ECE271, Chapter 4 Reading Report

WeiHao Kuang

May 14, 2018

1 Chapter Outline

1. Introduction

   In the world of hardware description languages (HDL) there are two main languages that are in use; the first one is System Verilog, and the second one is VHDL. Most of the principles between System Verilog and the VHDL are the same; the main difference between the two is their syntax. Both System Verilog and VHDL are widely used today in the field and there are many benefits of using a HDL to design a circuit compared to a schematic drawing tool, the only downside of using an HDL is the learning curve but once the curve is mastered; designing and debugging become much more efficient than using a schematic drawing program, such as Lattice Diamond's schematic builder.

2. Combination Logic

   Combinational circuits can be designed using a HDL; using an HDL to design a combinational circuit is fairly straightforward, but there are a couple of things to consider. When designing a circuit using a HDL one must consider the behavioral and structural aspects of the circuit; behavioral being the relationship between input and output and structural being how the logic blocks, known as modules, are comprised of other simpler modules. Using a HDL to make combinational circuits require the user to be familiar with the language's syntax and the functions in language; what the language is capable of doing versus what it can not do. In order to get the code to represent and behave like an actual circuit the designer must familiarize themselves the how the numbering works in language, what the precedence of operations are, they must be able to understand what internal variables (variables that can only be changed by modules in the circuit) are, versus external variables (variables that are directly changed by external sources) and most importantly When using any programing language the user must know how to write code that is readable and incorporates useful comments explaining the functions of code.

3. Structural Modeling

   Structural modeling is exactly how it sounds, this type of modeling deals with how a module in a circuit is comprised of other smaller modules that split a complex task into smaller, more manageable parts. Breaking down the complex modules of a circuit will make it easier to describe the behavior in HDL; Modulating complex modules makes code management a lot easier since the readability and debuggability of the code increase as the code becomes less bundled together.

4. Sequential Logic

   Sequential circuits can also be designed using HDLs as well, but compared to designing combinational circuits, sequential circuit design is more complex. Although designing sequential circuits proves to be more tasking, it is also is much more interesting since there are lots of things that sequential circuits can do that are very similar to functions of computer. Using either System Verilog or VHDL a designer can easily design a complex sequential circuit that would be nearly impossible to make on a schematic drawing program. HDL code that represents a latch, a D-latch, a register, or a flip-flip takes no more than a 20 lines of code each, whether the coding language be in System Verilog or VHDL. This demonstrates the efficiency of HDLs when compared to schematic drawing programs such as Lattice Diamond's schematic drawer.

5. More Combinational Logic

   There are few more functionalities of HDLs that can be used to make more complex and efficient combination circuits, functionalities such as the *always/process* statement in system Verilog and VHDL respectively. The *always/process* statement usually describes sequential logic, but if used correctly where all the outputs respond only to the inputs. In doing so the *always/process* statement can then be used to describe combinational logic. Along with the *always/process* statement, case statements and if statements can be used to create multi-level logic that increases the capabilities of a circuit, while simultaneously increasing the complexity as well. There are also two more types of assignment in HDLs: non-blocking and the blocking. Non-blocking assignment in HDLs refers to assignments that are processed simultaneously, and blocking assignment are processed in chronological order just like other programming languages. When all of these functionalities of the HDLs come to light, one can the how much more powerful of a design tool HDLs are compared to schematic drawing.

6. Finite State Machines

   Using HDLs designers can code the functions and the structure of a finite state machines (FSM). FSMs that are coded can be translated into block and state transition diagrams using Synplify Premier tools; because only a block diagram is made designers need to be extra cautious about whether they correctly coded their FSM, since the block diagram will not show inputs and outputs on arcs and states.

7. Data Types*

   Since the syntax of System Verilog and VHDL widely differ, their data types will differ as well. In general the syntax rules in System Verilog are less restrictive when compared to VHDL that is because Unlike System Verilog, VHDL is very strict about their data typing system, which protects the users from committing serious errors in the code, but since there are so many rules to follow VHDL is often at times very cumbersome compared to System Verilog. System Verilog on the other hand uses the *logic* data type very loosely where nearly all signals can be *logic*, the only exception is when signals have multiple drivers, when this happens the signals must be declared as net *wire or tri.* On the other hand, VHDL explicitly brings in libraries that have pre-built functions for operations that the required for a task, whilst System Verilog does that implicitly in the code. When comparing the two HDLs System Verilog seems more modern, and VHDL out and cumbersome, especially when comparing side-by-side code of the same module that is coded in System Verilog and VHDL.

8. Parameterized Modules*

   Modules that are are written with variable-width inputs and outputs are known are parameterized modules. Parameterization of the modules make implementing things that require large amounts of case statements or conditionals to be easily written. An example of parameterization of a module would be making a $N : 2^N$ decoder, by setting the appropriate output bits to $'1'$, which get rid of requiring N amount of case statements that would be in normal modules without parameterizing. Having parameterized modules modulates the code making it easier to read and the debug.

9. Testbenches

   Modules that made and used to test out another module are know as testbenches. The module that is being tested is known as the *device under test* (DUT). A testbench can be as simple as just inputing certain signals at given intervals, or they can be as complex as self-checking testbenches, whose functions changes according to the DUT. Testbenches are a nice addition to the debugging tool bench, since it can be written to suite the needs of the module to be tested it makes testbenches very versatile and easy to test modules with.

10. Summary

    HDLs are very power tools that can be used for synthesis and simulation of digital circuits, they are much more efficient for designing a digital circuits, when compared to the cumbersome schematic drawing programs, such as Lattice Diamond's Schematic builder. HDLs make editing and debugging designs less of a hassle since the design is in code, where changes are just code edits.HDL simulations of the of the design can check values inside the system itself, which can not be done physically.

2 Grey Box Exploration

1. The first blurb is on page 175, *The term "bug" predates the invention of the computer. Thomas Edison called the "little faults and difficulties" with his inventions "bugs" in 1878. The first real computer bug was a moth, which got caught between the relays of the Harvard Mark II electromechanical computer in 1947. It was found by Grace Hopper, who logged the incident, along with the moth itself and the comment "first actual case of bug being found."*

   (a) Bugs are inevitable when it comes down to designing digital circuits, prototype circuits may not alway behave the way they were intended too, but when the final product hits the market they better be. Not too long ago, at the start of 2018 news about two devastating bugs many computer chips (namely CPUs) were announce to the general public, these bugs had to potential to allow attackers to get access to data previously considered completely protected. All of the variants of this underlying vulnerability involve a malicious program gaining access to data that it shouldn't have the right to see; the information that was deemed private was no longer so because of these vulnerabilities. The crazies part of this is that these bugs date back 20 years, and any chip that was manufactured within the last 20 years would have been affected. 20 years unnoticed is really surprising, since testing on chips is very rigorous; the fact that this slipped, and for 20 years is really scary. This puts into perspective how important simulation and testing is for circuits so that bugs like this do not show up when a product is in mass production. [1][2][3]

2. The second blurb is on page 220, *Some tools also call the module to be tested the unit under test (UUT).*

   (a) A unit under test (UUT) is the object that is being tested. It is also known as device under test (DUT).The connection points are connected with test points of system. DUT mentioned a UUT or DUT can usually be tested in a simple testbench, that testbench can be as simple a stimulating the the circuit with value at certain times to test for the outputs of the circuit, or the testbench that is used is self-checking with changes the testing parameters based on the the values of the DUT/UUT such advance testbenches are overkill for simple circuits, but for large and complex circuits it is a must, especially if that large and complex circuits has changes to the code very often, having a self-checking testbench would make the debugging process a lot more simpler since very little changes to testbench needs to be made in order for it to test the new code.[4][5]

3 Figures

Two figures have been chosen from the book because they are effective at illustrating a concept that was discussed in chapter 4. Figure 1, (Figure 4.20 from the book) was selected because it shows the synthesized version of the 7-segment display decoder (used in ECE 272 Lab 4) which was very helpful for understanding how the code for the 7-segment display driver in Verilog translated directly into a block diagram. Seeing the code be translated into a block diagram, made it easier to think of HDLs as more of a description for connecting hardware, rather than just another programming language being able to think about how the hardware will be connected makes the code writing a bit more easy than is one were to think of HDLs only as a programming language and not considering the hardware that will be used.

Figure 2 (Figure 4.25 from the book) was selected because it show the power of Synplify Premier, and the weakness at the same time. The synthesized circuits throughout the book are only block diagrams that do not have the schematic of the actual logic gates that are involved in making the module that will have the functions described in the code. This good and bad at the same time, the being there will be less screen clutter the and block diagrams are much easier to understand compared to the actual schematics with logic gates. The bad part about only having block and state transition diagram is that there error checking cannot be done by simple inspection of the diagram, since the logic gates can be seen, the one way to counter this downside, is to run simulations, and the be extra careful when writing the code.

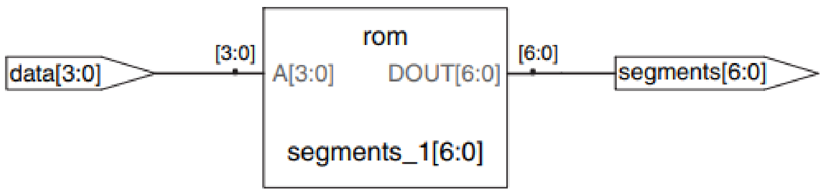


Figure 4.20 `sevenseg` **synthesized circuit**

Figure 1: Synthesized 7-segment Display Decoder

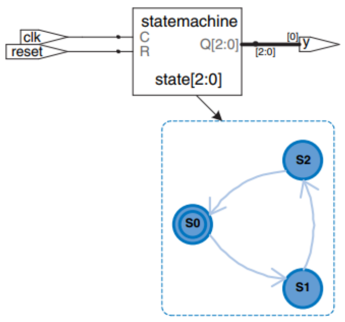


Figure 4.25 `divideby3fsm` **synthesized circuit**

Figure 2: Time graphs/diagram for short, general, and critical paths

4 Example Problems

Example problems are attached at the end.

5 Glossary

These terms where all obtained from Google, by searching up the term.

1. endianness

   refers to the sequential order used to numerically interpret a range of bytes in computer memory as a larger, composed word value. It also describes the order of byte transmission over a digital link.

2. synthesis

   noun:

   1. combination or composition, in particular; the combination of ideas to form a theory or system.

3. idiom

   noun:

   1. a group of words established by usage as having a meaning not deducible from those of the individual words (e.g., rain cats and dogs, see the light).

   2. a characteristic mode of expression in music or art.

4. delay

   verb:

   1. make (someone or something) late or slow.

   noun:

   1. a period of time by which something is late or postponed.

5. parameterize

   Verb:

   1. describe or represent in terms of a parameter or parameters.

6. bugs

   noun:

   1. a small insect.

   2. (entomology) an insect of a large order distinguished by having mouth-parts that are modified for piercing and sucking.

   3. a miniature microphone, typically concealed in a room or telephone, used for surveillance.

   4.an error in a computer program or system

6 Interview Question

Question 4.2 Explain the difference between blocking and nonblocking assignments in SystemVerilog. Give examples.

Figure 3: Blocking vs Non-blocking

There are two types of assignment in most HDLs languages; first one being blocking assignment and the second being non-blocking assignments. The fundamental difference between blocking and non-blocking assignments is that for blocking assignments, the code is read in chronological order like statements in other languages, such as C/C++ or Python. One of the reason that blocking assignments are used when the order of the assignments matter in terms of the output of the circuit, blocking statements will keep the assignments in order so that the process order dependent output will not get disrupted and also if a value needs to be assigned the value is directly assigned in one clock cycle, as opposed to N cycles, N being however many lines it takes to propagate a value to its destination for non-blocking assignments. Non-blocking assignments are assignments that are are processed in parallel. These statements are usually denote by an arrow with and equals sign: "<=". While blocking assignments are usually denote by just an "=" sign. [6][7][8]

Blocking Assignment Example:

/*pretest 2 will get pretest1 in one clock cycle since everything is executed in order*/
$pretest1 = 1'b1;$
$pretest2 = pretest1;$

Non-blocking assignment example:

/*pretest 2 will get pretest1 in 2 clock cycles since there are two line of code that will be executed at the same time*/
$pretest1 <= 1'b1;$
$pretest2 <= pretest1;$

7 Reflection

Lots of things where discussed in this chapter namely information about hardware description languages (HDL). The two main types of HDLs that are used in the industry as of today is System Verilog and VHDL. It was very interesting to see the side-by-side comparisons of System Verilog and VHDL code modules, having two languages to compare makes each language standout, in terms of the syntax. From the example code I could see that one of the things for VHDL is that everything is done explicitly, almost like C, the code takes in pre-built libraries and uses them for operations that are crucial for tasks in the module, but on the other hand System Verilog just use operation symbols and everything is done more implicitly, like C++. One of the things that I loved about the chapter was having the synthesized circuits, these figures were very helpful for learning about how the code turns into block diagrams and "schematics". Being able to make the connection to hardware from code was very interesting to me since I first approached HDLs like it was another programming language. As I started to think about the code and modeling some of them on ModelSim I slowing began to realize that I should start visualizing or drawing out the block diagrams for the code, that way I would be able to understand what the code is say without having to run a synthesis tool on it every time I need to see a block diagram for a module. The thing I had the most trouble understanding was the syntax of VHDL, for some reason I just had a hard time familiarizing myself with how different and more explicit compared to the System Verilog code that we have been introduced to in ECE 272 lab. Another thing that I don't quite understand it the one of th blurbs in the book, it was on page 211, I could not find why a 3-bit encoder is used in a synthesis tool rather than a 2-bit encoder as suggested in the code. The only guess that I have as to why a 3-bit encoder is used instead of a 2-bit is because having a bigger bus size will act as safety net just in case a 2-bit bus is not enough to complete the synthesis. Overall I thought this chapter was a very interesting and insightful. Some of the code and figures helped understand how HDLs work and the how they apply in the logic design. This chapter helped me understand the value HDLs in the field of digital logic design.

8 Questions for Lecture

1. What are some of the other types of the HDLs that are used the industry, or is just VHDL and System Verilog?

2. What is the best way to gain a better understanding of how System Verilog works, since the book only covered the basics.

3. VHDL seems very archaic and cumbersome to me when compared to System Verilog, is there any specific instances that VHDL should be used instead of System Verilog?

4. In on of the blurbs it mentions that a synthesizer used in to make the circuits had a 3-bit encoder instead of a 2-bit encoder, what is the reason to use a 3-bit encoder over a 2-bit encoder for a synthesis tool?

References

[1] J. Fruhlinger, "Spectre and meltdown explained: What they are, how they work, what's at risk." https://www.csoonline.com/article/3247868/vulnerabilities/spectre-and-meltdown-explained-what-they-are-how-they-work-whats-at-risk.html, 2018.

[2] A. Berke, "How early were spectre and meltdown discovered?." https://medium.com/freeman-spogli-institute-for-international-studies/how-early-were-spectre-and-meltdown-discovered-80f8763d5a3c, 2018.

[3] D. A. Wong, "Complete list of cpus vulnerable to meltdown / spectre rev. 8.0." https://www.techarp.com/guides/complete-meltdown-spectre-cpu-list/.

[4] Weetech, "Unit under test (uut)." http://www.weetech.de/en/news-info/tester-abc/unit-under-test-uut/.

[5] S. Capkun and F. K. Gürkaynak, "Using verilog for testbenches." http://www.syssec.ethz.ch/content/dam/ethz/special-interest/infk/inst-infsec/system-security-group-dam/education/Digitaltechnik_14/14_Verilog_Testbenches.pdf.

[6] A. Fox, "Blocking and nonblocking in verilog." http://www.asic-world.com/tidbits/blocking.html, 2014.

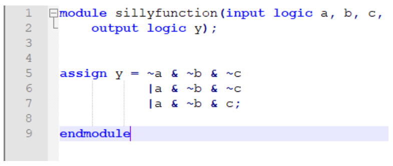
[7] N. (forum), “Difference between blocking and nonblocking assignment verilog.” https://electronics.stackexchange.com/questions/91688/difference-between-blocking-and-nonblocking-assignment-verilog.

[8] NANDLAND, “Blocking vs. nonblocking in verilog.” https://www.nandland.com/articles/blocking-nonblocking-verilog.html, 2014.

Example Problem: Simulation and Synthesis Using Model Sim and Synplify Pro

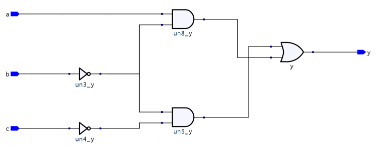
May 14, 2018

1 System Verilog Example 4.1

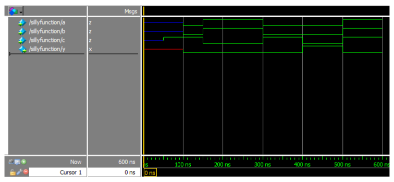


```
module sillyfunction(input logic a, b, c,
     output logic y);


assign y = ~a & ~b & ~c
             |a & ~b & ~c
             |a & ~b & c;

endmodule
```

(a) System Verilog source

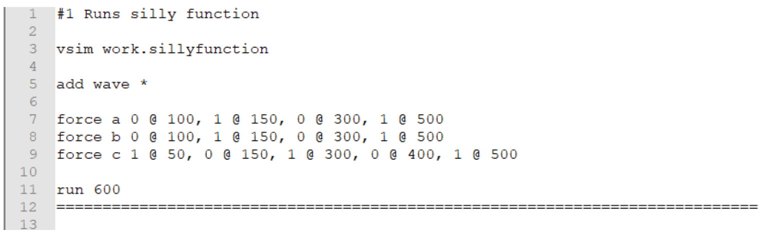


(b) SynplifyPro synthesis



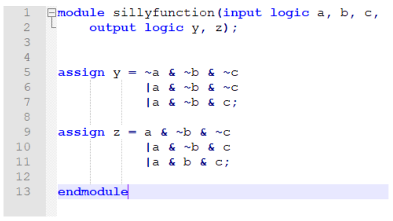
(c) ModelSim simulation

Figure 1: Example 4.1



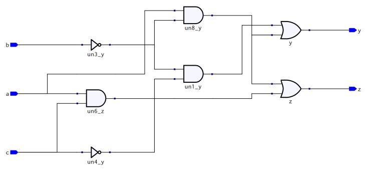
```
#1 Runs silly function

vsim work.sillyfunction

add wave *

force a 0 @ 100, 1 @ 150, 0 @ 300, 1 @ 500
force b 0 @ 100, 1 @ 150, 0 @ 300, 1 @ 500
force c 1 @ 50, 0 @ 150, 1 @ 300, 0 @ 400, 1 @ 500

run 600
============================================================================

```

Figure 2: This do file can be used to simulate both example 4.1 and the variation within ModelSim.

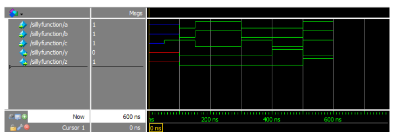


```
module sillyfunction(input logic a, b, c,
     output logic y, z);


assign y = ~a & ~b & ~c
             |a & ~b & ~c
             |a & ~b & c;

assign z = a & ~b & ~c
             |a & ~b & c
             |a & b & c;

endmodule
```

(a) System Verilog source



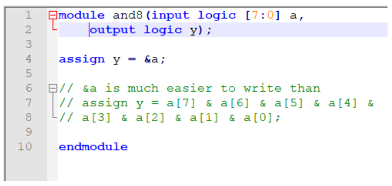
(b) SynplifyPro synthesis



(c) ModelSim simulation

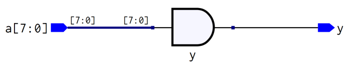
Figure 3: System Verilog Variation of Example 4.1

2 System Verilog Example 4.4

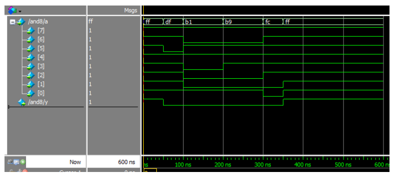


```
module and8(input logic [7:0] a,
     output logic y);

assign y = &a;

// &a is much easier to write than
// assign y = a[7] & a[6] & a[5] & a[4] &
// a[3] & a[2] & a[1] & a[0];

endmodule
```

(a) System Verilog source

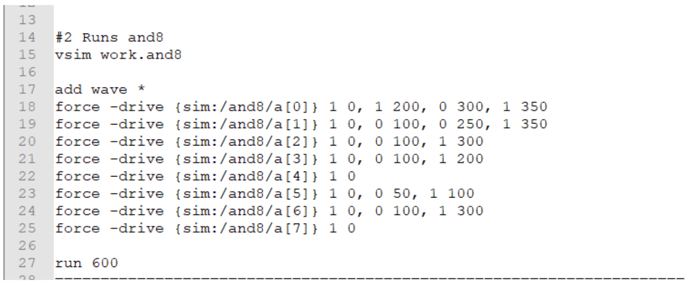


(b) SynplifyPro synthesis



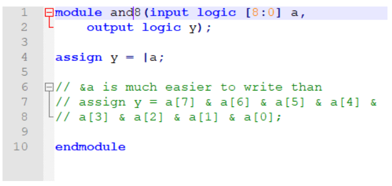
(c) ModelSim simulation

Figure 4: System Verilog Example 4.4



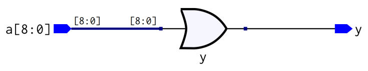
```
#2 Runs and8
vsim work.and8

add wave *
force -drive {sim:/and8/a[0]} 1 0, 1 200, 0 300, 1 350
force -drive {sim:/and8/a[1]} 1 0, 0 100, 0 250, 1 350
force -drive {sim:/and8/a[2]} 1 0, 0 100, 1 300
force -drive {sim:/and8/a[3]} 1 0, 0 100, 1 200
force -drive {sim:/and8/a[4]} 1 0
force -drive {sim:/and8/a[5]} 1 0, 0 50, 1 100
force -drive {sim:/and8/a[6]} 1 0, 0 100, 1 300
force -drive {sim:/and8/a[7]} 1 0

run 600
--------------------------------------------------------------------
```

Figure 5: This do file can be used to simulate both example 4.4 and the variation within ModelSim.

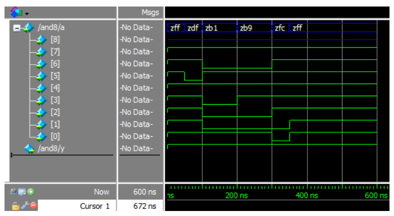


```
module and8(input logic [8:0] a,
     output logic y);

assign y = |a;

// &a is much easier to write than
// assign y = a[7] & a[6] & a[5] & a[4] &
// a[3] & a[2] & a[1] & a[0];

endmodule
```

(a) System Verilog source



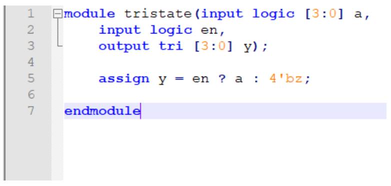
(b) SynplifyPro synthesis



(c) ModelSim simulation

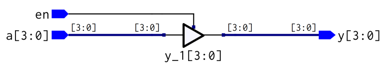
Figure 6: System Verilog Variation Example 4.4

3 System Verilog Example 4.10

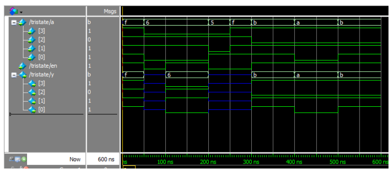


```
module tristate(input logic [3:0] a,
    input logic en,
    output tri [3:0] y);

    assign y = en ? a : 4'bz;

endmodule
```

(a) System Verilog source

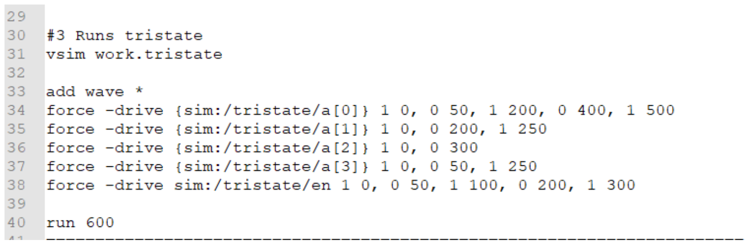


(b) SynplifyPro synthesis



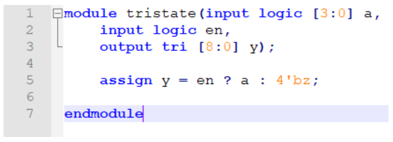
(c) ModelSim simulation

Figure 7: System Verilog Example 4.10



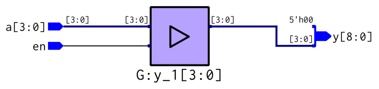
```
#3 Runs tristate
vsim work.tristate

add wave *
force -drive {sim:/tristate/a[0]} 1 0, 0 50, 1 200, 0 400, 1 500
force -drive {sim:/tristate/a[1]} 1 0, 0 200, 1 250
force -drive {sim:/tristate/a[2]} 1 0, 0 300
force -drive {sim:/tristate/a[3]} 1 0, 0 50, 1 250
force -drive sim:/tristate/en 1 0, 0 50, 1 100, 0 200, 1 300

run 600
------------------------------------------------------------------------
```

Figure 8: This do file can be used to simulate both example 4.10 and the variation within ModelSim.

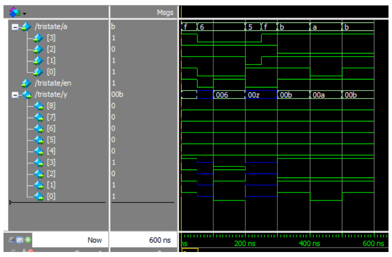


```
module tristate(input logic [3:0] a,
    input logic en,
    output tri [8:0] y);

    assign y = en ? a : 4'bz;

endmodule
```

(a) System Verilog source



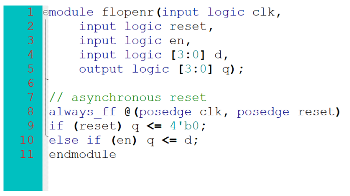
(b) SynplifyPro synthesis



(c) ModelSim simulation

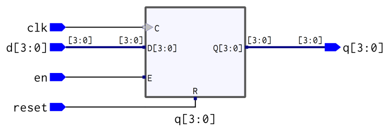
Figure 9: System Verilog Variation Example 4.10

4 System Verilog Example 4.19

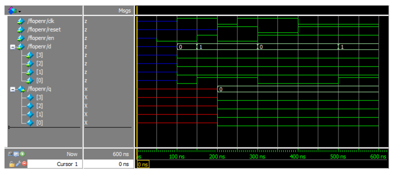


```
module flopenr(input logic clk,
    input logic reset,
    input logic en,
    input logic [3:0] d,
    output logic [3:0] q);

// asynchronous reset
always_ff @(posedge clk, posedge reset)
if (reset) q <= 4'b0;
else if (en) q <= d;
endmodule
```

(a) System Verilog source

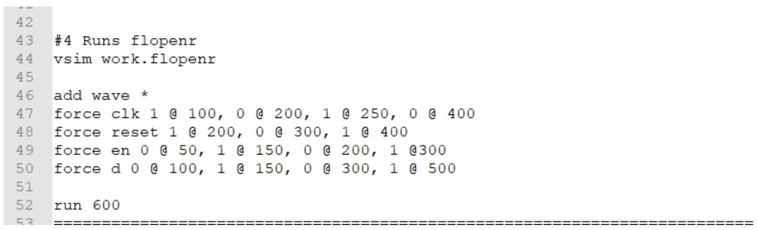


(b) SynplifyPro synthesis



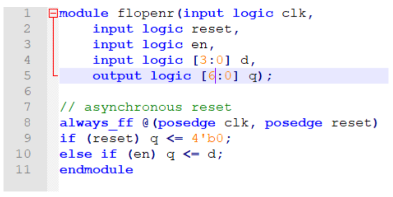
(c) ModelSim simulation

Figure 10: System Verilog Example 4.19



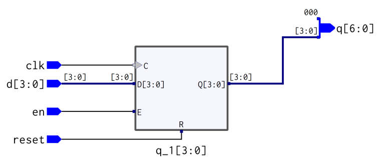
```

#4 Runs flopenr
vsim work.flopenr

add wave *
force clk 1 @ 100, 0 @ 200, 1 @ 250, 0 @ 400
force reset 1 @ 200, 0 @ 300, 1 @ 400
force en 0 @ 50, 1 @ 150, 0 @ 200, 1 @300
force d 0 @ 100, 1 @ 150, 0 @ 300, 1 @ 500

run 600
==========================================================================
```

Figure 11: This do file can be used to simulate both example 4.19 and the variation within ModelSim.

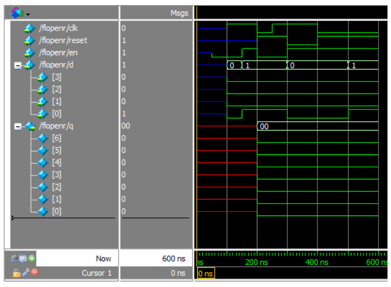


```
module flopenr(input logic clk,
    input logic reset,
    input logic en,
    input logic [3:0] d,
    output logic [6:0] q);

// asynchronous reset
always_ff @(posedge clk, posedge reset)
if (reset) q <= 4'b0;
else if (en) q <= d;
endmodule
```

(a) System Verilog source



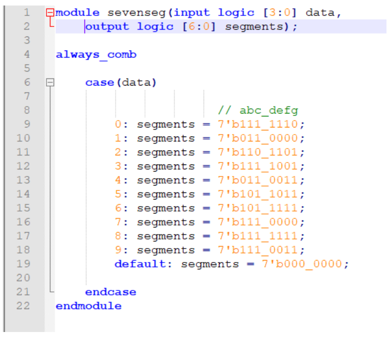
(b) SynplifyPro synthesis



(c) ModelSim simulation

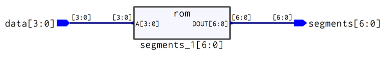
Figure 12: System Verilog Variation HDL Example 4.19

5 System Verilog Example 4.24

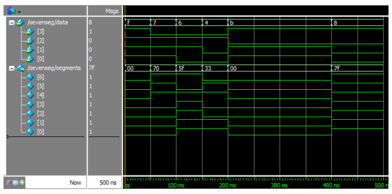


```
module sevenseg(input logic [3:0] data,
    output logic [6:0] segments);

always_comb

    case(data)

                    // abc_defg
        0: segments = 7'b111_1110;
        1: segments = 7'b011_0000;
        2: segments = 7'b110_1101;
        3: segments = 7'b111_1001;
        4: segments = 7'b011_0011;
        5: segments = 7'b101_1011;
        6: segments = 7'b101_1111;
        7: segments = 7'b111_0000;
        8: segments = 7'b111_1111;
        9: segments = 7'b111_0011;
        default: segments = 7'b000_0000;

    endcase
endmodule
```

(a) System Verilog source

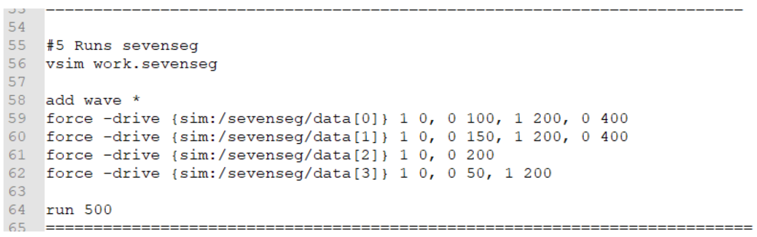


(b) SynplifyPro synthesis



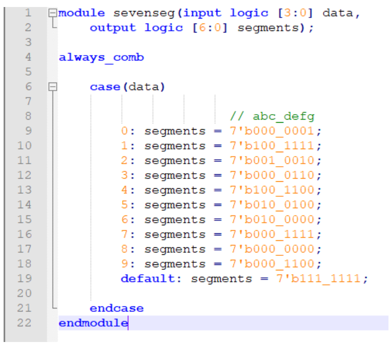
(c) ModelSim simulation

Figure 13: System Verilog Example 4.24



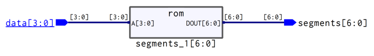
```
------------------------------------------------------------------------

#5 Runs sevenseg
vsim work.sevenseg

add wave *
force -drive {sim:/sevenseg/data[0]} 1 0, 0 100, 1 200, 0 400
force -drive {sim:/sevenseg/data[1]} 1 0, 0 150, 1 200, 0 400
force -drive {sim:/sevenseg/data[2]} 1 0, 0 200
force -drive {sim:/sevenseg/data[3]} 1 0, 0 50, 1 200

run 500
========================================================================
```

Figure 14: This do file can be used to simulate both example 4.24 and the variation within ModelSim.

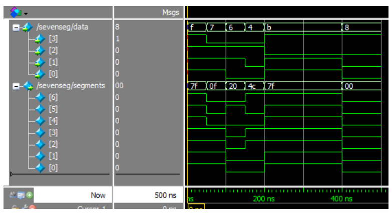


```
module sevenseg(input logic [3:0] data,
    output logic [6:0] segments);

always_comb

    case(data)

                    // abc_defg
        0: segments = 7'b000_0001;
        1: segments = 7'b100_1111;
        2: segments = 7'b001_0010;
        3: segments = 7'b000_0110;
        4: segments = 7'b100_1100;
        5: segments = 7'b010_0100;
        6: segments = 7'b010_0000;
        7: segments = 7'b000_1111;
        8: segments = 7'b000_0000;
        9: segments = 7'b000_1100;
        default: segments = 7'b111_1111;

    endcase
endmodule
```

(a) System Verilog source



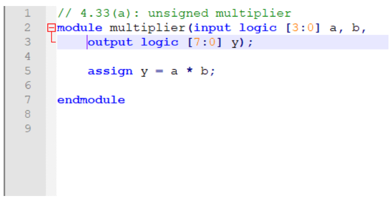
(b) SynplifyPro synthesis



(c) ModelSim simulation

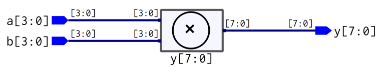
Figure 15: System Verilog Variation Example 4.24

6 System Verilog Example 4.33

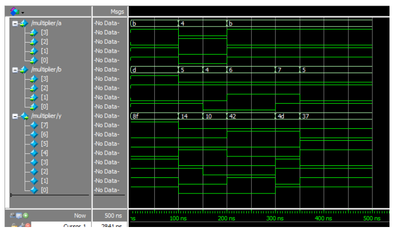


```
// 4.33(a): unsigned multiplier
module multiplier(input logic [3:0] a, b,
   output logic [7:0] y);

   assign y = a * b;

endmodule
```

(a) System Verilog source

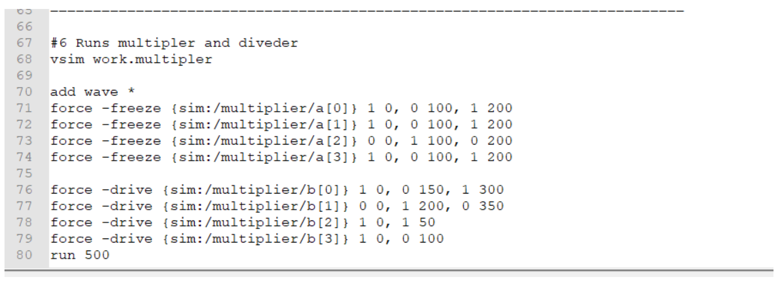


(b) SynplifyPro synthesis



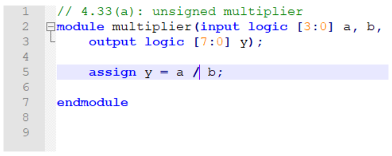
(c) ModelSim simulation

Figure 16: System Verilog Example 4.33



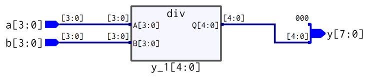
```
--------------------------------------------------------------------------

#6 Runs multipler and diveder
vsim work.multipler

add wave *
force -freeze {sim:/multiplier/a[0]} 1 0, 0 100, 1 200
force -freeze {sim:/multiplier/a[1]} 1 0, 0 100, 1 200
force -freeze {sim:/multiplier/a[2]} 0 0, 1 100, 0 200
force -freeze {sim:/multiplier/a[3]} 1 0, 0 100, 1 200

force -drive {sim:/multiplier/b[0]} 1 0, 0 150, 1 300
force -drive {sim:/multiplier/b[1]} 0 0, 1 200, 0 350
force -drive {sim:/multiplier/b[2]} 1 0, 1 50
force -drive {sim:/multiplier/b[3]} 1 0, 0 100
run 500
```

Figure 17: This do file can be used to simulate both example 4.33 and the variation within ModelSim.

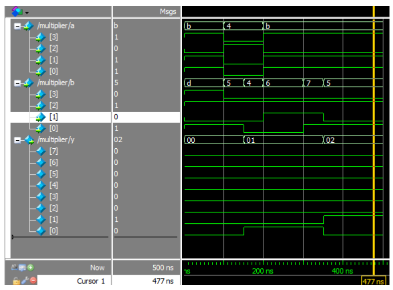


```
// 4.33(a): unsigned multiplier
module multiplier(input logic [3:0] a, b,
    output logic [7:0] y);

    assign y = a / b;

endmodule
```

(a) System Verilog source



(b) SynplifyPro synthesis



(c) ModelSim simulation

Figure 18: System Verilog Variation Example 4.33