ECE 449, Fall 2017
Project 3: Netlist Construction

Initial Release Due: 10/20 (Fri.), by the end of the day
Final Release Due: 11/03 (Fri.), by the end of the day
Project Report Due: 11/06 (Mon.), by the end of the day

1 Netlist Construction

By analyzing the semantics of the statements, we have extracted the descriptions of the wires, the components, and their interconnections in a circuit. The objective of this project is to design a data structure called *netlist* to represent the circuit for simulation and to implement a program to actually construct the netlist from the descriptions of the circuit.

A netlist is consisting of *signal net*s (or simply *net*s) that pass signals around and components that perform computations. The reason to have signal nets instead of wires in a netlist is because the former are much easier to handle in simulation when buses are presented. In the netlist, each wire of width k is broken into k signal nets where each net passes exactly one bit of signal. For example, a wire representing a 8-bit bus converts to 8 signal nets in the netlist. Note that for Project 3 and 4, the components are predefined gates, while for the bonus project, the components could be modules defined in the same EasyVL file.

While you can always search the outcome of Project 2 to locate wires, components, and their interconnections for simulation without using the netlist data structure, it is not efficient to do so for large circuits. In order to perform circuit simulation efficiently, the simulator relies on the netlist data structure to provide the following information. First, one should be able to access all the gates and nets in the circuit. Second, for each gate, one should be able to obtain input signals and to compute output signals. Third, for each net, one should be able to determine the signal from the gates driving it. While the details on computing the signals will be introduced in Project 4, this project focuses on the *structural* representation, i.e. nets, component, and interconnections, of the circuit as a netlist. Therefore, the netlist data structure should allow one to access:

- Gates and nets in the circuit;
- Nets connected to each gate;
- Gates connected to each net.

You should design the netlist data structure following the principles of object-oriented design (OOD), where a complex system is decomposed into manageable pieces, i.e. the *object*s. An intuitive but usually quite effective method to identify objects in a system is to look for nouns in the requirements. For the netlist, this method leads to three obvious types of objects: the netlist itself, the gates, and the nets. On the other hand, other entities in the circuit, like the pins, could be quite useful to model connections and can be part of your design. Anyway, you are free to choose the objects in your design as long as the above requirements are fulfilled.

From Project 2, we obtained the descriptions of wires and components in a module. The netlist data structure is populated using the descriptions to represent the circuit as specified by the module. Since all the gates have predefined semantics, an important task during netlist construction is to verify that pins of each gate match its semantics, which we will discuss in detail in the next two sections.

2 Semantics of Pins

Since the pins are used to represent the wires connected to a component, we must first make sure that the wires they refer to do exist and that the bit ranges, if specified, match the definition of wires. The requirements are summarized in the following rules. In addition, since you may need to use a single data type to handle the optional msb/lsb fields for a pin, the rules also specify a method to update them so that they can be processed consistently later, no matter they are specified in the EasyVL file or not.

- The name of a pin must be the name of some wire.
- If the wire is not a bus (whose width is 1), then both msb/lsb of the pin must be -1 as being assigned during the transition from **PINS** to **PIN_NAME**. Note that this rule implies that the range is not specified.
- If the wire is a bus (whose width is at least 2),
  - If neither msb nor lsb is specified (both are -1), then the msb should be updated to width -1 and the lsb should be updated to 0, since the pin refers to the whole bus.
  - If both msb and lsb are specified, then it must be true that width $>$ msb $\geq$ lsb ≥ 0.
  - If the msb is specified but the lsb is not (the lsb is -1), then it must be true that width $>$ msb ≥ 0. The lsb should be updated to be the same as the msb, since the pin refers to a single bit within the bus.

3 Structural Semantics of Gates

Then, we verify the structural semantics of gates, i.e. the pins of each gate have the desired number of bits. For Project 3 and 4, our simulator should support the following predefined gates from the EasyVL language: `and`, `or`, `xor`, `not`, `buf`, `tris`, `evl_clock`, `evl_dff`, `evl_one`, `evl_zero`, `evl_input`, `evl_output`. We group them as follows and discuss their structural semantics. Note that according to Project 2, for each gate, its name is optional.

- ```
  and name(out,in_1,in_2,...,in_k);
  or  name(out,in_1,in_2,...,in_k);
  xor name(out,in_1,in_2,...,in_k);
  ```
  These three types of gates should have at least 3 pins and each pin should have a width of 1 bit, i.e. connected to a single net. The first pin `out` is the output of the gate and the rest of the pins `in_1`, `in_2`, ..., `in_k` are the k inputs of the gate.

- ```
  not name(out,in);
  buf name(out,in);
  ```
  These two types of gates should have exactly 2 pins and each pin should have a width of 1 bit. The first pin `out` is the output of the gate and the second pin `in` is the input of the gate.

- ```
  evl_dff name(q,d,clk);
  tris    name(out,in,en);
  ```
  These two types of gates should have exactly 3 pins and each pin should have a width of 1 bit. The first pin is the output and the rest two are the inputs.

- ```
  evl_clock name(clk);
  ```
  The global clock is specified by this gate type. It has a single output that should have a width of 1 bit.

- ```
  evl_one   name(out_1,out_2,...,out_k);
  evl_zero  name(out_1,out_2,...,out_k);
  evl_input name(out_1,out_2,...,out_k);
  ```
  These three types of gates should have at least 1 pin 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 pins `out_1`, `out_2`, ..., `out_k` are the k outputs of the gate.

- ```
  evl_output name(in_1,in_2,...,in_k);
  ```
  The `output` gates should have at least 1 pin and each pin can have an arbitrary width. The pins `in_1`, `in_2`, ..., `in_k` are the k inputs of the gate.

You may already have figured out the functional semantics of the gates, i.e. their boolean functionalities in a circuit, from their types, though we will delay the discussions until Project 4.

4 Suggestions

For the initial release, it is recommended to support 1-bit wires and a single type of gates like `and`. This helps to ensure that you'll be able to locate wires for pins, and to validate their widths.

Support for buses and the rest of the gates could be added for the final release with possible code refactoring.

5 Required Program Output

To verify the correctness of netlist construction, your program should output the netlist into an output file. Similar to Project 1 and 2, the output file name is the name of the EasyVL file concatenated with **.netlist**, e.g. the output file for **test.evl** should be **test.evl.netlist** .

We start with an example circuit **bus.evl** as follows.

```
module top;

  wire [1:0] in;
  wire out;

  evl_zero(in[0]);
  evl_one(in[1]);

  and(out, in[0], in[1]);

  evl_output sim_out(out, in);

endmodule
```

The content of **bus.evl.netlist** should be.

```
module top
nets 3
  net in[0] 3
    evl_zero 0
    and 1
    evl_output sim_out 1
  net in[1] 3
    evl_one 0
    and 2
    evl_output sim_out 1
  net out 2
    and 0
```

```
    evl_output sim_out 0
components 4
  component evl_zero 1
    pin 1 in[0]
  component evl_one 1
    pin 1 in[1]
  component and 3
    pin 1 out
    pin 1 in[0]
    pin 1 in[1]
  component evl_output sim_out 2
    pin 1 out
    pin 2 in[0] in[1]
```

Formally, the output file should have the following format.

```
module module_type
nets M_number_of_nets
  net net_1_name K1_number_of_pins
    pin_1_gate_type[ pin_1_gate_name] pin_1_position
    pin_2_gate_type[ pin_2_gate_name] pin_2_position
    ...
    pin_K1_gate_type[ pin_K1_gate_name] pin_K1_position
  net net_2_name K2_number_of_pins
    pin_1_gate_type[ pin_1_gate_name] pin_1_position
    pin_2_gate_type[ pin_2_gate_name] pin_2_position
    ...
    pin_K2_gate_type[ pin_K2_gate_name] pin_K2_position
  ...
  net net_M_name KM_number_of_pins
    pin_1_gate_type[ pin_1_gate_name] pin_1_position
    pin_2_gate_type[ pin_2_gate_name] pin_2_position
    ...
    pin_KM_gate_type[ pin_KM_gate_name] pin_KM_position
components N_number_of_components
  component gate_1_type[ gate_1_name] L1_number_of_pins
    pin W1_pin_1_width net_1_name net_2_name ... net_W1_name
    pin W2_pin_2_width net_1_name net_2_name ... net_W2_name
    ...
    pin WL1_pin_L1_width net_1_name net_2_name ... net_WL1_name
  component gate_2_type[ gate_2_name] L2_number_of_pins
    pin W1_pin_1_width net_1_name net_2_name ... net_W1_name
    pin W2_pin_2_width net_1_name net_2_name ... net_W2_name
    ...
```

```
  pin WL2_pin_L2_width net_1_name net_2_name ... net_WL2_name
...
component gate_N_type[ gate_2_name] LN_number_of_pins
  pin W1_pin_1_width net_1_name net_2_name ... net_W1_name
  pin W2_pin_2_width net_1_name net_2_name ... net_W2_name
  ...
  pin WLN_pin_LN_width net_1_name net_2_name ... net_WLN_name
```

The format is summarized as follows.

- Similar to Project 1 and 2, the output file is organized into lines and the strings on each line are separated by spaces.
- The first line describes the module and its type.
- Then, after a line that shows the number of nets, the nets are written to the file following the order their corresponding wires appear in the EasyVL file, and from the lowest bit (0) to the highest for the multiple nets belong to the same wire.
  - The name of each net is determined as follows: if the net is derived from a wire of width 1, then its name is the same as the wire; otherwise, for a wire of width $k+1$, e.g. the wire `a` as defined in `wire [k:0] a;`, it is broken into $k+1$ nets `a[0]`, `a[1]`, ..., `a[k]`.
  - Each net occupies multiple lines depending on the pins it is connected to. The first line shows its name and the number of the pins. The pins are then written to the file following the order their corresponding gates appear in the EasyVL file.
  - Each pin occupies a line showing the type and the name of the gate it belongs to, and its position within the gate. The name of the gate should be omitted if it is empty. The first pin of a gate is assigned a position 0, the second is assigned 1, and so on.
- Finally, after a line that shows the number of gates (components in general), the gates are written to the file following the order they appear in the EasyVL file.
  - Each gate occupies multiple lines depending on the pins it contains. The first line shows its type, its name, and the number of its pins. The name should be omitted if it is empty. The pins are then written to the file following the order they appear within the `()`.
  - Each pin occupies a line showing its width and the names of the nets connected to it. The nets are shown following the order of bits from the least significant to the most significant.
- The indentations are added to the beginning of each line to make the golden output more readable. It is up to you to implement this feature or not. However, no extra space should be introduced in the middle of the line.

As a reminder, it is always a good idea to run the golden simulator in order to understand the above format.