Lab 1 Report

Group 3: Dennis Zang (704766877), Daniel Park (104809832),

Samuel Pando (904809319)

October 23, 2020

Part 1: 1 BIT ALU

[Components and Implementation Logic]

In our design for the 1 bit ALU, our top-level module is "bitALU", which depends only on the submodules "mux2_1" and "addbit". Since we are not allowed to use anything other than gate primitives for structural Verilog, implementing these submodules is required, as we cannot use conditionals and in-code arithmetic in behavioral Verilog. Details of the implementations of these three modules are as described below.

[bitALU]

"bitALU" is the top-level module that depends on a 2-to-1 multiplexer and an add/subtract submodule (to be explained below). It generally acts as an interface for using the underlying "addbit" module in order to perform both addition and subtraction. For addition (CTRL = 0), it's inputs for A, B, and CIN into "addbit" are unchanged and left as-is for "addbit" to process. However, when performing subtraction, input B will be inverted using a not gate and a 2-to-1 multiplexer to toggle between B and not-B between the two operations. Note that when performing subtraction, CIN for the least significant bit (not relevant here, but will be later when we need to build a ripple-adder in part 2) needs to be set to 1 for the result to be correct.

[mux2_1]

The submodule "mux2_1" is really just a simple multiplexer that toggles between inputs a and b using input sel. The logic works by calculating "(a AND !sel)" and "(b AND sel)", which would each be set if a, b, and sel are the correct values. The final output o would be taking the OR between these two values. (Note: behavioral Verilog would not allow for the aforementioned literals, but the logic works like this.)

[addbit]

"addbit" is the submodule responsible for adding the inputs a, b, and cin to calculate the sum s and carry-out cout. The high-level logic works like this: s would be set if an odd

number of inputs (a, b, and cin) are set, while cout would be set if at least 2 inputs (a, b, and cin) are set. This logic is implemented using only logic gate primitives.

[mux3_1] (unused)

As defined in the specs, we used “mux2_1” to create a 3-to-1 multiplexer, “mux3_1”. It is not used and not necessary in the final design for the adder or the ALU. The logic works like this: we first use a 2-to-1 multiplexer to toggle between values a and c (as they correspond to sel = 00 and sel = 10) using sel[1], then we take that result and toggle it with input b using another 2-to-1 multiplexer with sel[0] (as b corresponds to sel = 01).

[Waveforms/Testbench]

The testbench consists simply of all relevant inputs for “mux2_1”, “mux3_1”, “addbit” and “bitALU”. The test cases are as follows:

[“mux2_1”]

Here, we simply toggled inputs a and b collectively as (a = 0, b = 0), (a = 1, b = 0), and (a = 0, b = 1), and input sel = 0 or 1 to confirm functionality.

[“mux3_1”]

We tested “mux3_1” the same as “mux2_1”, but instead we toggled (a = 0, b = 0, c = 0), (a = 1, b = 0, c = 0), (a = 0, b = 1, c = 0), and (a = 0, b = 0, and c = 1) with (sel = 00, 01, and 11) for confirm.

[“addbit”]

For “addbit” we just added all possible inputs for cin, a, and b to observe all possible values.

[“bitALU”]

For “bitALU”, we also put in all possible combinations of CIN, A, and B for the cases of CTRL = 0 or CTRL = 1, then checked all of their corresponding outputs.

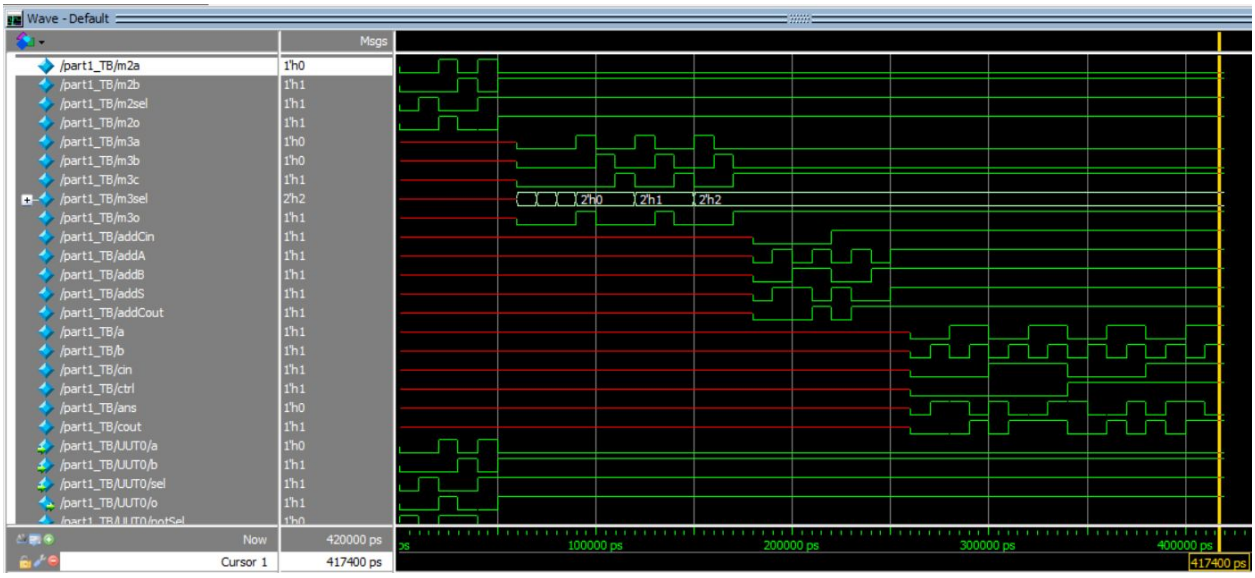

Figure 1.1: The waveforms of 4 units-under-test. The ordering is “”mux2_1”, “mux3_1”, “addbit”, and “bitALU”.

[Challenges Faced]

This part was relatively simple. The main challenge is connecting the relevant modules to Part 2 and debugging the modules. One thing that we could have saved more time was to write the testbench and debug the submodules for Part 1 before proceeding to Part 2.

Sources on what I found on structural verilog:

http://users.ece.utexas.edu/~patt/04s.382N/tutorial/verilog_manual.html

“Structural verilog is composed of module instances and their interconnections (by wires) only”

http://content.inflibnet.ac.in/data-server/eacharya-documents/53e0c6cbe413016f23443704_INFIEP_33/4/LM/33-4-LM-V1-S1__structural_modeling_questions.pdf

Part 2: 16-BIT ALU

[Components and Implementation Logic]

In addition to the modules in Part 1, we added 4 new modules to perform each of the requested operations in the specs: “pportMux2_1”, “pportMux16_1”, “mux14_1”, and “bit16AddSub”. These 4 submodules are used for various operations used inside the top-level module “bit16ALU”. Details for these submodules are as follows.

[“pportMux2_1”]

This is simply just a new version of “mux2_1” from Part 1, except that inputs a and b are now 16-bits. It is composed of 16 instances “mux2_1”, one for each bit.

[“pportMux16_1”]

This is a 16-input version of “pportMux2_1”, which is later used to select the necessary output S for the top-level module. It is composed of 4-stages, each stage filtering out half of the 16-bit inputs based on the bits of sel.

[“mux16_1”]

This is a 16-input version of “mux2_1”. Each of the inputs are 1-bit, and will later be used to select the Overflow output for “bit16ALU”.

[“bit16AddSub”]

This submodule is basically a ripple-adder that makes use of “bitALU” to perform addition or subtraction where necessary in “bit16ALU”. Inputting the same control signal into each “bitALU” and the carry-in for the LSB will allow for addition or subtraction.

Furthermore, for the top level module “bit16ALU”, we implemented the logic for each requested operation. The potential outputs for each operation are computed, then selected using the new aforementioned multiple-bit multiplexers using ALUCtrl. Details of each operation in “bit16ALU” (and corresponding ALUCtrl values) are as listed below.

{Subtraction} (ALUCtrl = 0000)

For the resulting difference, we made an instance of “bit16AddSub” with the ctrl set constantly to 1. The inputs A and B are fed into “bit16AddSub” inputs a and b, and the result will later be potentially selected by an instance of “pportMux16_1” to output S.

{Addition} (ALUCtrl = 0001)

Like subtraction, we need a separate instance of “bit16AddSub”. Everything is the same, except the ctrl input is constantly set to 0.

{Bitwise OR} (ALUCtrl = 0010)

For the OR operation, we simply applied the or gate primitive on all bits of A.

{Bitwise AND} (ALUCtrl = 0011)

For the AND operation, we simply applied the and gate primitive on all bits of A.

{Decrement} (ALUCtrl = 0100)

This is implemented the exact same way as subtraction, except only “bit16ALU” input A is input into “bit16AddSub” input a, while “bit16AddSub” input b is set to a constant 16’h0001 and ctrl is set to 1.

{Increment} (ALUCtrl = 0101)

This is implemented the exact same way as addition, except only “bit16ALU” input A is input into “bit16AddSub” input a, while “bit16AddSub” input b is set to a constant 16’h0001 and ctrl is set of 0.

{Invert} (ALUCtrl = 0110)

The two’s complement of input A is found by applying the not gate to every bit of A and adding one. Overflow is set if adding one causes an overflow.

{Arithmetic Shift Left} (ALUCtrl = 1100)

Arithmetic Shift Left is identical to Logical Shift Left. See Logical Shift Left below.

{Arithmetic Shift Right} (ALUCtrl = 1110)

Arithmetic Shift Right is implemented by shifting over bits once so a more significant bit overwrites the less significant bit adjacent to it. The most significant bit is left untouched to maintain the number’s sign. The process is repeated B number of times. Overflow never occurs.

{Logical Shift Left} (ALUCtrl = 1000)

Logical Shift Left is implemented by shifting all bits over so a less significant bit overwrites the more significant bit adjacent to it. The least significant bit is replaced with a zero. The process is repeated B number of times. Overflow occurs whenever a new bit is pushed into the sign position.

{Logical Shift Right} (ALUCtrl = 1010)

Logical Shift Right is implemented by shifting all bits over so a more significant bit overwrites the less significant bit adjacent to it. The most significant bit is replaced with a zero. The process is repeated B number of times. Overflow never occurs.

{Set on Less than or Equal} (ALUCtrl = 1001)

For this operation, we set all bits except the LSB to 0. The LSB is set to 1 if both of the two following conditions are satisfied:

1) Either A is negative and B is positive, the difference is negative, or the difference is 0

2) Cannot have both A being positive and B being negative

The difference is found using a separate instance “bit16AddSub” (we could have reused the one from subtraction, but better code readability this way) with (a = A), (b = B), and (ctrl = 1). The signs of A and B are checked by looking at their MSB’s.

One thing to note is that the output Zero depends only on the final output S, in which Zero is set if S = 16’h0000 and unset otherwise.

[Waveforms/Testbench]

The testbench tests each ALU operation. Several tests were written for Addition, Subtraction, and Set on Less than or Equal to ensure correct behavior. Operations were tested with negative and positive values when the logic for their behavior made such a distinction necessary. The results of the testbench can be seen below through the waveform.

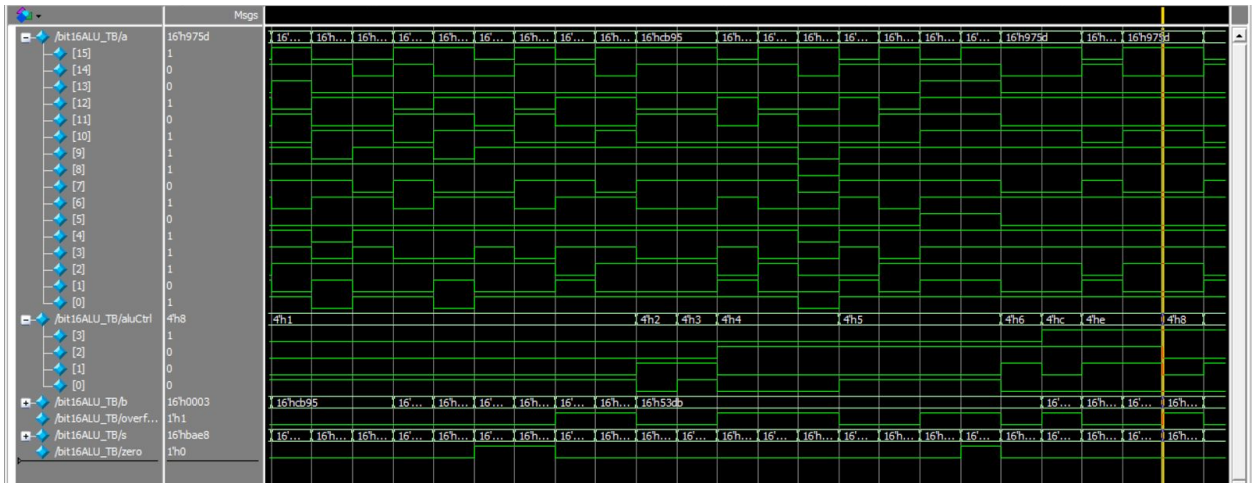

Figure 2.1 ALU Testbench Input A and ALU Ctrl. The figure displays the waveforms for the input and ALU Ctrl. Due to the large number of bits, the waveform is broken into three pictures.

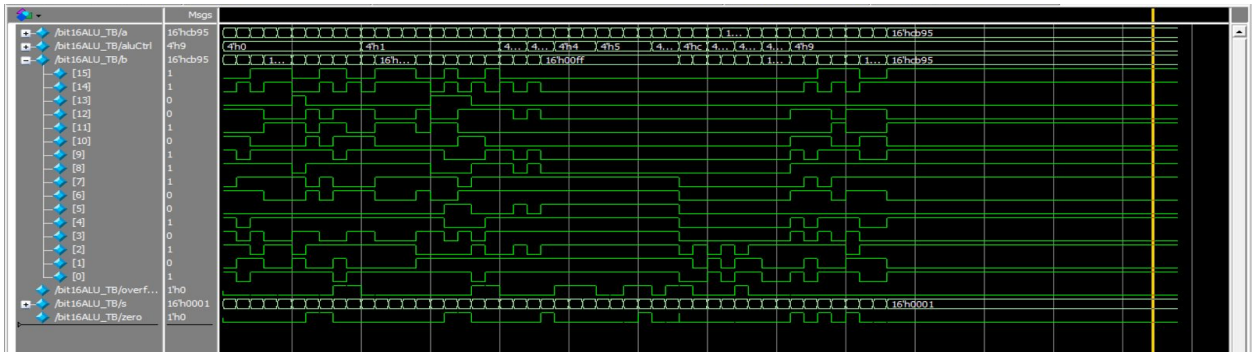

Figure 2.2 ALU Testbench Input B. Figure 2 of 3 for the waveform of the testbench. Here the secondary input to the ALU can be observed.

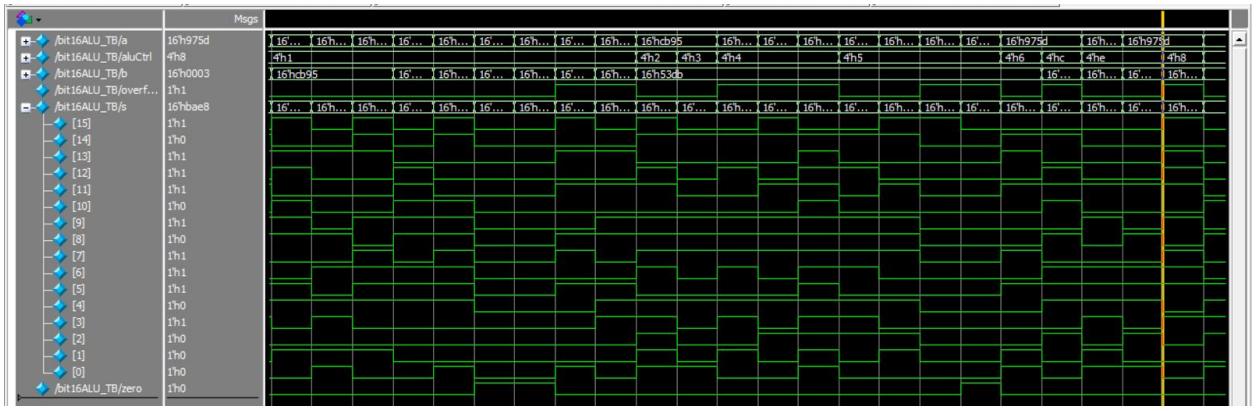

Figure 2.3 ALU Testbench Output. Figure 3 of 3 for the waveform of the testbench. Here the output bit values can be observed.

[Challenges Faced]

One of the main challenges was using Verilog primitives to represent boolean logic. Several operations normally taken for granted had to be designed with every bit in mind. As a side effect, using primitives made the code verbose and increased the amount of time spent debugging. Another challenge was keeping track of many different variables. Since the multiplexor chose from pre-calculated answers, those answers all had to be computed before one could be chosen. This is in contrast to higher level languages where only the single necessary result is computed (at least from the view of the programmer). Lastly there were some difficulties using behavioral code. It was discovered that our behavioral code was missing a timeunit specification.

Part 3: Register File

[Components and Implementation Logic]

The design for the register file was relatively simple compared to the other parts. I accomplished it with the use of the module “reg_file”, and the file itself was written with behavioral code.

[reg_file]

This module consists of 7 inputs and 2 outputs. It also contains the registers themselves. First, there is clock input and a reset input. The clock is important because all of our logic will be executing on positive clock edges. The reset input, on a positive clock edge, clears all the registers, setting all their values to zero. Next, there is the WrEn input, which enables the write operation for later use. After that, we have the Rw, Ra, and Rb, which give the 5-bit register addresses for the write and read registers respectively. Finally, we have busW, which gives the data to write to the register to write to address Rw given a high WrEn signal. The two outputs are busA and busB, which are the read results from registers with addresses Ra and Rb. Reset is given priority and write, when possible, always occurs before a read operation.

[Waveforms/Testbench]

For the testbench, I used a module named “reg_file_tb.”

[reg_file_tb]

For this testbench, I started by disabling write and reading some values from the registers. As I expected, at the start, the values were all unknown. Immediately after, I set the reset input to high and watched to make sure the registers were all set to 0. After that, I wrote to a register and then read from the same register to make sure that the order was preserved and the value was written. Finally, I disabled write and tried to read from the attempted write to see if it was stopped.

Figure 3.1: The waveform from the register file testbench. As we can see, at the outset, the read values show up unknown. After reset, they read zero. Writes change the values in the register (confirmed by reads) and disabling writes works as intended.

[Challenges Faced]

Overall, this problem was less difficult since behavioral code was allowed. If there were any challenges, it was implementing the testbench and making sure that all operations occurred in the proper order. But since this part contained some similarities to the counter from the previous lab (at least on the surface), the process was smooth.

Lab Questions:

1) The implementation difference between structural and behavioral verilog is that structural verilog only allows instances of wire, registers, and various submodules (including gate primitives like “and()”, “or()”, and “not()”), while behavioral verilog doesn’t have these constraints. Execution-wise, behavioral verilog executes in a more linear fashion (like with conventional programming) with some exceptions, while structural verilog will always act like the synthesized hardware directly. For instance, we can compare our implementation of “mux2_1” with a behavioral implementation given below.

```
module behavioralMux (a, b, sel, out);
   input a;
   input b;
   input sel;
   output out;

   wire selA;
   wire selB;
   always @(a, b, sel)
      begin
         selA <= a & !sel;
         selB <= b & sel;
         out <= selA | selB;
      end
endmodule
```

Here, verilog can execute the logic, and it will behave as it should, but it cannot be directly synthesized (without the compiler). With structural verilog, every wire, register, and submodule is enumerated, and can be directly synthesized physically by hand with much less hassle.

2) An asynchronous multiplexer would work by performing assignments immediately (of course with some transmission delay if that is given) and implies a combinational circuit. This would most likely be implemented in behavioral verilog with an always block that is triggered by any of the inputs. A synchronous multiplexer would work on updates based on the rising or falling edge of a clock signal (implying a sequential circuit with flip-flops). A likely implementation using behavioral verilog with an always block that triggers on a rising or falling edge of a clock signal, which would then trigger the embedded combinational logic nested inside the always block.

3) An arithmetic shifter keeps the sign of a two’s complement number while a logical shifter performs a shift without considering the sign of the number it shifts. In practice, a difference is only present in the case of arithmetic right shifters and logical right shifters. An arithmetic right shifter will fill its most significant bits with the bit that was previously located in the MSB spot. However a logical right shifter will fill those same most significant bits with zero.

4) One could implement an arithmetic shifter without behavioral verilog by shifting bits over once B times. (If shifting right, the last bit is left untouched. If shifting left, the LSB is replaced with a 0.) So a single shift of length B would be broken into B shifts of length one. However since we don't know the value of B, we would compute all the results for all possible values of B and use a multiplexor to choose the correct value.

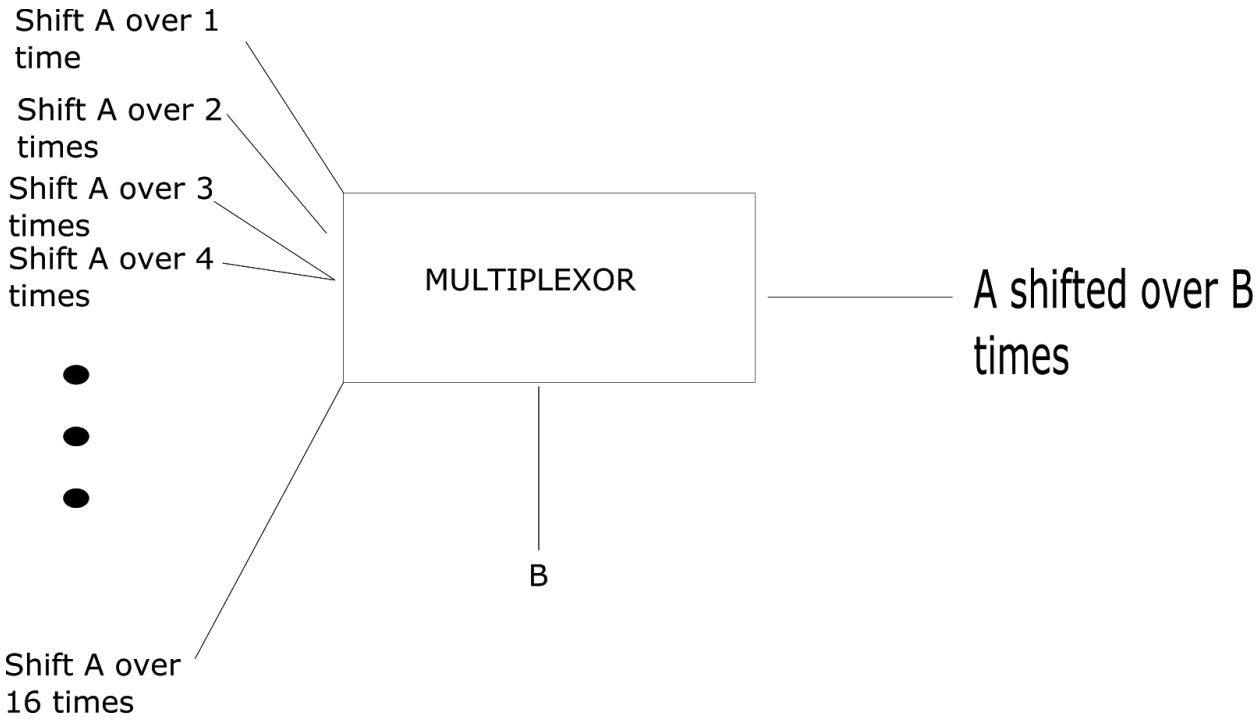

Figure 4.1 Non-Behavioral Shifter. The diagram above displays a possible design for an arithmetic shifter without using Behavioral Verilog.

Member Contributions:

Dennis Zang: Part 1 & 2

Daniel Park: Part 2

Samuel Pando: Part 3