Lab 1: Simple Combinational Logic

Daniel Simone

22 September 2021

1 Introduction

1.1 Class Information

Course: ECE 206
Demo Lab section: B02
Lecture Instructor: Professor Sharad Malik

2 Write-Up

2.1 Tutorial Review Questions

2.1.1 Question 1.1

Describe the fundamental differences between a program written in a high-level language, e.g. Java, and a program written in Verilog? How does your approach to program design change when programming in Verilog?

The differences between a high-level language like Java and Verilog include:

- Verilog has concurrency, where each statement is executed in parallel, rather than sequentially as it would in Java;
- Verilog allows bit-level control of the hardware it models, thus requiring more careful design from the onset;
- Verilog code is often mapped to actual hardware gates after simulation via synthesis, but some operations and constructs like multiplication and division are not synthesizable into physical hardware;
- Verilog only has two types: wires and registers, as opposed to Java's eight primitive types and theoretically infinite objects.

These differences force the programmer to have the final implementation in mind when coding and think in terms of wires and registers, rather than numbers, characters, words, and other types of data that we may be more familiar with.

2.1.2 Question 1.2

What does it mean for a wire to be driven? What value does a wire take when it is not being driven? What about when it is being driven by more than one driver?

What a driven wire means is that the wires derives its value from the combinational logic element (like a gate) or state-storing element (like a flip-flop) that it is attached to - wires cannot store values themselves. When a wire is not being driven, it has the value Z (high impedance), and when there are multiple drivers, it has the value X (indeterminate).

2.1.3 Question 1.8

What is the difference between continuous and procedural assignment in Verilog? Which one describes hardware more naturally? Why?

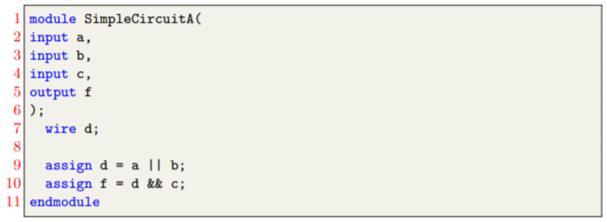
Continuous assignment uses wires and the "assign" statement to drive those wires (where the "assign" can also be implicit).

Procedural assignment uses the reg datatype and the "always" statement to essentially drive a reg like a wire. Procedural assignment also allows use of the if-else, case, and Tri-state logic statements.

Continuous assignment models hardware more naturally, since the logic works more closely to how wires in a physical circuit would.

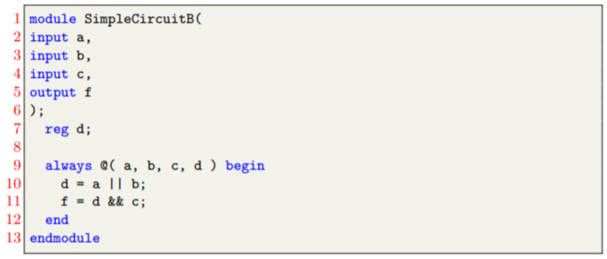
2.1.4 Question 1.11

Read and understand the following Verilog modules:



```
module SimpleCircuitA(
input a,
input b,
input c,
output f
);
  wire d;

  assign d = a || b;
  assign f = d && c;
endmodule
```



```
module SimpleCircuitB(
input a,
input b,
input c,
output f
);
  reg d;

  always @( a, b, c, d ) begin
    d = a || b;
    f = d && c;
  end
endmodule
```

a *Classify each module as either a structural or behavioral model.*

Module A is structural, while Module B is behavioral.

b *How does the execution of code differ between these two implementations?*

The execution is different because in Module A, changes to inputs a, b, c, or d elsewhere in the module will also change d and/or f, but in Module B, only changes to inputs a, b, c, or d outside the always statement will change d and/or f.

c *The sensitivity list for the always procedure in SimpleCircuitB contains four signals. Let's say the value of the signal a changes and the procedure is triggered. Consequently, the value of d changes, but is the procedure triggered again? Does the signal d need to be in the sensitivity list?*

If the value of d changes inside the always statement, then the always statement will not be called again. It is always a good idea to include all variables on the right of the statements in the body in the sensitivity list, so d should be included regardless, as it affects the signal f.

d *How might you rewrite the sensitivity list to be better?*

This sensitivity list can be replaced with just the * statement.

e *For this circuit, which style of writing seems more intuitive? Why?*

The style of writing that seems more intuitive is module A, since the continuous assignment code more closely models what a physical circuit would work like. The code is also more straightforward, since it does not require the sensitivity list. The sensitivity list is very unintuitive to anyone unfamiliar with Varilog.

2.1.5 Question 2.1

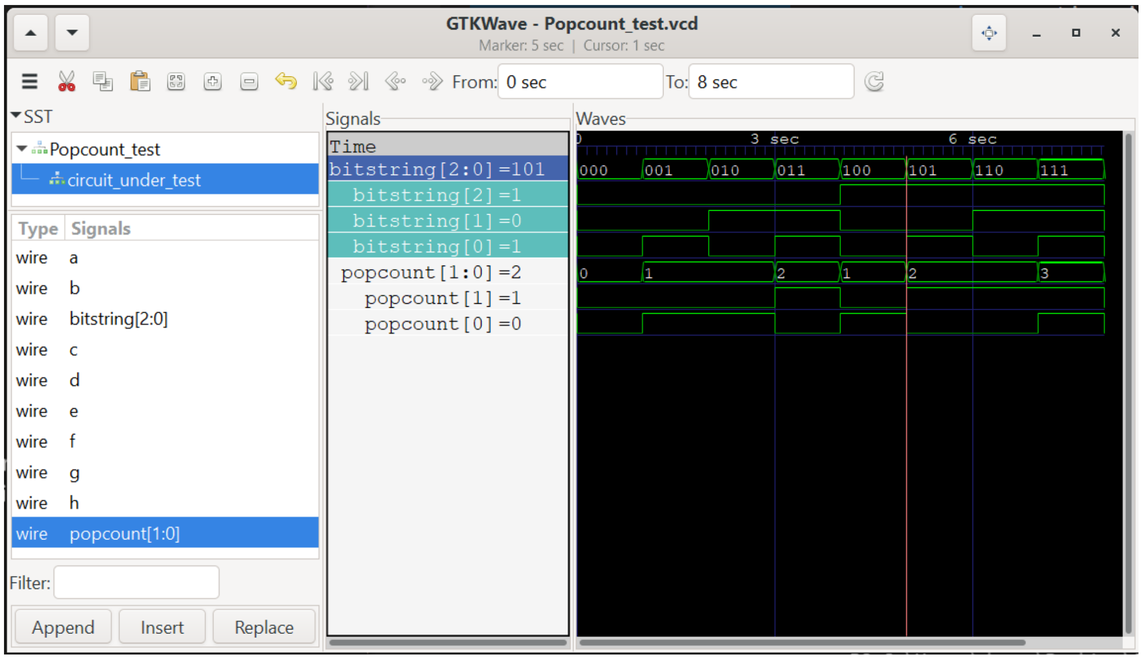
Explain why breaking down complex digital circuit designs into modules can be useful for hardware designers.

Breaking down circuits into modules is useful because it makes code for a circuit more readable, allows code modules to be reused, and prevents duplication of code since a module can be called as many times as needed.

2.2 Write-Up Questions

2.2.1 Question 1

Place the marker at a time in the simulation such that the "Signals" pane shows values when bitstring held a binary representation of the number 5. Leave circuit under test selected in the "SST" pane so that all the signals are visible and it displays that they are all wire datatypes. Take a screenshot showing the waveforms for all the bits within bitstring and popcount, as well as any other signals you feel like adding, over the course of the simulation. Do the waveforms for the individual bits convey any additional information than the values shown for the multi-bit signal if it were not expanded?


GTKWave - Popcount_test.vcd
Marker: 5 sec | Cursor: 1 sec

From: 0 sec To: 8 sec

SST
Popcount_test
circuit_under_test

Type	Signals
wire	a
wire	b
wire	bitstring[2:0]
wire	c
wire	d
wire	e
wire	f
wire	g
wire	h
wire	popcount[1:0]

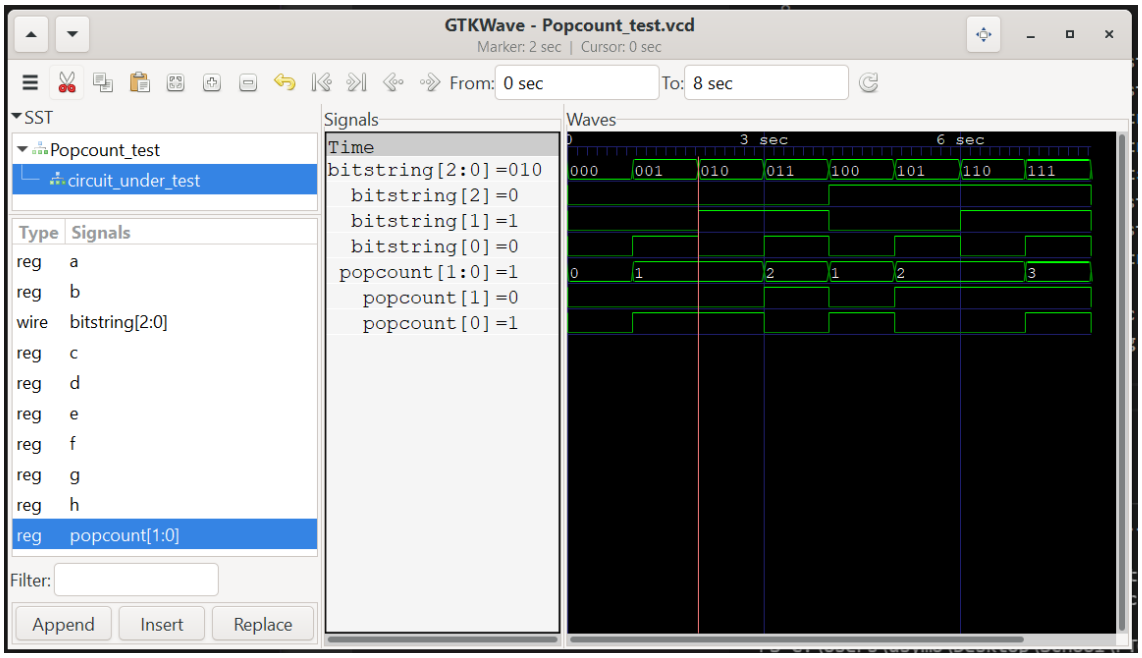
```
Time
bitstring[2:0] =101
  bitstring[2] =1
  bitstring[1] =0
  bitstring[0] =1
 popcount[1:0] =2
   popcount[1] =1
   popcount[0] =0
```

Figure 2.1: Waveforms

The waveforms for the individual bits also convey the information of when the individual bits were logical high (1) or logical low (0). Instead of seeing them stacked on top of each other, it is also instructive to see them individually, since multiple different variations of input bits lead to the same output bits, which is otherwise concealed in the stacked format. For instance, inputs 011 and 101 return the same output of 2.

2.2.2 Question 2

Place the marker at a time in the simulation such that the "Signals" pane shows values when bitstring held a binary representation of the number 2. Leave circuit under test selected in the "SST" pane so that all the signals are visible and that it displays that they are all regs, except for bitstring. Take a screenshot showing the values of the vectors bitstring and popcount, as well as any other signals you feel like adding, over the course of the simulation. How do the values shown in the "Waves" pane for the multi-bit vectors show that the circuit is computing the correct value at all points in time?



Type	Signals
reg	a
reg	b
wire	bitstring[2:0]
reg	c
reg	d
reg	e
reg	f
reg	g
reg	h
reg	popcount[1:0]

```
Time
bitstring[2:0] =010
  bitstring[2] =0
  bitstring[1] =1
  bitstring[0] =0
 popcount[1:0] =1
   popcount[1] =0
   popcount[0] =1
```

Figure 2.2: Waveforms

The values in the "Waves" pane show that the circuit is computing the correct value at all points in time because, especially if one sets the "popcount" wave to decimal notation, it is clearly visible that the correct number of 1's in the "bitstring" signal are being counted. It is easy to correlate bitstring values, like "011," to popcount values, like "2."

2.2.3 Question 3

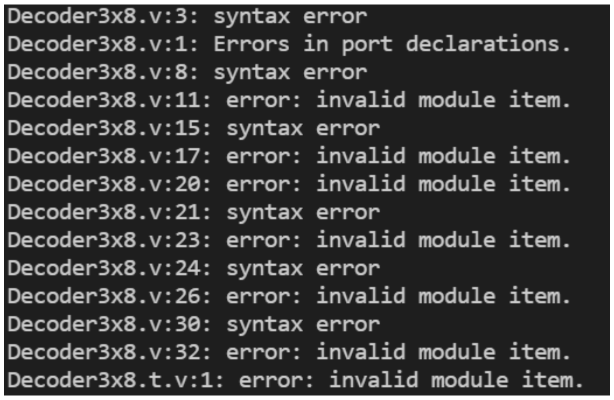
Why might you prefer to write this module in structural or behavioral Verilog? What are the benefits of each choice, if any?

You might prefer structural Verilog since it describes the circuit more accurately to the physical hardware than behavioral Verilog. It is also more intuitive, and thus good for simpler circuits.

On the other hand, behavioral Verilog allows the use of if-else, case, and Tri-State statements which may make the coding process easier. This makes coding more complex circuits easier.

2.2.4 Question 4

Describe the warning or error messages the Icarus Verilog compiler generated. What were the errors that caused those messages?



```
Decoder3x8.v:3: syntax error
Decoder3x8.v:1: Errors in port declarations.
Decoder3x8.v:8: syntax error
Decoder3x8.v:11: error: invalid module item.
Decoder3x8.v:15: syntax error
Decoder3x8.v:17: error: invalid module item.
Decoder3x8.v:20: error: invalid module item.
Decoder3x8.v:21: syntax error
Decoder3x8.v:23: error: invalid module item.
Decoder3x8.v:24: syntax error
Decoder3x8.v:26: error: invalid module item.
Decoder3x8.v:30: syntax error
Decoder3x8.v:32: error: invalid module item.
Decoder3x8.t.v:1: error: invalid module item.
```

Figure 2.3: Error Codes

Icarus Verilog produced three kinds of errors, and the following were their solutions:

- Syntax error:
  - Needed to add a bracket "[" during the declaration of the output wire d.
  - Needed to add "@(*)" to the "always begin" statement.
  - Needed to add "endcase" to conclude the "case" statement.

- Errors in port declarations:
  - Produced by the fact that the output wire needed a bracket, so this port was erroneously declared.
- Error: invalid module item
  - Produced by the register ”d” missing a semicolon after its declaration.

2.2.5 Question 5

For a combinational circuit, it is possible to write a Boolean formula for each output bit that specifies its value in terms of the input. Does the buggy code, as shown in Figure 1.1, successfully implement the decoder combinational circuit? If so, support your answer by writing a Boolean formula for each output bit. If not, justify your claim using the behavior shown in GTKWave.

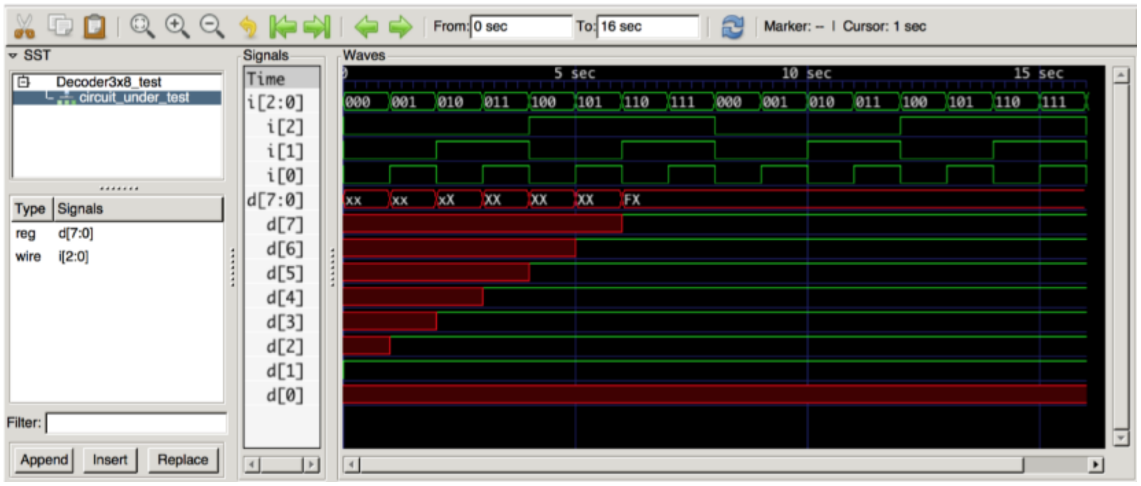


Figure 2.4: Reference Image of Erroneous Simulation

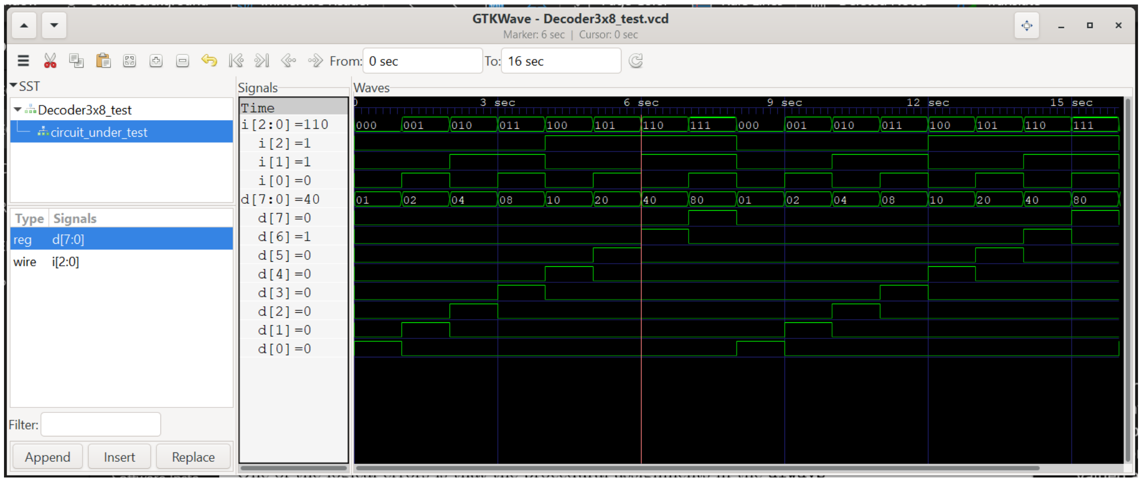
No, this is the incorrect behavior. There are several problems visible from the reference image:

- A decoder is supposed to transform a multi-bit signal into a unary one. However, as can clearly be seen from the GTKWave behavior, multiple output ”d” bits are valued at 1 simultaneously, producing a multi-bit output signal. There should only be one output ”d” bit that corresponds to each 3-bit number in the input ”i” bits.

- Many output ”d” bits also have an ”XX” or ”XF” value, which is also not an acceptable unary value.
- There is never an assignment for d[0] - it does not correspond to any multi-bit signal - when d[0] should correspond to ”000” in binary.

2.2.6 Question 6

In GTKWave, place the marker at a time in the simulation such that the “Signals” pane shows values when i held a binary representation of the number 6. Take a screenshot showing the waveforms for all the bits within i and d, as well as any other signals you feel like adding, over the course of the simulation.



```
Time
i[2:0] =110
  i[2] =1
  i[1] =1
  i[0] =0
d[7:0] =40
  d[7] =0
  d[6] =1
  d[5] =0
  d[4] =0
  d[3] =0
  d[2] =0
  d[1] =0
  d[0] =0
```

```
i[2:0]: 000 001 010 011 100 101 110 111 000 001 010 011 100 101 110 111
d[7:0]: 01 02 04 08 10 20 40 80 01 02 04 08 10 20 40 80
```

Figure 2.5: Behavioral 3x8 Decoder Waveform

2.2.7 Question 7

Why might you prefer to write this module in structural or behavioral Verilog? What are the benefits of each choice, if any?

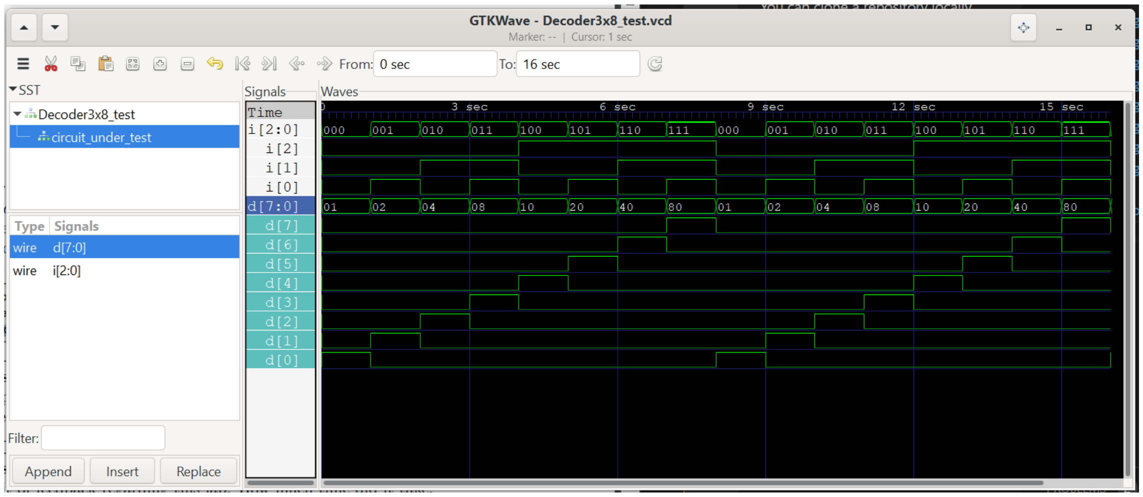


Figure 2.6: Structural 3x8 Decoder Waveform

The decoder may be preferred to be written in structural Verilog because the binary input values are clearly visible in the code, making the code more realistic to a physical circuit. You also have more control over bitwise behavior in structural Varilog. Behavioral Verilog may be preferable since the cases clearly show the decimal equivalents of the binary values, making the code more human-readable. Behavioral Varilog also includes if-else and Tri-State Bugger statements which may be helpful, and allows you to more easily add cases for more input bits.

2.2.8 Question 8

Please leave a bit of feedback regarding this lab. How much time did it take, how difficult was it, did you get stuck anywhere? There are no wrong answers here – we'll be using the feedback to adjust this lab for the future.

This lab took me about 7 hours. The longest part of the lab was definitely the reading. It was especially unclear as to what the difference between structural and behavioral Verilog is in theory, beyond the fact that structural uses wires and behavioral uses registers.

This paper represents my own work in accordance with University regulations.

Daniel Simone