rise transition and the data-fall transition. Lines 11-13 specify the required hold time between the clock-rise transition and the data-rise transition.

2.5 .timing (Timing)

This file contains the relevant timing information needed to propagate timing from the input:

%>myTimer <.tau2015> <.timing> <.ops> <output file>

This format follows the TAU 2014 Contest timing file format, but will also include required arrival times. You will be expected to support all the keywords found in `simple.timing` (reproduced below for clarity).

```
clock <PI> <clock period> <low %>
at <PI> <early rise at> <early fall at> <late rise at> <late fall at>
slew <PI> <early rise slew> <early fall slew> <late rise slew> <late fall slew>
rat <PO> <early rise rat> <early fall rat> <late rise rat> <late fall rat>
load <PO> <early and late capacitance>
```

The `clock` keyword denotes the single clock signal <PI> of the design. The <`clock period`> is a decimal specifying the total time of one rise and fall transition of the clock signal. The <`low %`> is decimal specifying the percentage of <`clock period`> where the clock signal is low. By default, the clock signal starts low and rises at the appropriate time. For example, if <`clock period`> is 10 and <`low %`> is 30, then the clock signal rises at at 3. The `at` keyword denotes the arrival time at a primary input <PI>. The <`early rise at`> field specifies the early rise arrival time of <PI>. The <`early fall at`> field specifies the early fall arrival time of <PI>. The <`late rise at`> field specifies the late rise arrival time of <PI>. The <`late fall at`> field specifies the late fall arrival time of <PI>. The `slew` keyword denotes the input slew at <PI>. The <`early rise slew`> field specifies the early rise slew of <PI>. The <`early fall slew`> field specifies the early fall slew of <PI>. The <`late rise slew`> field specifies the late rise slew of <PI>. The <`late fall slew`> field specifies the late fall slew of <PI>. The `rat` keyword denotes the required arrival time at <PO>. The <`early rise rat`> field specifies the early rise required arrival time of <PO>. The <`early fall rat`> field specifies the early fall required arrival time of <PO>. The <`late rise rat`> field specifies the late rise required arrival time of <PO>. The <`late fall rat`> field specifies the late fall required arrival time of <PO>. All clock periods, arrival times, and required arrival times will be specified in picoseconds (ps). The <`load`> field specifies the early and late pin capacitance assertion of <PO>, which is specified in fF. Below shows an example for `simple.timing`.

```
01. clock tau2015_clk 50 50
02. at inp1 0 0 5 5
03. at inp2 0 0 1 1
04. at tau2015_clk 0 0 0 0
05. slew inp1 0 0 1 1
06. slew inp2 0 0 1 1
07. slew tau2015_clk 0 0 1 1
08. rat out 10 10 20 20
09. load out 4
```

Line 01 specifies the clock signal `tau2015_clk` with a clock period of 50 *ps* and a duty cycle of 50% (i.e., the signal rises at 25). Lines 02-04 specify the respective early rise, early fall, late rise and late fall arrival times of primary inputs `inp1`, `inp2`, and `tau2015_clk`. Lines 05-07 specify the respective early rise, early fall, late rise and late fall input slews of primary inputs `inp1`, `inp2`, and `tau2015_clk`. Lines 08 specifies the respective early rise, early fall, late rise and late fall required arrival time of primary output `out`. Line 09 specifies the output load asserted at the primary output `out`.

2.6 Command and Operations (.ops)

This file contains the set of operations or commands that are to be supported in the contest that is part of the input:

`%>myTimer <.tau2015> <.timing> <.ops> <output file>`

You are expected to process this file one line at a time, e.g., you *should not* change the given order of operations. In the following, the first set is a set of commands that modify the circuit's topology and timing profile, while the second set is a set of commands that query timing information. The former set is not expected to produce output, but are required to keep the design consistent.

Circuit modification commands. This set of commands have the ability to change a design's topology at the (*i*) gate-level, (*ii*) net-level, and (*iii*) pin-level. No output is expected.
Gate-level:

- `insert_gate <new cell instance name> <new cell type>`: creates a new gate in the design. This newly-created cell is *not* connected to any other gates or wires. The new gate name is guaranteed not to conflict with any existing names in the current design.
- `repower_gate <cell instance name> <new cell type>`: changes the size or level of an existing gate, e.g., `NAND2_X2` to `NAND2_X3`. The gate's logic function and topology is guaranteed to be the same, along with the currently-connected nets. However, the pin capacitances of the new cell type could be different.

- `remove_gate <cell instance name>`: removes a gate from the design. This is guaranteed to be called after the gate has been disconnected from the design (see pin-level commands below).

Net-level:

- `insert_net <net name>`: creates an empty net object with the unique identifier `<net name>`. By default, it will not be connected to any pins, and have no parasitics (**.spef**). This net will be connected to existing pins in the design by the command `connect_pin`, and parasitics will be loaded by the command `read_spef`.

- `read_spef <.spef>`: reads in a **.spef** file which includes a set of net parasitics. If the spef file contains a net that already has parasitics in the design, it should be overwritten. Any **.spef** files will be located in the same directory as the wrapper (**.tau2015**).

- `remove_net <net name>`: removes a net from the design. By default, if a net is connected to pins, the pins should be automatically disconnected from the net. The corresponding parasitics should also be removed.

Pin-level:

- `connect_pin <pin name> <net name>`: connects the pin to the corresponding net. The `<pin name>` will either have the `<cell name>:<cell pin name>` syntax (e.g., `u4:ZN`), or be a primary input (e.g., `inp1`). The `<net name>` will match an existing net read in from a **.spef** file.

- `disconnect_pin <pin name>`: disconnects the pin from the net it is connected to. The `<pin name>` will either have the `<cell name>:<cell pin name>` syntax (e.g., `u4:ZN`) or be a primary input (e.g., `inp1`).

Timing queries. This set of commands probe the design to report timing information.

- `report_at -pin <pin name> [-rise|-fall] [-early|-late]`: reports the arrival time in picoseconds (*ps*) at <pin name>, where the pin will be in the current design, i.e., no internal spef nodes. Options `-rise` and `-fall` are mutually exclusive, and respectively specify the desired transition. Options `-early` and `-late` are mutually exclusive, and respectively specify the desired mode. By default, the `-early -rise` options are selected. In this command, the `-pin` switch is required. All other switches are optional. The output is further discussed in Section 3.1.

- `report_rat -pin <pin name> [-rise|-fall] [-early|-late]`: reports the required arrival time in picoseconds (*ps*) at <pin name>, where the pin will be in the current design, i.e., no internal spef nodes. Options `-rise` and `-fall` are mutually exclusive, and respectively specify the desired transition. Options `-early` and `-late` are mutually exclusive, and respectively specify the desired mode. By default, the `-early -rise` options are selected. In this command, the `-pin` switch is required. All other switches are optional. The output is further discussed in Section 3.2.

- `report_slack -pin <pin name> [-rise|-fall] [-early|-late]`: reports the post-CPPR slack in picoseconds (*ps*) at <pin name>, where the pin will be in the current design, i.e., no internal spef nodes. Options `-rise` and `-fall` are mutually exclusive, and respectively specify the desired transition. Options `-early` and `-late` are mutually exclusive, and respectively specify the desired mode. By default, the `-early -rise` options are selected. In this command, the `-pin` switch is required. All other switches are optional. The output is further discussed in Section 3.3.
- `report_worst_paths [-pin <pin_name>] [-numPaths <int>]`: reports the `<int>` paths (e.g., from primary input to primary output) with the worst *non-positive* post-CPPR slack(s). If `-pin` is specified, this command should specify the worst `<int>` paths through `<pin_name>`, where the pin can be any data timing pin in the design. By default `-numPaths` is set to 1. The output is further discussed in Section 3.4. In this command, all switches are optional.

Each command will be on its own separate line. An example `simple.ops` file looks like:

```
report_slack -pin inp1
report_at -pin out
report_slack -pin u3:a -late -rise
repower_gate u3 INV_X3
report_slack -pin u3:a -rise -late
```

3 Output File

This section explains the output of the commands from Section 2.6, and provide some sample output. All output should be directed into `<output file>` from the command

`%>myTimer <.tau2015> <.timing> <.ops>` **<output file>**

3.1 report_at

This command's output is expected to be a single decimal value in picoseconds (*ps*). For example, if the early rise arrival time at `inp1` is 2.5*ps*, `report_at -pin inp1 -early -rise` would produce 2.5 on its own separate line:

```
%> report_at -early -rise -pin inp1
%> 2.5
```

3.2 report_rat

This command's output is expected to be a single decimal value in picoseconds (*ps*). For example, if the late fall required arrival time at `inp1` is 5.0*ps*, `report_rat -pin u1:a -late -fall` would produce 5.0 on its own separate line:

```
%> report_rat -pin u1:a -late -fall
%> 5.0
```

3.3 report_slack

This command's output is expected to be a single decimal value in picoseconds (*ps*). For example, if the late rise slack at `inp1` is 3.0*ps*, `report_slack -late -pin inp2 -rise` would produce 3.0 on its own separate line:

```
%> report_rat -late -pin inp2 -rise
%> 3.0
```

3.4 report_worst_paths

This command's output is expected to have the following syntax:

```
report_worst_paths <paths reported>
Path 1: <path type> <post-CPPR path slack> <path length> <mode>
<Deepest pin in path, e.g., primary output or data-end of test> <transition>
<Immediate upstream pin> <transition>
...
<Primary input> <transition>
Path 2: <path type> <post-CPPR path slack> <path length> <mode>
...
```

<paths reported> and <path length> are expected to be integers, and the <post-CPPR path slack> should be in picoseconds. <path type> can either be `Setup` or `Hold`, where the path is traced from an explicit timing test at a latch, or `RAT`, which the path is traced from a primary output. For the set of paths, only print out the paths with *non-positive post-CPPR* slacks. If <numPaths> is greater than the total number of paths through the pin, only output the total number of paths through the pin. The number of paths that are reported should correspond to the set of output paths. The paths should be in order of criticality according to their *worst slack*. Do not print any path with positive slack. For each path, the <mode> should be `E` (early) or `L` (late). Within each path, the transition should be `R` (rise) or `F` (fall). For example, `report_worst_paths -numPaths 2 -pin inp1` could produce:

```
01. report_worst_paths 1
02. Path 1: hold -30.0 6 E
03. f1:d F
04. u4:o F
05. u4:a R
06. u1:o R
07. u1:a F
08. inp1 F
```

Line 01 specifies that the output contains one (1) path, even though <numPaths> was specified as 2. Line 02 specifies the first path with a slack of -30 *ps*, a length of 6, and the path is during early mode. Lines 03-08 specify the path, where each line specifies a design pin and its corresponding transition.

4 Updates

- 02/09/2015 [v1.15]: removed semi-colon in the `report_worst_paths` output.
- 01/05/2015 [v1.14]: corrected `create_net` to `insert_net` in the **.ops** file.
- 12/08/2014 [v1.13]: reverted information about duty cycle, and corrected the example of "`clock tau2015_clk 50 50`".
- 12/08/2014 [v1.12]: removed outdated information regarding clock duty cycle.
- 11/12/2014 [v1.11]: added `clock` field to **.lib** syntax, clarified `non_unate` transitions and `timing_sense`, clarified `report_slack output`.
- 11/11/2014 [v1.10]: added **.lib** table template for timing tests, added command `insert_net`, corrected path type syntax.
- 11/10/2014 [v1.9]: changed format of **.tau2015**, split the `.lib` into early and late.
- 11/07/2014 [v1.8]: amended `report_worst_paths` example with path type.
- 10/31/2014 [v1.7]: added `set_load` keyword support for **.timing**.
- 10/30/2014 [v1.6]: added `slew` keyword support for **.timing**.
- 10/01/2014 [v1.5]: fixed `report_worst_paths` output path format to be consistent, clarified <`read_spef`> file location, clarified **.ops** processing.
- 10/01/2014 [v1.4]: release version.