IIT Armour College of Engineering
ILLINOIS INSTITUTE OF TECHNOLOGY

ECE 449 - OBJECT ORIENTED COMPOTER SIMULATION

FALL 2014

NAME: HARSHITA RAVI SHANKAR

CWID: A20327797

PROJECT 2: SYNTACTIC ANALYSIS

INTRODUCTION

In project one, we had converted the EasyVL file into sequence of tokens. As an advanced version of it, in project two, here we group the tokens of the EasyVL file into statements. The components are extracted and it then interconnects leveraging the semantics of the statements. That sequence is further broken down into a sequence of statements, where each statement is divided in four types, viz., module, endmodule, wire and component. Visual c++ is used to write the program and the golden simulator is used to test the result.

IMPLEMENTATION APPROACH

Initially the EasyVL file is grouped into sequence of tokens, which is further broken down into a sequence of statements. Each statement should be among one of the four types as given below:

- 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 '**;**'

There are various other modules that are being used here. They have been invoked in the main function. They have been explained below.

A. Program implementation in Source file

- Structure element (evl_name): It defines tokens, statements, wires, components, modules and pins.
- Boolean module: This returns true or false value depending on the requirements as explained below.
    - Boolean Module, Extract token from Line: We first check for spaces or comments. We then extract each token depending on its type (name, single or number). Every line of the .evl file needs to be checked and accordingly store the line number.
    - Boolean Module, Extract token from File: If there is no input file, an error message will be displayed, else it will be extracted from the input files.
    - Boolean Module, Store token to file: Similar to the one as described in project one, but it is defined in another module which returns a Boolean value (True or False).
    - Boolean module to test the token as semicolon: When statements are ended in EasyVL file, as defined in statement types, we differentiate them using semicolon.
    - Boolean module moves the tokens into statements using the statements as defined above. All the tokens are moved to statements accordingly.
    - Boolean module, group tokens into statements using token file and line number which we have stored earlier, then separate each token into different modules accordingly.

- Display statements according to the respective types.
- Store statements to file with the extension .statements.

B. Implementation of Module and Main function:

- **Wire module:** A wire module is one which begins with the wire token and end with a single token ';'. There are 10 states and 11 transitions in this Finite State Machine (FSM). To extract all the wires defined in a WIRE statement, you should always start from the INIT state and consume one token per state transition. If all the tokens are consumed at the same time the state reaches DONE, and hence the statement is a valid one. If any of the tokens do not match any out-going transition for the current state, violation of the syntax occurs and an error message will be displayed. A WIRE statement may have an optional part specifying that the wires should be buses. This optional part is captured by 5 states starting at BUS. For simplicity, EasyVL requires that every bus must be defined in the form [width-1:0] and if the optional part is present, the bus width must be at least 2. The separated list of wire names are then captured by the two states, WIRES and WIRE NAME. Therefore, each time a new wire name is identified, we can store the name and the width in some data structure.

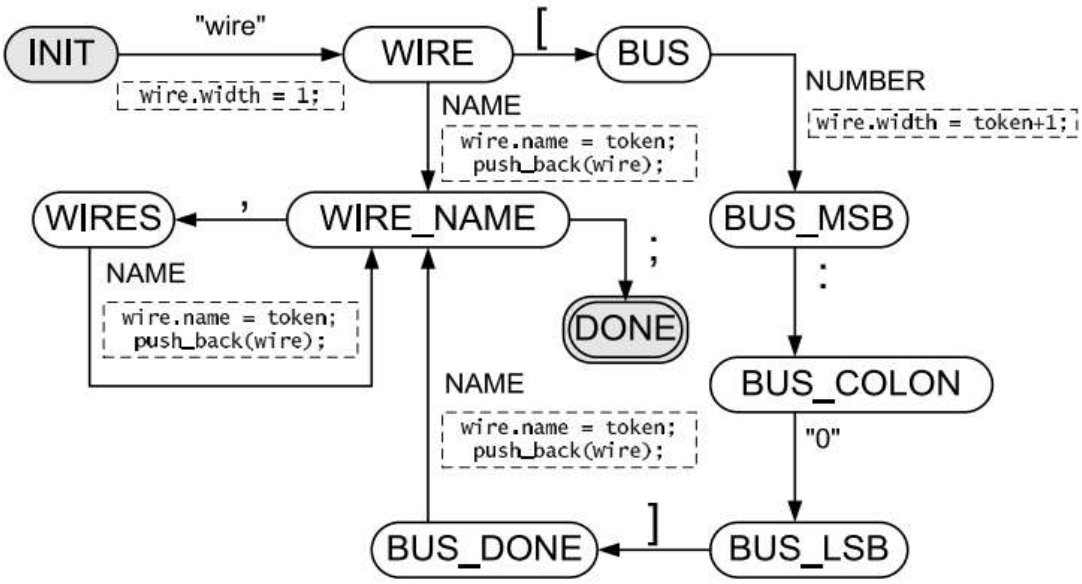


Fig1: FSM for WIRE statements

- **Component Module:** A component module sequence starts with a NAME token which is not among module, wire, or endmodule, and ends with a SINGLE token '**;**'. The component type has to be defined according to test.evl file. If

the name is not specified, it is considered as NONE by default. Pins that are defined within the component module, is the associated criteria for its pin configuration. Pins names supposed to be similar to wire name. FSM for the component module design is defined in Fig2.

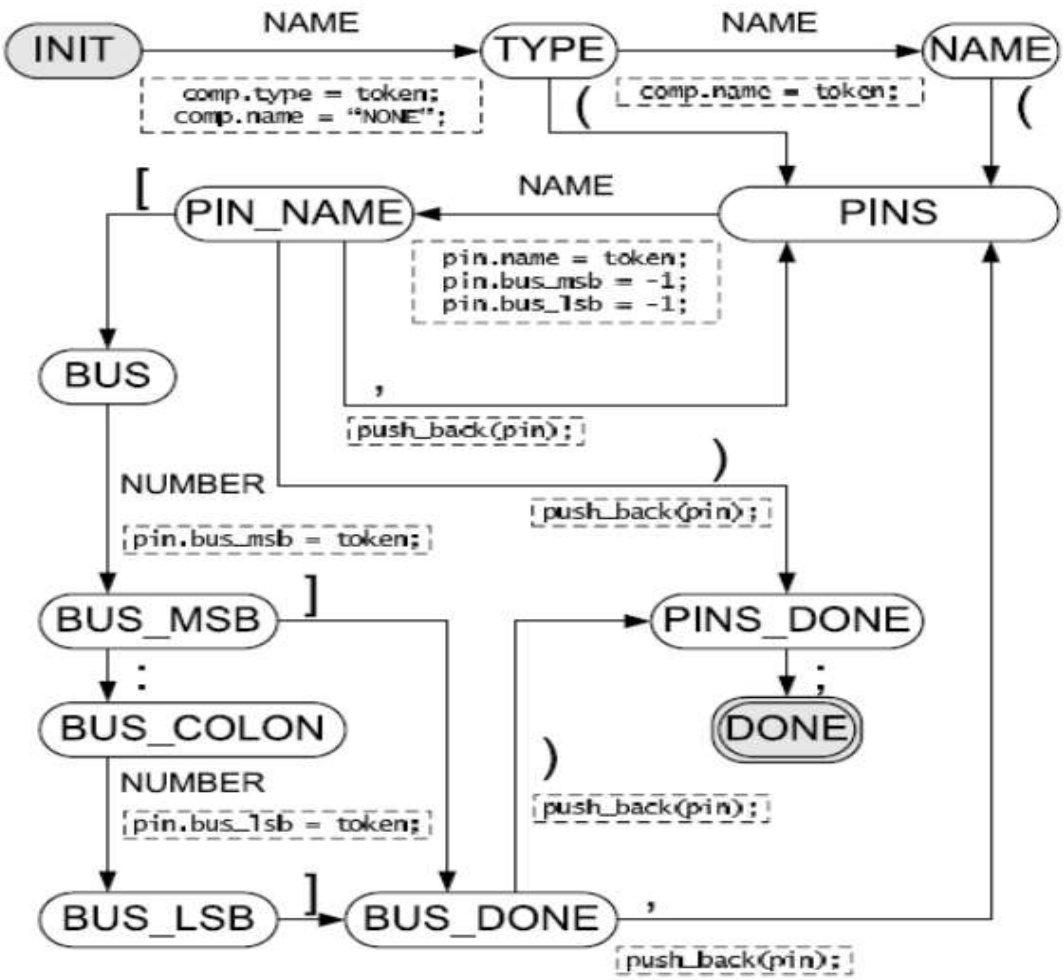


Fig2: FSM for COMPONENT module

- **Pin Module:** Each of the pins that are defined are associated with the respective components. The criterion for the implementation of pin module is as follows.
  - The name of the pin and wire should be the same.
  - If the wire is not a bus (width is 1), then both MSB and LSB of the pin must be -1 as 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 (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, the condition width > MSB >=LSB >= 0 must be true. If the MSB is specified but the LSB is not (LSB is -1), the condition width > MSB >= 0 must be true. The LSB should be updated to be the same as the MSB, since the pin refers to a single bit within the bus.

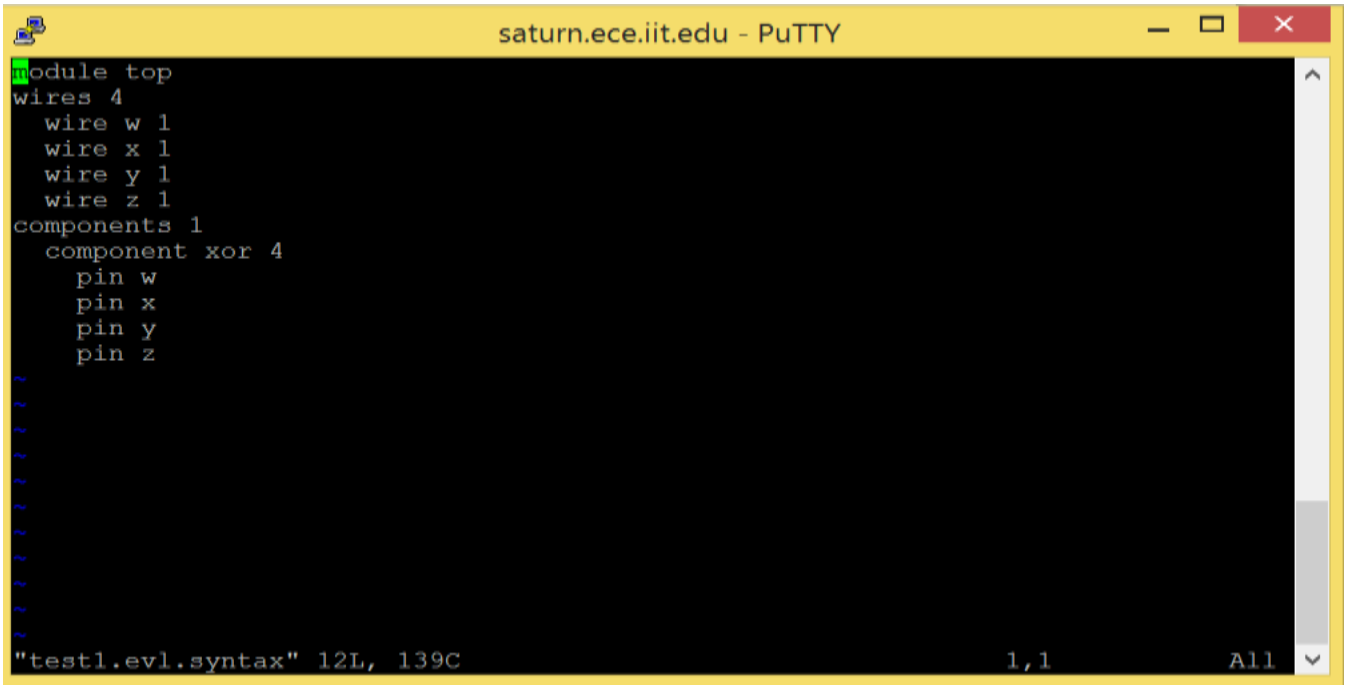
CONCLUSION

Syntactic analysis program that is written in C++ takes the name of the EasyVL file as the second command-line argument. After the file is processed to obtain tokens, the tokens are stored into an output file so that we can extract those and group tokens into statements. Then the output file is used to group tokens into statements .Then statements are analyzed to be processed one by one. The screenshots of the testcases and its outputs have been attached below.

Test Cases and its Outputs

1. Testcase 1

Output of Testcase 1



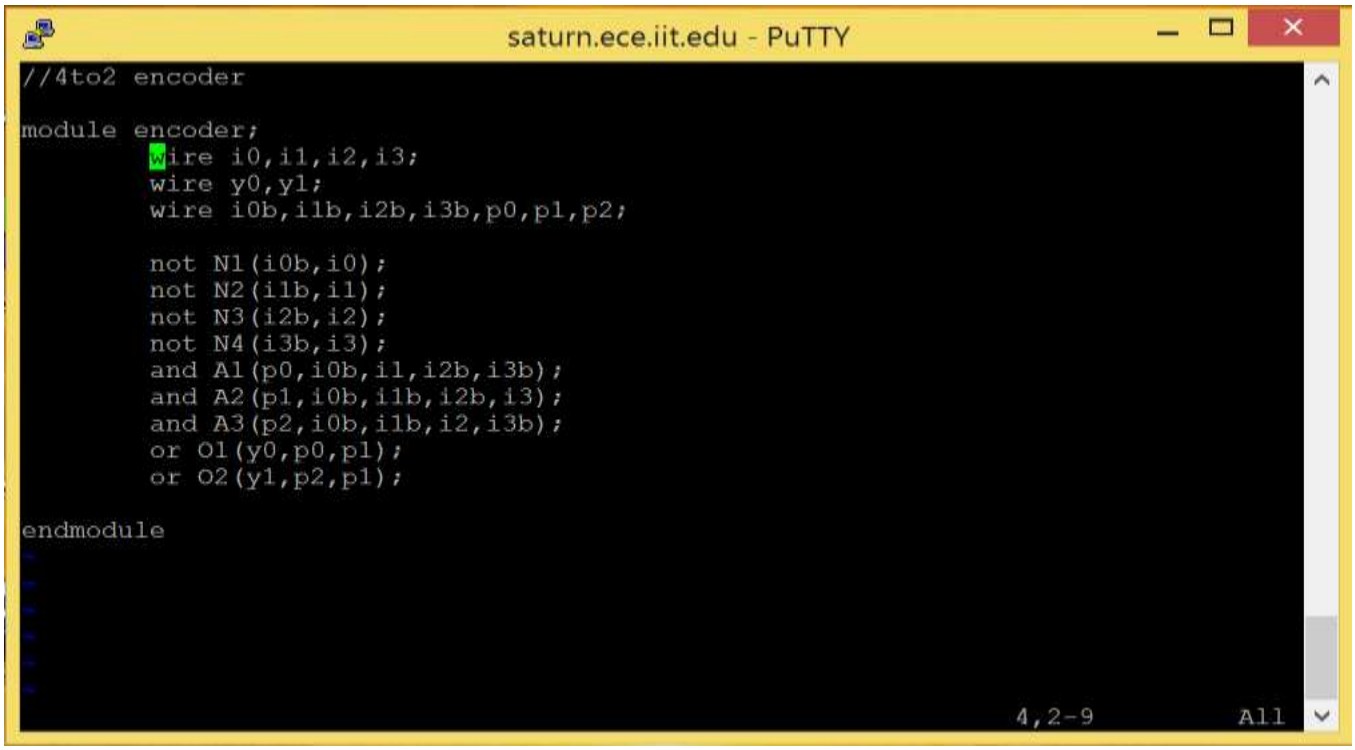
2. Testcase 2

```
//2:1 mux

module mux(a,b,sel,y);
        wire a,b,s;
        wire sbar,a1,b1;
        wire y;
        not N(sbar,s);
        and A1(a1,a,sbar);
        and A2(b1,b,s);
        or out(y,a1,b1);

endmodule

~
~
~
~
~
~
~
~
~
~
"test2.evl" 13L, 158C                                   6,2-9         All
```

Output of Testcase 2

```
module mux
wires 7
  wire a 1
  wire b 1
  wire s 1
  wire sbar 1
  wire a1 1
  wire b1 1
  wire y 1
components 4
  component not N 2
    pin sbar
    pin s
  component and A1 3
    pin a1
    pin a
    pin sbar
  component and A2 3
    pin b1
    pin b
    pin s
  component or out 3
    pin y
    pin a1
    pin b1
                                                            1,1           All
```

3. Testcase 3



Output of Testcase 3