# ECE 271, Chapter 5 Reading Report

WeiHao Kuang

May 28, 2018

## 1 Chapter Outline

1. Introduction

   In today's digital systems there are lots of moving parts; which include the modules that make up the combinational and sequential logic of the circuit. Often times these logic blocks are made up of smaller and simpler parts, that to their own right are black boxes themselves, these smaller blocks are useful for demonstrating hierarchy, modularity, and regularity withing a circuit design. Having a strong understanding in the basic building blocks of digital circuits will make designing and debugging a lot more efficient, due to how the digital circuits are designed today.

2. Arithmetic Circuits

   Arithmetic circuits are essential for building computers, there block are responsible for carrying out the many arithmetic operations that are done in computers such as: addition, subtraction, comparisons, shifts, multiplication, and division. When it comes to implementing an operation such as addition there are three common ways to implementation a carry propagate adder: ripple carry adder, carry-lookahead, and prefix adder. The speed of the adder implementations is in ascending order, where the prefix adder is the faster of the 3 common implementation. Of course, the faster the adder is the most power and hardware it takes to run, so depending on the need of the circuit an adder can be super fast and power hungry or it can be comparatively slower but take less hardware and power to run. As for subtraction implementing this is straightforward, the idea is really not to subtract but add the two's complement of a number $B$ to another $A$: $A + \overline{B} = A - B$ , therefore achieving the subtraction without using comparators. When the design wants to have more than one mathematical operation in one given block they can create what is known as an Arithmetic/Logical Unit (ALU), which can perform multiple operations, and is often the heart of most computer systems. One of mathematical operations that can be implemented are shifters and rotators which moves the index of the binary either to the left or two the right to a specified number of indexes, the 3 common used shifters are: logical shifters, arithmetic shifters, and rotators. Shifting is basically dividing (shift right) or multiplying (shift left) by $2^N$ (N is bits shifted). As mentioned earlier, arithmetic blocks can also divide and multiply, like other operation the speed is dependent one the implementation and the hardware that's used. The biggest take away about arithmetic logic blocks is that there are crucial for computers and that there will always be trade offs: faster calculations = more hardware/power used.

3. Number Systems

   The number system that is used in computers includes ints, whole numbers, and fractions, rational numbers both of which are important to the operation of the computer's processes. Both number types can be directly represented by 1's and 0's in their correct significance place. There are two ways to represent fraction in binary form: fixed-point notation and floating-point notation. The fixed point notation is analogous to the decimal point used in the number with whole bits and rational bits. In this form the bits before the decimal point represents the whole number and the bits after the decimal point represents the rational number. The other notation that can be used to present rational number is the floating-point notation, this notation is the like the scientific notation that is used in the decimal system, primarily used to represent very large/small, numbers due not having a fixed number of int and rational bits. The notation that is used in floating-point is very similar to the scientific

notation as well: $2.10 \times 10^2$ (scientific notation) and $1.1010010 \times 2^7$ (floating-point notation). Having a handle on the numbering system used in computers is really helpful for designing digital systems, or writing program.

4. Sequential Building Blocks

   Two commonly used building blocks in higher level sequential circuits are counters, and shift registers. Sequential logic counters are blocks that have the function as their name suggests: count. Counter cycle through $2^N$ possible outcomes in binary order, incrementing on the clock edge. Counters are used in almost all computers and digital systems, since they are the grounds for flip-flops. The other block that is commonly is used is a shift register; shift registers function by shifting bits of information to the end to of the pipe (formed by N-flip-flops), on every rising edge new bit is shifted in from $S_{in}$ and the last bit will be shifted to and held by $S_{out}$ in doing so this creates a block that can store inputs can process them in linear order based on which was inputted first, and example of this block in use is a console receiving button pushes from a wired or wireless controller. Using these smaller blocks larger sequential block can be broken down so that it is easier to understand.

5. Memory Arrays

   Digital systems require the used of memory to store the information that will be used and generated in the circuit. The biggest classification type for memory are volatile random access memory (RAM) and non-volatile read-only memory (ROM). There are three major types of memory arrays that are used toady: dynamic random access memory (DRAM), static random access memory (SRAM), and read only memory (ROM). Each memory type had their trade-offs and differences; as well as similarities. The things that are similar with all three types of memory arrays is that all memory has one or more ports that allow for read and/or write access to a memory address; The more ports the more addresses that can be read at the same time. In DRAM there the data is store on a capacitor, and an nMOS transistor the system can read and write values onto the capacitor. The contents on the capacitor are constantly refreshed because the charge weakens gradually as time passes. On the other hand SRAM does not store the information in a capacitor, but instead the on cross coupled inverters, which restore degraded values as well, if speeds where compared the fastest of volatile memory would flip-flops and the slowest would be the DRAM. As for ROM this type of memory gets its data during manufacturing and can not have its data overwritten, unless it is re-programmable ROM which has a mechanism that allows re-writing of data on the hardware, the principles that used for programmable ROMS (PROMS) can be applied to flash memory as well (only in flash memory there is extra circuitry that is used to perform the re-programming.

6. Logic Arrays

   Logic arrays are made when logic gates can be programmed by the user as to how there are connected with on another, and the connection as made completely in HDL and there will be an absence of having to connect wires in specific ways. More widely know application of logic arrays are programmable logic arrays (PLA) and field programmable logic arrays (FGPA); PLAs can only perform combinational logic, while FPGA can do both combinational and sequential logic. PLAs are made from two level combinational logic, that come from one AND array followed by an OR array. On the other hand FGPAs are built from a reconfigurable gate, that are not limited to just AND and OR's. Often times FGPA integrate multipliers, high-speed I/0s, analogy-to-digital converters (ADC), and processors; in general terms FGPAs are more robust compared to the PLAs, but they are more expensive and more complicated to use.

7. Summary

   There are many aspects to consider when designing or trying to understand a digital system some of the more important things to consider are arithmetic blocks which are in just about every computer, also it is worth understanding the numbering system used in that environment to avoid any unforeseen problems. The memory type that will be used in the circuit is important, because there are trade-offs that cause problems for a digital system if not considered carefully. And lastly FPGAs, and PLAs are a convenient way to make logic blocks without having to connect the wires manually, since everything is done in HDL.

## 2 Grey Box Exploration

1. The first blurb is on page 259, *Floating-point arithmetic is usually done in hardware to make it fast. This hardware, called the floating-point unit (FPU), is typically distinct from the central processing unit (CPU). The infamous floating-point division (FDIV) bug in the Pentium FPU cost Intel $475 million to recall and replace defective chips. The bug occurred simply because a look up table was not loaded correctly.*

   (a) Bugs are inevitable when it comes down to designing digital circuits, prototype circuits may not always behave the way they were intended too, but when the final product hits the market they better be. On October 19, 1994 a professor by the name of Thomas Nicely discovers a critical flaw in Intel's new Intel Pentium processor at the time. Although the issue was a called for a simple fix and didn't really affect the average everyday consumer, one chance in more than nine billion of encountering an inaccurate result as a consequence of the error. Yet, it snowballed in to a mess because of the national press coverage and which labelled the bug the first computer hardware problem to have made the headline worldwide, well before the Millennium Bug. Through this bug Intel dramatically improved our validation methodology to quickly capture and fix errata, and investigated innovative ways to design products that are error-free right from the beginning, and then overall improved their feedback reception services so that the communication between consumers and the company with strengthened. 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. Rigorously testing and validating will be enough to prevent all the bugs that may surface, but it certainly will prevent the game breaking bugs that can potentially end a company. [1][2][3]

2. The second blurb is on page 274, *FPGAs are the brains of many consumer products, including automobiles, medical equipment, and media devices like MP3 players. The Mercedes Benz S-Class series, for example, has over a dozen Xilinx FPGAs or PLDs for uses ranging from entertainment to navigation to cruise control systems. FPGAs allow for quick time to market and make debugging or adding features late in the design process easier.*

   (a) There are two main types of re programmable hardware that is discussed in this book: FGPAs and PLAs, The fundamental difference between an field-programmable array (FGPA) versus a programmable logic array (PLA) is that FPGAs are able to perform sequential logic and and combinational logic, whereas PLAs can only perform combinational logic since they are much older compared to FPGAs, around 10 years older to be precise. In general FPGAs are more robust and cost more than the average PLA since there is a lot more hardware on the FPGA compared to the PLA, which means FPGAs are more suited for complex logic functions since they can have up to 100,000 tiny logic block with in them. Things that a FPGA can provide over other hardware would be: Performance, Time to Market, Cost, Reliability, and Long-Term Maintenance. Reliability and Cost efficiency is very important when it comes to designing products and FGPA deliver on both, in terms of reliability FPGAs, which do not use OSs, minimize reliability concerns with true parallel execution and deterministic hardware dedicated to every task. In terms of cost FGPA are are most cheaper compared to ASIC solutions that out there, since customizing the FGPAs eliminates the the fabrication costs of associated with customized ASIC devices. Also the last thing that is worth mentioning is how easy it is to prototype an idea on an FGPA, all that is needed is HDL files to program the FGPA and voila there is working prototype of an idea, without having to go through the prototype fab process which takes extra time and money from the budget. The best thing about being able to prototype on the FGPA is that if there needs to be any changes it can be done directly via HDL modifications, rather than manually change wire connections. When it comes down to it FGPAs and PLAs are both very good options for hardware choices for projects. [4][5][6]

# 3 Figures

Two figures were selected for chapter 5 because they are great at illustrating their topic. Figure 5.15 was selected for the effectiveness at showing how the schematic of the an N-bit ALU, this was the figure cleared some of the confusing bits for the the ALU earlier on in the book, this schematic illustrates exactly how the ALU works, which is a lot better than just reading about it without understanding how everything interfaces with each other.

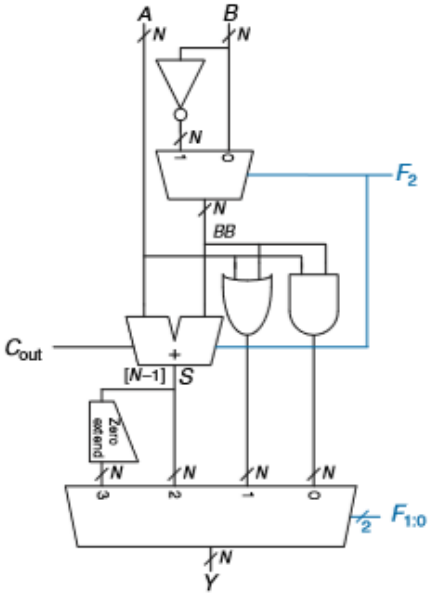



Figure 5.15 *N*-bit ALU

Figure 1: N-bit ALU schematic

Figure 5.42 was selected for highlighting how a simple block of memory works, the figure show a 4 by 3 array that can store up to 12 bits of information. This figure helps with visualizing how memory blocks work for the most part, the schematic shows that reader how the bits are stored and how the bitlines are connected with the stored bits and form data outputs.

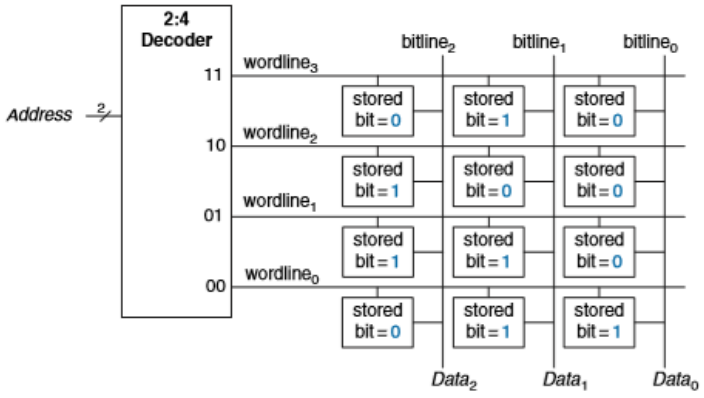



Figure 5.42 4 × 3 memory array

Figure 2: 4x3 Memory Array Illustration/Block Diagram

# 4 Example Problems

Example HDL code with simulation and written example problems are attached to the end of the page.

# 5 Glossary

All of the definitions are found from Google, by looking up the term in the defination searcher.

1. magnitude

   noun:

   1. the great size or extent of something.

   2. size

   3. the degree of brightness of a star. The magnitude of an astronomical object is now reckoned as the negative logarithm of the brightness; a decrease of one magnitude represents an increase in brightness of 2.512 times. A star with an apparent magnitude of six is barely visible to the naked eye.

2. flag

   noun:

   1. a piece of cloth or similar material emblem of a country or institution or as a decoration during public festivities.

   2. a device, symbol, or drawing typically resembling a flag, used as a marker

   verb:

   1. mark (an item) for attention or treatment in a specified way.

   2. direct (someone) to go in the specified direction by waving a flag or using hand signals.

3. overflow

   verb:

   1. (especially of a liquid) flow over the brim of a receptacle.

   noun:

   1. the excess or surplus not able to be accommodated by an available space.

   2. the generation of a number or some other data item that is too large for an assigned location or memory space.

4. mantissa

   noun:

   1. the part of a logarithm that follows the decimal point.

   2. the part of a floating-point number that represents the significant digits of that number, and that is multiplied by the base raised to the exponent to give the actual value of the number

5. static

   adjective:

   1. lacking in movement, action, or change, especially in a way viewed as undesirable or uninteresting.

   2. (of a memory or store) not needing to be periodically refreshed by an applied voltage

   3. concerned with bodies at rest or forces in equilibrium.

6. dynamic

   adjective:

   1. (of a process or system) characterized by constant change, activity, or progress.

# 6 Interview Question

**Question 5.2** *Binary coded decimal* (*BCD*) representation uses four bits to encode each decimal digit. For example, $42_{10}$ is represented as $01000010_{BCD}$. Explain in words why processors might use BCD representation.

Figure 3: Why Binary Coded Decimal?

The reasons the processor use binary coded decimal can be many factor, one of reason that the processor favors binary coded decimal (BCD) is because that processor is an older generation that favors BCD because of its ease to decode and encode. Another reason why a a processor might take a liking to BCD is due to how easy to scale a number by a factor of 10, which makes it useful when a non-integer quantity needs to be represented. And one of last reason that a processor might use BCD is that it is simple to implement a hardware algorithm for the BCD converter. It is very useful in digital systems whenever decimal information is given either as inputs or displayed as outputs.[7][8][9]

Here are some reasons why BCD should not be used the first reason it that BCD is a pain to to do arithmetic with and often times requires a complex designed arithmetic/logic Unit (ALU) for the calculations and it would be very hard to recognize carries. The second and last reason why BCD should not be used is because it no longer is true binary therefore it requires more space to store the the data which would not be space efficient enough if the environment has very limited resources to work with.[7][8][9]

# 7 Reflection

This chapter was very interesting, it showed more complex blocks of HDL and concepts that directly relate to the computers. I really enjoyed how detailed the explanations and the figures for code implementations were, much like the previous chapter this chapter had System Verilog and VHDL blocks next each other for comparison which made the reading and compared more easier. In the chapter most of the HDL blocks made sense, but on occasion I ran across an implementation that I struggled to understand such as the the memory implementation for the RAM and ROM, it not necessarily that I don't understand the block it's more like under what circumstances will implementing random-access memory (RAM) and read-only memory (ROM) be useful on an FGPA other than for educational purposes. This chapter focused a lot on trade offs such trade off mentioned are the adders; there speed versus the amount of power and hardware it takes to implement the adder, general rule of thumb is faster calculations = more hardware/power used. This aspect of the chapter reminded me a lot about the trade off in programming data structures; where faster the application is the more RAM it takes upon executing. The same is true for hardware implementation of blocks, if a circuit is fast and is able to calculate and process lots of data chances are it will take a lot more hardware to make it. This chapter also did a good job explaining the decimal point numbering system for binary very well, especially on the part for float-point notation (which is very similar scientific notation) after reading this section the whole and rational mixed numbers make more sense I can do the conversions very well after I practiced on some of the example problems provided in the book. One of the main focuses of the this chapter was memory and explaining by means of examples and HDL blocks, as mentioned earlier this section of the book did a wonderful job at explain the differences between RAM: SRAM and DRAM and ROM, in this section I learned a lot about the basics of how memory works in the hardware level which was rather intriguing since I have only every though of memory as there to hold information, but after learning more about how each part is connected with one another was very interesting. In this chapter I got to think about some cool topics and reinforced what I already knew hardware blocks, this chapter did a great job at explain the topics it covered, on of the major complaints that I have about the chapter was that there isn't enough example HDL blocks that show how other more complex combinational and sequential operations are implemented in HDL. Overall I believe this chapter is a very good stepping stone for a lot of topics, and provided enough information for people to start think and asking questions about the topics that were covered in the chapter.

# 8 Questions for Lecture

1. What are other implementations of the an adder that is widely used in the field?
2. If need be is there precision for the numbers that go up to 128-bits, 256-bits?
3. If I were to implement the the HDL for RAM in HDL example 5.7, what would be the fastest clock speed that I would attain? Would it be the fastest clock speed that is available on the FGPA that it is implemented on?
4. Are the principle used to make SSD fairly similar to that of the flash memory, since they are both nonvolatile, if so what make the transfer rate of the SSD so much faster than that of an ordinary flash drive?

## References


[1] D. Athow, "Pentium fdiv: The processor bug that shook the world." https://www.techradar.com/news/computing-components/processors/pentium-fdiv-the-processor-bug-that-shook-the-world-1270773, 2014.

[2] J. Markoff, "Company news; flaw undermines accuracy of pentium chips." https://www.nytimes.com/1994/11/24/business/company-news-flaw-undermines-accuracy-of-pentium-chips.html, 1996.

[3] A. Villas-Boas, "Intel recalled a major chip in 1995 and turned them into keychains inscribed by the ceo — and the message speaks to intel's current crisis." http://www.businessinsider.com/intel-pentium-p5-chip-recall-1995-keychain-ceo-inscribed-2018-1, 2018.

[4] Wiki, "Field-programmable gate array." https://en.wikipedia.org/wiki/Field-programmable_gate_array#History.

[5] Quora, "What are the most common uses for fpga today?." https://www.quora.com/What-are-the-most-common-uses-for-FPGA-today.

[6] N. Instruments, "Introduction to fpga technology: Top 5 benefits." http://www.ni.com/white-paper/6984/en/, 2012.

[7] Quora, "What are the benefits of bcd code ?." https://www.quora.com/What-are-the-benefits-of-BCD-Code.

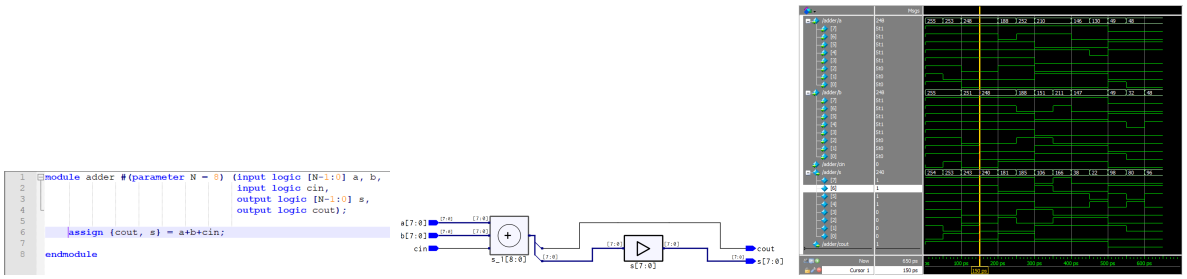
[8] I. Technology, "Binary-coded decimal." http://www.idc-online.com/technical_references/pdfs/electronic_engineering/Binary_Coded_Decimal.pdf.

[9] WikiBooks, "Practical electronics/binary-coded decimal." https://en.wikibooks.org/wiki/Practical_Electronics/Binary-coded_Decimal.

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

May 28, 2018

## 1 System Verilog Example 5.1

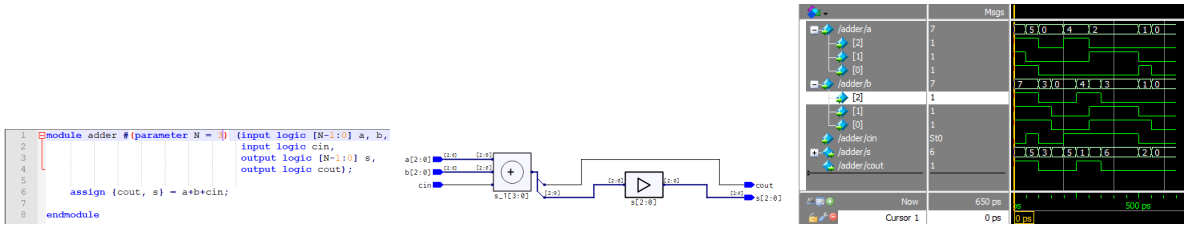



(a) System Verilog source (b) SynplifyPro synthesis (c) ModelSim simulation

Figure 1: Example 5.1

```
vsim work.adder
add wave *

/*for simulating the adder variation only a[0] - a[2] and b[0] - b[2]
  wil be used, since N is equal to 3 in the variation */

force -drive {sim:/adder/a[7]} 1 0, 0 500
force -drive {sim:/adder/a[6]} 1 0, 0 200, 1 250, 0 400
force -drive {sim:/adder/a[5]} 1 0, 0 300, 1 500
force -drive {sim:/adder/a[4]} 1 0, 0 450, 1 500
force -drive {sim:/adder/a[3]} 1 0, 0 300
force -drive {sim:/adder/a[2]} 1 0, 0 100, 1 200, 0 300
force -drive {sim:/adder/a[1]} 1 0, 0 50, 1 300, 0 500
force -drive {sim:/adder/a[0]} 1 0, 0 100, 1 500, 0 550

force -drive {sim:/adder/b[7]} 1 0, 0 500
force -drive {sim:/adder/b[6]} 1 0, 0 250, 1 350, 0 400
force -drive {sim:/adder/b[5]} 1 0, 0 300, 1 500
force -drive {sim:/adder/b[4]} 1 0, 0 550, 1 600
force -drive {sim:/adder/b[3]} 1 0, 0 300
force -drive {sim:/adder/b[2]} 1 0, 0 100, 1 250, 0 350
force -drive {sim:/adder/b[1]} 1 0, 0 150, 1 300, 0 500
force -drive {sim:/adder/b[0]} 1 0, 0 150, 1 300, 0 550

force -drive {sim:/adder/cin} 0 0, 1 50, 0 100, 1 200, 0 500

run 650
```

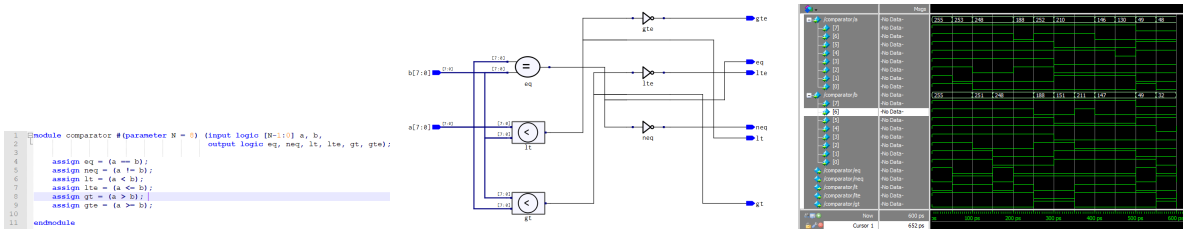
Figure 2: This do file can be used to simulate both example 5.1 and the variation within ModelSim.




(a) System Verilog source (b) SynplifyPro synthesis (c) ModelSim simulation

Figure 3: System Verilog Variation of Example 5.1

# 2 System Verilog Example 5.2

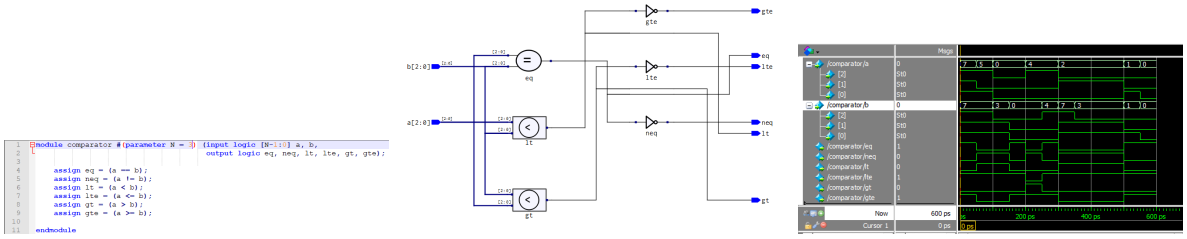


(a) System Verilog source (b) SynplifyPro synthesis (c) ModelSim simulation

Figure 4: System Verilog Example 5.2

```
vsim work.comparator
add wave *

/*for simulating the comparator variation only a[0] - a[2] and b[0] - b[2]
  wil be used, since N is equal to 3 in the variation */

force -drive {sim:/comparator/a[7]} 1 0, 0 500
force -drive {sim:/comparator/a[6]} 1 0, 0 200, 1 250, 0 400
force -drive {sim:/comparator/a[5]} 1 0, 0 300, 1 500
force -drive {sim:/comparator/a[4]} 1 0, 0 450, 1 500
force -drive {sim:/comparator/a[3]} 1 0, 0 300
force -drive {sim:/comparator/a[2]} 1 0, 0 100, 1 200, 0 300
force -drive {sim:/comparator/a[1]} 1 0, 0 50, 1 300, 0 500
force -drive {sim:/comparator/a[0]} 1 0, 0 100, 1 500, 0 550

force -drive {sim:/comparator/b[7]} 1 0, 0 500
force -drive {sim:/comparator/b[6]} 1 0, 0 250, 1 350, 0 400
force -drive {sim:/comparator/b[5]} 1 0, 0 300, 1 500
force -drive {sim:/comparator/b[4]} 1 0, 0 550, 1 600
force -drive {sim:/comparator/b[3]} 1 0, 0 300
force -drive {sim:/comparator/b[2]} 1 0, 0 100, 1 250, 0 350
force -drive {sim:/comparator/b[1]} 1 0, 0 150, 1 300, 0 500
force -drive {sim:/comparator/b[0]} 1 0, 0 150, 1 300, 0 550

run 600
```

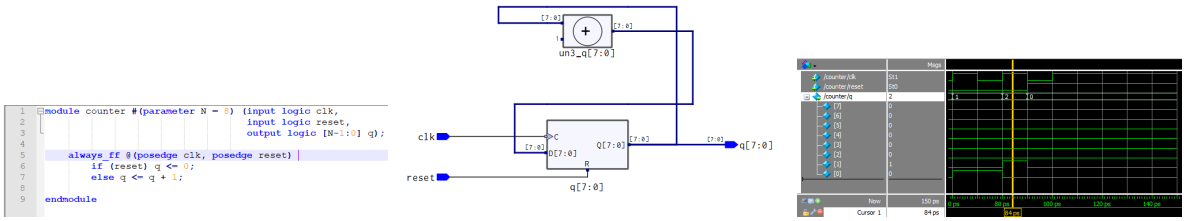
Figure 5: This do file can be used to simulate both example 5.2 and the variation within ModelSim.



(a) System Verilog source (b) SynplifyPro synthesis (c) ModelSim simulation

Figure 6: System Verilog Variation Example 5.2

# 3 System Verilog Example 5.3

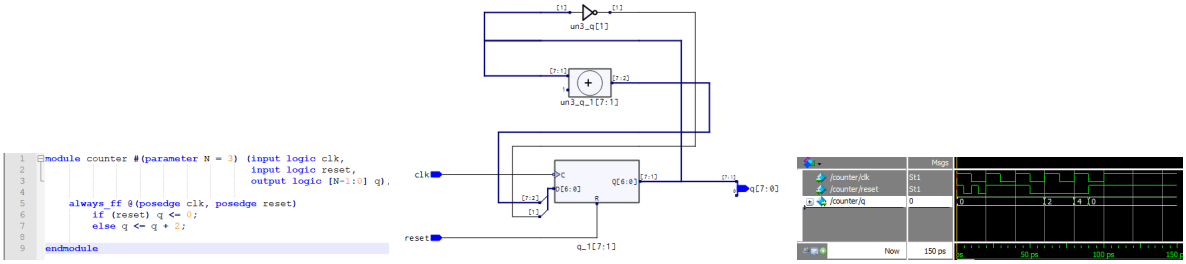



(a) System Verilog source (b) SynplifyPro synthesis (c) ModelSim simulation

Figure 7: System Verilog Example 5.3

```
vsim work.counter

add wave *

force -drive sim:/counter/reset 1 0, 0 5, 1 10, 0 15, 1 20, 0 50, 1 90
force -drive sim:/counter/clk 1 0, 0 10, 1 20, 0 30, 1 40, 0 50, 1 60, 0 70, 1 80, 0 90, 1 100

run 150
```

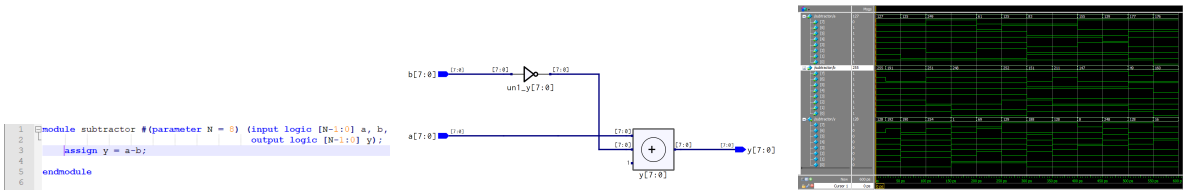
Figure 8: This do file can be used to simulate both example 5.3 and the variation within ModelSim.




(a) System Verilog source (b) SynplifyPro synthesis (c) ModelSim simulation

Figure 9: System Verilog Variation Example 5.3

# 4 System Verilog Example 5.4

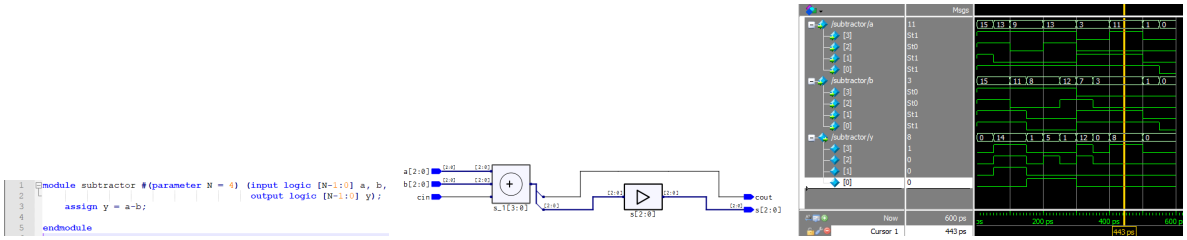


(a) System Verilog source (b) SynplifyPro synthesis (c) ModelSim simulation

Figure 10: System Verilog Example 5.4

```
vsim work.subtractor
add wave *

/*for simulating the subtractor variation only a[0] - a[3] and b[0] - b[3]
  wil be used, since N is equal to 4 in the variation */

force -drive {sim:/subtractor/a[7]} 0 0, 1 100, 0 200, 1 400
force -drive {sim:/subtractor/a[6]} 1 0, 0 200, 1 250, 0 400
force -drive {sim:/subtractor/a[5]} 1 0, 0 300, 1 500
force -drive {sim:/subtractor/a[4]} 1 0, 0 450, 1 500
force -drive {sim:/subtractor/a[3]} 1 0, 0 300, 1 400, 0 500
force -drive {sim:/subtractor/a[2]} 1 0, 0 100, 1 200, 0 300
force -drive {sim:/subtractor/a[1]} 1 0, 0 50, 1 300, 0 500
force -drive {sim:/subtractor/a[0]} 1 0, 1 500, 0 550

force -drive {sim:/subtractor/b[7]} 1 0, 0 500, 1 550
force -drive {sim:/subtractor/b[6]} 1 0, 0 20, 1 100, 0 300, 1 350, 0 400
force -drive {sim:/subtractor/b[5]} 1 0, 0 300, 1 500
force -drive {sim:/subtractor/b[4]} 1 0, 0 550, 1 600
force -drive {sim:/subtractor/b[3]} 1 0, 0 300
force -drive {sim:/subtractor/b[2]} 1 0, 0 100, 1 250, 0 350
force -drive {sim:/subtractor/b[1]} 1 0, 0 150, 1 300, 0 500
force -drive {sim:/subtractor/b[0]} 1 0, 0 150, 1 300, 0 550

run 600
```

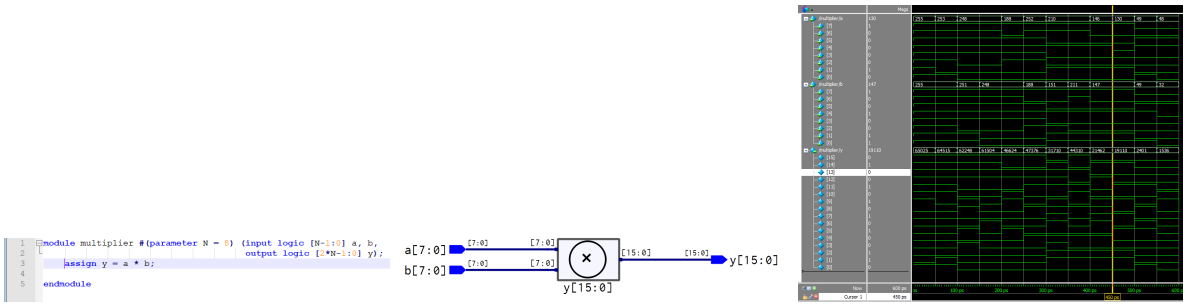
Figure 11: This do file can be used to simulate both example 5.4 and the variation within ModelSim.



(a) System Verilog source (b) SynplifyPro synthesis (c) ModelSim simulation

Figure 12: System Verilog Variation HDL Example 5.4

# 5 System Verilog Example 5.5

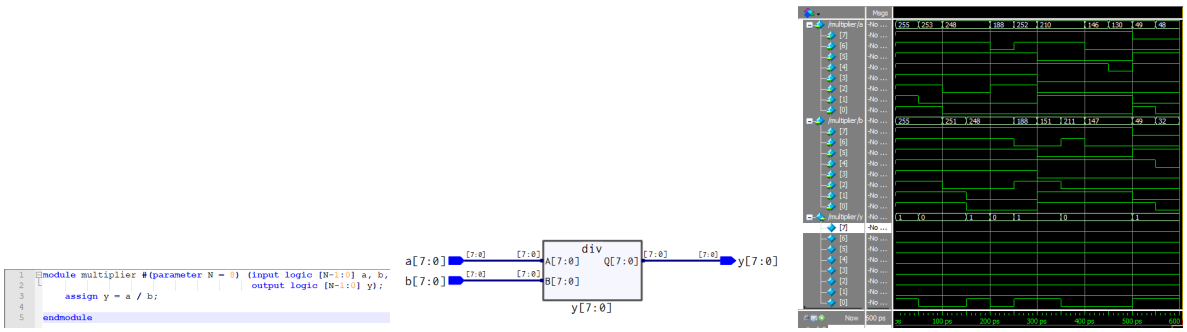



(a) System Verilog source (b) SynplifyPro synthesis (c) ModelSim simulation

Figure 13: System Verilog Example 5.5

```
vsim work.multiplier

add wave *

force -drive {sim:/multiplier/a[7]} 1 0, 0 500
force -drive {sim:/multiplier/a[6]} 1 0, 0 200, 1 250, 0 400
force -drive {sim:/multiplier/a[5]} 1 0, 0 300, 1 500
force -drive {sim:/multiplier/a[4]} 1 0, 0 450, 1 500
force -drive {sim:/multiplier/a[3]} 1 0, 0 300
force -drive {sim:/multiplier/a[2]} 1 0, 0 100, 1 200, 0 300
force -drive {sim:/multiplier/a[1]} 1 0, 0 50, 1 300, 0 500
force -drive {sim:/multiplier/a[0]} 1 0, 0 100, 1 500, 0 550

force -drive {sim:/multiplier/b[7]} 1 0, 0 500
force -drive {sim:/multiplier/b[6]} 1 0, 0 250, 1 350, 0 400
force -drive {sim:/multiplier/b[5]} 1 0, 0 300, 1 500
force -drive {sim:/multiplier/b[4]} 1 0, 0 550, 1 600
force -drive {sim:/multiplier/b[3]} 1 0, 0 300
force -drive {sim:/multiplier/b[2]} 1 0, 0 100, 1 250, 0 350
force -drive {sim:/multiplier/b[1]} 1 0, 0 150, 1 300, 0 500
force -drive {sim:/multiplier/b[0]} 1 0, 0 150, 1 300, 0 550

run 600
```

Figure 14: This do file can be used to simulate both example 5.5 and the variation within ModelSim.

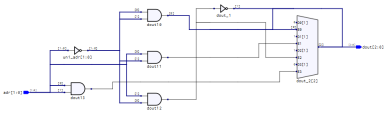


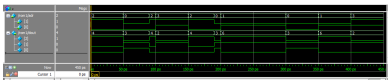

(a) System Verilog source (b) SynplifyPro synthesis (c) ModelSim simulation

Figure 15: System Verilog Variation Example 5.5

# 6 System Verilog Example 5.8

```
module rom(input logic [1:0] adr, output logic [2:0] dout)

    always_comb case(adr)
        2'b00: dout <= 3'b011;
        2'b01: dout <= 3'b110;
        2'b10: dout <= 3'b100;
        2'b11: dout <= 3'b010;
    endcase
endmodule
```





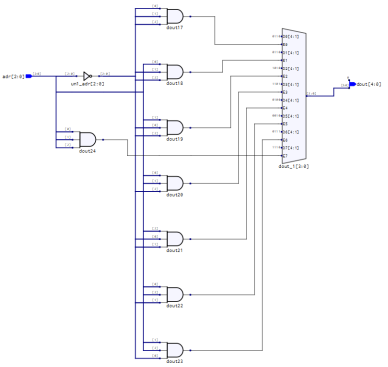
(a) System Verilog source (b) SynplifyPro synthesis (c) ModelSim simulation

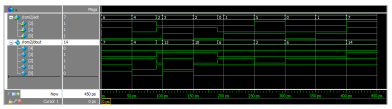
Figure 16: System Verilog Example 5.8

```
vsim work.rom1 /* this will be rom2 when testing variation, every instance of rom1
                  will be replace with rom2 in this test file when simulating rom2*/
add wave *

/*for rom1 simulation only adr[0] - adr[1] is used, because the input bus in only
 2 bits versus the 3 bit input bus on rom2 (variation) */

force -drive {sim:/rom1/adr[0]} 0 0, 1 100, 0 150, 1 200, 0 300, 1 350
force -drive {sim:/rom1/adr[1]} 1 0, 0 50, 1 90, 0 190, 1 400
force -drive {sim:/rom1/adr[2]} 1 0, 0 90, 1 250, 0 300, 1 400

run 450
```

Figure 17: This do file can be used to simulate both example 5.8 and the variation within ModelSim. I would have to change a couple of things in the do file but the numbers will be the same.

```
module rom(input logic [2:0] adr, output logic [4:0] dout);
    always_comb case(adr)
        3'b000: dout <= 4'b0110;
        3'b001: dout <= 4'b0110;
        3'b010: dout <= 4'b1010;
        3'b011: dout <= 4'b1101;
        3'b100: dout <= 4'b0100;
        3'b101: dout <= 4'b0010;
        3'b110: dout <= 4'b0111;
        3'b111: dout <= 4'b1110;
    endcase
endmodule
```





(a) System Verilog source (b) SynplifyPro synthesis (c) ModelSim simulation

Figure 18: System Verilog Variation Example 5.8