ECE 449, Fall 2014
Project 2: Syntactic Analysis

Initial Release Due: 10/03 (Fri.), by the end of the day
Final Release Due: 10/10 (Fri.), by the end of the day

1 Overview

Complex digital systems are usually built in a hierarchical manner by creating *component*s and interconnecting them together using *wire*s. Hardware description languages like Verilog, VHDL, and EasyVL in particular for this course, allow to describe such hierarchical and structural relationships between components and wires in a textual form such that a system prototype can be quickly created by reusing existing designs. A key feature of such languages is that one can define a blueprint of the same kind of components, called *module*, that can be used to construct those components in other modules. For example, you may define a module for 1-bit 16-to-1 mux'es such that any time you need a 1-bit 16-to-1 mux, e.g. for a register file, you can construct it directly from the blueprint without elaborating the many gates inside.

Clearly, when you define the blueprint, i.e. the module, it is necessary to define its inputs and outputs, called its *port*s, as well as how the inputs and outputs are related. For the structural relationship that we consider in this course, we define a module by describing the ports that could either be inputs or outputs, the components that could either be primitive logic gates or be constructed from other modules, and the wires that connect them together. Each wire could be a *bus* that allows to bundle multiple bits of information together. Moreover, as both components and primitive gates may have multiple inputs and outputs, we define *pin*s to be where the wires are connected to a single component or a primitive gate. For example, a 2-input AND gate has 3 pins, and a 2-to-1 mux has 4 pins.

To simulate a system design, a simulator will need to identify the *top* module from the textual description. The top module is a module without any port, i.e. it has no input and output. It usually contains a component constructed from the module for the system itself, and other components and wires providing stimuli to drive the simulation.

We will not handle hierarchical designs until for the bonus project. Therefore, for Project 2, 3, and 4, we would assume all components to be primitive gates and there is a single module, which is also the top module in our designs. Nevertheless, we still need to study how to model the structural relationship and how to perform the simulation,

and in Project 2, we will perform syntactic analysis, i.e. to study the grammar of the EasyVL language that in particular defines wires, components, and pins.

2 Tokens and Statements

Once the the EasyVL file is converted to a sequence of tokens, syntactic analysis will group tokens into statements and extract components and interconnects leveraging the semantics of the statements.

In the EasyVL language, the sequence of tokens is broken down into a sequence of statements, where each statement should be among the following four types.

- **MODULE**: the sequence starts with a **NAME** token `module` and ends with a **SINGLE** token `;`.
- **ENDMODULE**: the sequence contains one **NAME** token `endmodule`.
- **WIRE**: the sequence starts with a **NAME** token `wire` and ends with a **SINGLE** token `;`.
- **COMPONENT**: the sequence starts with a **NAME** token which is not among `module`, `wire`, or `endmodule`, and ends with a **SINGLE** token `;`.

Note that except the bonus project where hierarchical designs are support, the EasyVL file should contain exactly one module (the top module). Therefore, the first statement must be **MODULE**, the last statement must be **ENDMODULE**, and any statement in between must be either **WIRE** or **COMPONENT**.

For example, consider **test.evl** shown below.

```
// a comment
module top;

   wire s0, s1, clk;

   evl_clock(clk);

   evl_dff(s0, s1, clk);

   not(s1, s0);

endmodule
```

There are 6 statements. The first one is **MODULE**. The second one is **WIRE**. Then there are three **COMPONENT** statements. The last one is **ENDMODULE**.

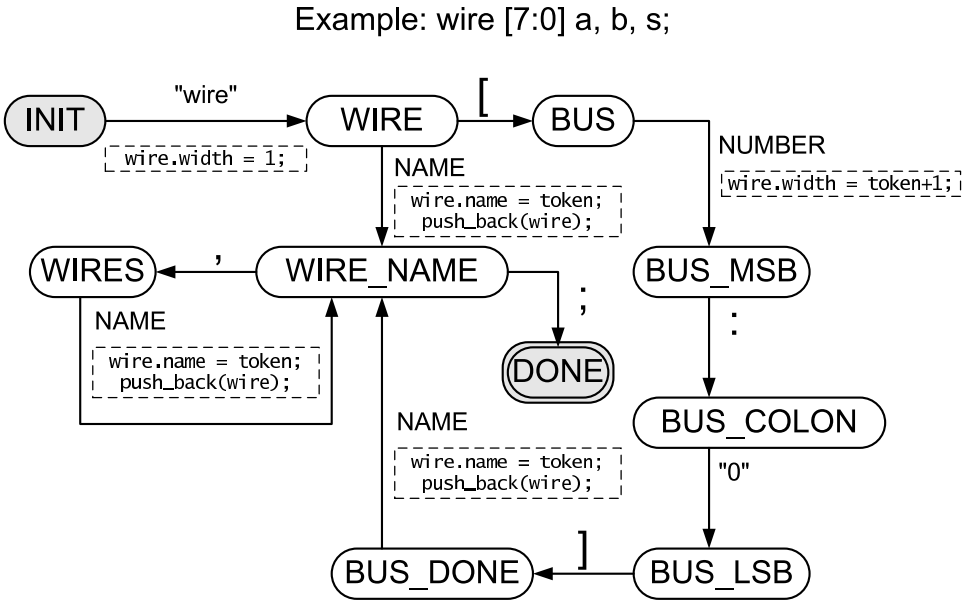

Figure 1: FSM for **WIRE** statements

3 Syntactic Analysis

While the definitions in the previous section can be used to construct statements from tokens, we need to further specify the internal structure of the statements, i.e. their syntax, in order to understand their semantics.

Obviously a **ENDMODULE** statement has no internal structure. A **MODULE** statement, on the other hand, has a very straightforward structure for the top module: there must be three tokens – `module`, a **NAME** token given the type of the module, and `;`. However, both the **WIRE** and **COMPONENT** statements are much more complicated because the former can define arbitrary number of wires and the latter can have many pins, not to mention that wires and pins could be buses or part of buses.

Therefore, we need a model to specify these two types of statements. Since there is no recursive structure within the syntax, we can use FSMs, i.e. the same model for synchronous circuits, to model them. This is not quite a coincidence since FSM is one of the most practical computation models.

3.1 WIRE Statements

The FSM for **WIRE** statements is shown in Fig. 1. There are 10 states and 11 transitions in this FSM. To extract wires defined in a **WIRE** statement, you should always start from the **INIT** state and consume one token per each state transition. Any token that would not match any out-going transition for the current state would be a violation of the syntax. If all the tokens are consumed at the same time the state reaches

DONE, the statement is a valid one. A **WIRE** statement may have an optional part specifying that the wires should be buses. This optional part is captured by the 5 states starting at **BUS**. For simplicity, EasyVL requires that every bus must be defined in the form `[width-1:0]` and that if the optional part is presented, the bus width must be at least 2. After this optional part, the `,` separated list of wire names are captured by the two states **WIRES** and **WIRE_NAME**. Each time a new wire name is identified, we can store the name and the width in some data structure.

3.2 COMPONENT statements

Similarly, the FSM for **COMPONENT** statements is shown in Fig. 2. The component name is optional and should be empty(`""`) if not specified. Within the `()`, there are multiple pins separated by `,` representing the connections of wires to the component. When the wire is a bus, a pin may connect to a range of bits within the bus. The range is optional and is specified by the most-significant-bit (msb) and the least-significant-bit (lsb) within the `[]` following the pin name. If both msb and lsb are specified, they are separated by `:`; otherwise the msb is specified such that the pin connects to only one bit within the bus.

3.3 Suggestions

Your initial release should at least support the syntax of the **MODULE** and **END-MODULE** statements since they are necessary to identify the top module in the test cases. It is recommended to support the **WIRE** statement for multiple wires so that you will understand how the FSM model works. It is up to you to support the bus in the **WIRE** statement or not.

Support for **COMPONENT** statements should be added last once your code works for all cases of the **WIRE** statement. Code refactoring may be necessary. Most likely it is a good idea to implement such support only for the final release.

4 Required Program Output

To verify the correctness of syntactic analysis, your program should store the syntax into an output file. Similar to Project 1, the output file name is the name of the EasyVL file concatenated with **.syntax**, e.g. the output file for **test.evl** should be **test.evl.syntax** .

The output file should have the following format with the items between `[]` being optional.

```
module module_type
wires M_number_of_wires
  wire wire_1_name wire_1_width
  wire wire_2_name wire_2_width
```

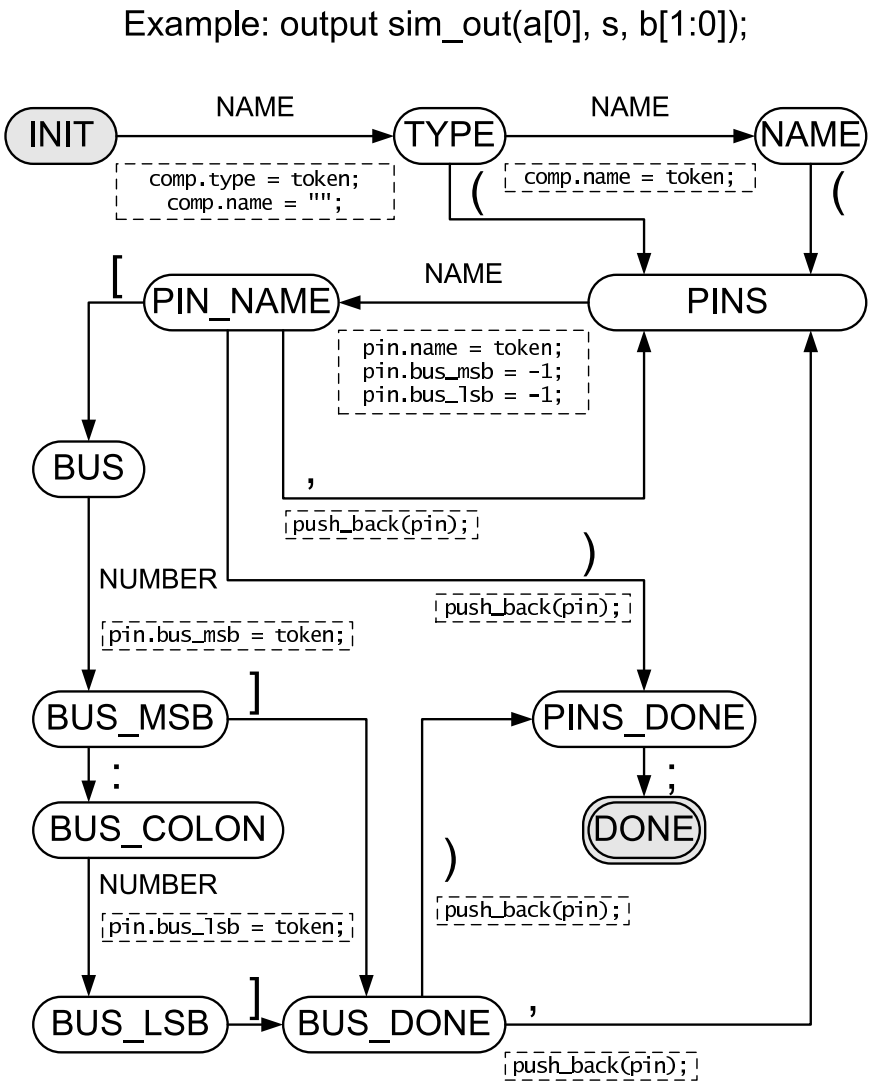

Figure 2: FSM for **COMPONENT** statements

```
  ...
  wire wire_M_name wire_M_width
components N_number_of_components
  component component_1_type[ component_1_name] L1_number_of_pins
    pin pin_1_of_component_1_name[ pin_1_msb][ pin_1_lsb]
    pin pin_2_of_component_1_name[ pin_2_msb][ pin_2_lsb]
    ...
    pin pin_L1_of_component_1_name pin_L1_msb pin_L1_lsb
  component component_2_type[ component_2_name] L2_number_of_pins
    pin pin_1_of_component_2_name[ pin_1_msb][ pin_1_lsb]
    ...
    pin pin_L2_of_component_2_name[ pin_L2_msb][ pin_L2_lsb]
  ...
  component component_N_type[ component_N_name] LN_number_of_pins
    pin pin_1_of_component_N_name[ pin_1_msb][ pin_1_lsb]
    ...
    pin pin_LN_of_component_N_name[ pin_LN_msb][ pin_LN_lsb]
```

For example, the content of **test.evl.syntax** should be.

```
module top
wires 3
  wire s0 1
  wire s1 1
  wire clk 1
components 3
  component evl_clock 1
    pin clk
  component evl_dff 3
    pin s0
    pin s1
    pin clk
  component not 2
    pin s1
    pin s0
```

The format is summarized as follows.

- Similar to Project 1, 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 wires, the wires are written to the file following the order they appear in the EasyVL file.

  - Each wire occupies one line showing its name and width.

- Finally, after a line that shows the number of components, the components are written to the file following the order they appear in the EasyVL file.
  - Each component may occupy multiple lines depending on the number of 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.
  - Then, each pin occupies an additional line showing its name, its msb, and its lsb. The pins should follow the order they appear within the `()`. The msb should be omitted if it is -1 and the lsb should also be omitted if it is -1.

- 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.