ECE 449, Fall 2014
Project 4: Logic Simulation

Initial Release Due: 11/21 (Fri.), by the end of the day
Final Release Due: 12/08 (Mon.), by the end of the day

1 Logic Simulation

With the netlist constructed in Project 3, we are now ready to design and implement algorithms in our simulator to simulate the corresponding circuit.

Recall that we are interested in simulating flip-flop-based synchronous circuits specified in the EasyVL language. Therefore, we will rely on the following FSM model of the circuits to define the meaning of logic simulation for the circuits specified in EasyVL.

- The *state* of the FSM is represented by the bits stored in the flip-flops.
- The *input* and the *output* of the FSM are represented by the input and the output bits of the circuit respectively.
- A synchronous circuit has a single clock that drives all the flip-flops. There is exactly one *state transition* in the FSM for each clock cycle.
- The *combinational part* of the circuit, i.e. the part obtained by removing all the flip-flops, defines how the output and the next state is computed from the input and the current state of the FSM.

Leveraging the FSM model, for a flip-flop-based synchronous circuit specified in EasyVL, by logic simulation we mean to determine the outputs of the FSM for a given number of state transitions with a given *initial state* and given input at each state transition. We can have the following iterative algorithm for logic simulation: there is one state transition for each iteration where the output and the next state of the FSM are computed from the input and the current state, and the next state will then be used as the current state for the next iteration. To simplify our simulator design, we assume that first, the initial state of the FSM is the state where all the flip-flops store a logical value of 0; second, only the first 1000 transitions should be simulated.

The correctness of the simulation can be determined by observing the outputs of the FSM. This is achieved by storing the logical values of the circuit outputs at the end of each state transition to files. The details will be discussed in the next section. Note that not all the circuits specified in EasyVL are synchronous circuits. The following requirements must be satisfied for the circuits used as test cases for Project 4:

- There is no cycle in the combinational part of the circuit.
- There is exactly one driver for each net, except for the nets driven by tri-state buffers.

You can be assured that the test cases provided with the golden simulator satisfy these requirements and please use the golden simulator to verify your own test cases. You are NOT required to handle these situations in your program.

2 Functional Semantics of Basic Gates

We first discuss the functional semantics of the basic gates that allows to simulate 7 out of the 10 golden test cases (except **tris_lut.evl**, **cpu8_flat.evl**, and **cpu32_flat.evl**).

2.1 Combinational Gates

The combinational gates include `and`, `or`, `xor`, `not`, `buf`, `evl_one`, and `evl_zero`. Their functionality can be specified by the boolean relations between their inputs and outputs.

- `and name(out,in_1,in_2,...,in_k);`
  If all the inputs `in_1`, `in_2`, ..., `in_k` are 1, then the output `out` is 1; otherwise `out` is 0.
- `or  name(out,in_1,in_2,...,in_k);`
  If all the inputs `in_1`, `in_2`, ..., `in_k` are 0, then the output `out` is 0; otherwise `out` is 1.
- `xor name(out,in_1,in_2,...,in_k);`
  If there are even number of 1's among the inputs `in_1`, `in_2`, ..., `in_k`, then the output `out` is 0; otherwise `out` is 1.
- `not name(out,in);`
  If the input `in` is 1, then the output `out` is 0; otherwise `out` is 1.
- `buf name(out,in);`
  If the input `in` is 1, then the output `out` is 1; otherwise `out` is 0.
- `evl_one name(out_1,out_2,...,out_k);`
  All the outputs `out_1`, `out_2`, ..., `out_k` are 1. Please note that the outputs could be buses.
- `evl_zero name(out_1,out_2,...,out_k);`
  All the outputs `out_1`, `out_2`, ..., `out_k` are 0. Please note that the outputs could be buses.

2.2 Flip-Flops

Recall that a flip-flop is defined as follows in EasyVL:

```
evl_dff name(q,d,clk);
```

Each flip-flop should store a bit of the state, either 0 or 1. At any point within an iteration of the simulation algorithm where the computation is performed for one state transition, the output `q` should take the same value as the bit of the state. At the end of each iteration and before the next iteration begins, the bit of the state should be updated to be the same as the input `d`. Note that as mentioned in Section 1, the bits stored in all the flip-flips are 0 initially.

For simplity, you can assume the clock network of the circuit is connected properly so that there is no need to verify the input `clk` actually connects to the global clock as defined by `evl_clock`.

2.3 Circuit Outputs

Circuit outputs in EasyVL can be grouped into output gates, while an output gate is defined as follows:

```
output name(in_1,in_2,...,in_k);
```

For each output gate, you should create a file to save the output signals (the inputs to the output gate) at the end of each state transition. The output file name is the name of the EasyVL file concatenated with **.**, the name of the output gate, and then **.evl_output**, e.g. the output file for an output gate `sim_out` in **test.evl** should be **test.evl.sim_out.evl_output** . Note that it is assumed that different output gates in a test case have different names so there will be no conflict of output file names.

The output file should have the following format.

```
N_number_of_pins
W1_pin_1_width
W2_pin_2_width
...
WN_pin_N_width
T1_pin_1_value T1_pin_2_value ... T1_pin_N_value
T2_pin_1_value T2_pin_2_value ... T2_pin_N_value
...
T1000_pin_1_value T1000_pin_2_value ... T1000_pin_N_value
```

Here are the details,

- The first line describes the number of pins of this output gate.
- The pin widths are then written to the file following the order they appear within the `()`.

- The logical values of the pins are written to the file. Since there are 1000 state transitions (`T1` to `T1000`), each additional line shows the logical values of the pins at the end of the corresponding state transition.

- The value of a pin should be written to the file as hexadecimal (base 16) integers. Valid digits include `0` to `9` and `A` to `F`. For a pin of w bits, there should be $\lceil w/4 \rceil$ (round up to the nearest integer) hexadecimal digits. The digits can be generated from right to left so that there is one for each 4 bits starting from the least significant bit of the pin (please use 0 to pad the last few bits if w is not a multiple of 4). In other words, you should keep the leading 0's when generating the output value (e.g. `0009` instead of `9` for a pin of width 15).

2.4 Circuit Inputs

Similar to circuit outputs, circuit inputs are grouped into input gates, while an input gate is defined as follows:

```
evl_input name(out_1,out_2,...,out_k);
```

For each input gate, you should prepare a file so you can read it in your C++ program to supply input signals (the outputs of the input gate) at the beginning of each state transition. The input file name is the name of the EasyVL file concatenated with **.**, the name of the input gate, and then **.evl_input**, e.g. the input file for an input gate `sim_in` in **test.evl** should be **test.evl.sim_in.evl_input** .

The input file follows the format shown below.

```
N_number_of_pins W1_pin_1_width W2_pin_2_width ... WN_pin_n_width
number_of_transitions pin_1_value pin_2_value ... pin_N_value
number_of_transitions pin_1_value pin_2_value ... pin_N_value
...
number_of_transitions pin_1_value pin_2_value ... pin_N_value
```

Here are the details,

- The first line describes the number of pins of this input gate and the width of each pin. The widths should match the connections specified in the EasyVL file.

- Each additional line introduces a set of values that should be assigned to the pins. The values should be used for `number_of_transitions` state transitions and then be updated according to the next line. If the end of file is reached before the simulation terminates, the values from the last line should be used for all the remaining state transitions.

- Similar to the output gates, the values are hexadecimal (base 16) integers. However, we allow to use the digits `a` to `f` as alternatives to `A` to `F` and to trim the leading 0's (e.g. `9` stands for `0009` for a pin of width 15).

3 Simulation with Additional Gates

We then discuss the gates that are necessary to simulate **tris_lut.evl**, **cpu8_flat.evl**, and **cpu32_flat.evl**.

3.1 Look-up Tables

A look-up table can be used to model a piece of read-only memory (ROM) in a simulation system whose content can be specified by designers before simulation starts. In EasyVL, look-up tables are specified as `evl_lut` gates.

```
evl_lut name(word,address);
```

A `evl_lut` gate should have exactly 2 pins and each pin can have an arbitrary width, i.e. connected to a single net or a set of nets specified as a range of bits in a bus. The first pin `word` is the output of the gate and the second pin `address` is the input of the gate.

For each `evl_lut` gate, you should prepare a file so you can read it in your C++ program to supply the output `word` depending on the input `address`. The file name is the name of the EasyVL file concatenated with `.`, the name of the `lut` gate, and then **.evl_lut**, e.g. the file for a `lut` gate `rom` in **test.evl** should be **test.evl.rom.evl_lut** .

The file follows the format shown below.

```
word_width address_width
word_0_value
word_1_value
...
word_N_value
```

Here are the details,

- The first line describes the width of `word` and the width of `address`. They should match the widths of the two pins.
- Each additional line introduces a word stored in the look-up table at the corresponding address. The address starts from 0 and is incremented by 1 per line.
- Similar to the `evl_input` gates, the values are hexadecimal (base 16) integers. We allow to use the digits `a` to `f` as alternatives to `A` to `F` and to trim the leading 0's (e.g. `9` stands for `0009` for a word of width 15).
- You don't need to specify all the words but at runtime, any attempt to access a word on an address that is not specified should be treated as an error that terminates the simulation immediately.

3.2 Tri-State Buffers and Four-Valued Logic

Many realistic hardware designs use tri-state buffers to construct extremely large multiplexers (mux's) as they consume much less hardware than those constructed from usual logic gates like `and` and `not`. However, as tri-state buffers cannot be specified in boolean logic, we have to extend our logic model in order to provide support to tri-state buffers.

The typical method to handle tri-state buffers and other realistic situations in logic simulation is to extend boolean logic to four-valued logic. Four-valued logic utilizes four logical values – 0, 1, Z, and X instead of two in boolean logic. While 0 and 1 have the same meaning as boolean logic, Z refers to the logical value on a net that is not driven by any gate, i.e. a floating net, and X refers to any logical value that is not known, e.g. the initial state of a flip-flop in a real circuit (note that we simplified our simulator design by assuming every flip-flop has an initial state of 0).

Recall the tri-state buffers are defined in EasyVL as follows:

```
tris name(out,in,en);
```

Therefore, if the input `en` is 1, then the output `out` takes the value of the input *in*; otherwise `out` is Z.

To simplify our simulator design, we assume that no X value appears during the simulation. For gates other than `buf` and `tris`, the Z values will not appear at the outputs and the inputs – it is therefore not necessary to change their functional semantics. For the `buf` gate, if the input is Z, then the output should also be Z. On the other hand, allowing Z values to appear at the outputs of `buf` and `tris` gates enables one to construct a mux by using a net to "or" its inputs. Note that a net may have multiple drivers after we introduce tri-state buffers. There are two valid situations you need to handle: if all drivers output Z, then the net should have a logical value of Z; otherwise, there should be exactly one driver outputing 0 or 1 and thus determining the logical value on the net, while all other drivers should output Z. Any other logical value configurations at runtime should result in an error that terminates the simulation immediately.

4 Suggestions

For the initial release, it is recommended to provide support for `not`, `evl_dff`, and `evl_output`. This helps to ensure that you understand how cycle simulation works.

Then, please consider to implement all the basic gates except `evl_input`, which should not be very difficult and would allow to simulate 5 out of the 10 golden test cases. Adding support for `evl_input` would allow to simulate 2 more golden test cases (**io.evl** and **counter_flat.evl**).

Finally, if you still have time, consider to implement `evl_lut` and `tris` for the remaining 3 golden test cases (**tris_lut.evl**, **cpu8_flat.evl**, and **cpu32_flat.evl**).

You could be interested in implementing certain features in the bonus project as two of the test cases there (**adder_test.evl** and **counter.evl**) do not need support of `evl_lut` and `tris`, while being able to simulate them will give you 6 points.

5 Required Program Output

The required program outputs are just those files created for the `evl_output` gates in the circuit as mentioned earlier.