Table of Contents

0. General Information

The zip file I have sent contains this document and some other folders. There are five folders as follows:

1. Verilog Code: This folder contains the Verilog code for the designs.
2. Test benches: This folder contains the test benches written to test just the higher level elements. I have not included the test benches for adder, multiplexer, decoder, register, CLOG2 function and the flag register. Basically I have included two test benches, one for the register file and the other one for the ALU. Both these test benches test both the structural and behavioral versions of those modules. Also the correctness of the flags was checked by getting the output directly from the ALU.
3. Do-TCL Files: This folder contains the scripts for running test benches as batch jobs.
4. Synthesis Results: This folder contains the ".syr" files of the ALU and Register File.
5. Bigger Schematics: This folder contains larger versions of the images used within this document. If the reader wants to get a better look at the designs or the simulation outputs, they can refer to here.

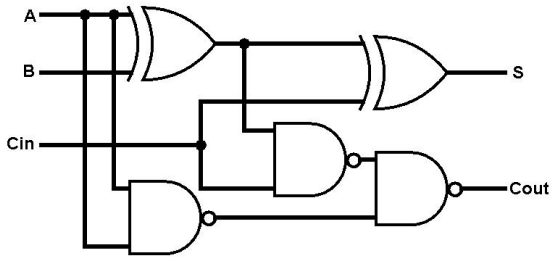
In general I have tried to use as much parameterization as possible. I also tried to follow the XST specification closely.

1. Primitives Implemented

In this section I will explain the design of some of the primitives that were used to implement the register file and the ALU.

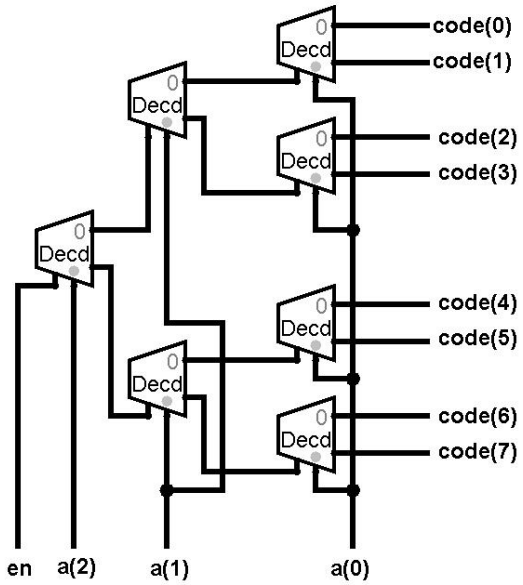
1.1. Full Adder 1-bit

This is the same design as in the first assignment, but this time I used the built-in Verilog gates without delay instead of using the gates that we implemented. The circuit is the same and as follows.



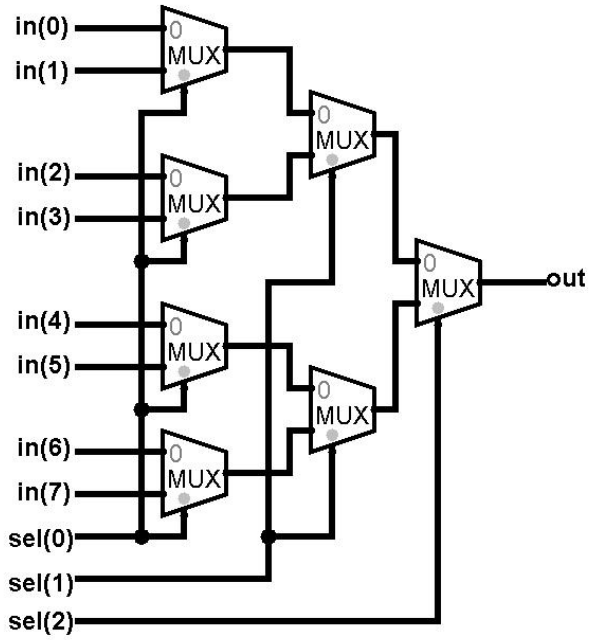
1.2. Decoder (Parameterized)

This is a parameterized tree-shaped decoder. The schematics will be given for an example 3-bit input decoder. Both the input size and the bit width of each input can be given as a parameter. You can for example create a 4-bit 3 by 2^3, 8-bit 2 by 2^2, or other types of decoders with variable input size and width.



1.3. Multiplexer (Parameterized)

This is a parameterized tree-shaped multiplexer. The schematics will be given for an example 8-input multiplexer. Both the input size and the bit width of each input can be given as a parameter. You can for example create 4-bit 8 to 1, 8-bit 4 to 1 or other types of multiplexers with variable input size and width.



1.4. CLOG2 Function

This is a simple function that calculates logarithm of a given input. It can calculate the result of up to 64. It is basically a comparator that compares the input with the powers of 2. It is not an actual logarithm function, but it can calculate required bit width for a given input size.

2. Register File Implementation

Here I will first explain the design of a simple parameterized register first. Then I will show the design of a structural register file and give simulation results. Lastly I have also included the design of a behavioral register file. Although the synthesized circuits may differ a good way to check correctness of the design is comparing the behavioral and structural implementations and seeing whether their results match exactly.

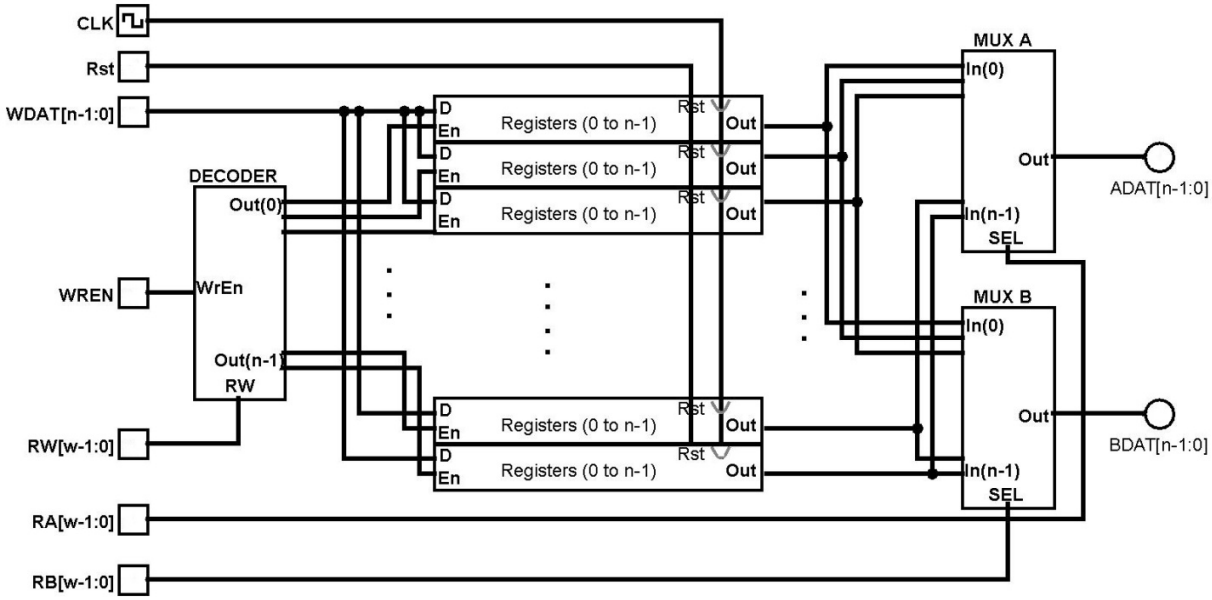
2.1. Register (Parameterized)

For a register, it is possible to use an array of D-Flip Flops using the “generate” statement, but it is cleaner to use the register format used in XST. Knowledge of the “generate” statement is already demonstrated in the previous assignment and within some of the other structures that I implemented for this assignment. The difference from the previous assignment is that this time the register implementation also includes an enable signal which controls whether the contents of the flip-flops will be updated or not in the next clock cycle. The enable signal is used in the register file implementation to write to some specific register cell. This is explained further in the next subsection.

2.2. Register File (Parameterized)

2.2.1. Design

The register file is implemented according to the specifications given in the assignment. Two parameters are passed to the design. The first one is the bit size, which is the number of bits in each register cell, and the second one is the address width, which determines the total number of register cells in the register file. For example a 16-word 16 bit register file can be generated by using 16 as bit size and 4 as address width ($2^4 = 16$). The parameterized design automatically generates the needed number of register cells, two multiplexers for the output and one decoder for the input of the appropriate size. Clock and reset signals are directly inserted to each register cell. Reset behavior is such that each register cell is updated with all "0"s. In this design to be able to reset just a single register cell, you need to write that location with all "0"s.



2.2.2. Simulation

Since showing the waveform of this would take too much space, I am going to use the "display" function to print out the results after read and write operations to specific cells of the register file. At the beginning of the outputs I have also used "$time" function to show the current simulation time. As can be seen a 5 ns delay is added so that the simulation result can be correctly read (If read, write operations and display operations are done at the same time, the simulator may first update then display or first display and then update. The order of operations in the same simulation time is not specified in the Verilog standard and can be arbitrary; alas this is not the behavior we want to see.). Since I have also included the Verilog test bench file, the reader may refer to there. The trace will contain the following. First reset is asserted. Then we read from each 16 locations to check that reset works as expected. I decided

to read from index 0 to index 15. Then we write to each 16 locations random data using "$random" and read from each 16 locations from the same address from two ports. For ease of reading the values in the registers are printed as decimals. It is possible to change it to binary by just updating "%d" with "%b" in the monitor function. The trace is as follows:

```
Simulator is doing circuit initialization process.
0 Simulation is started.
Finished circuit initialization process.
10 Reset is Asserted
25 Port A: Read Address = 0, Read Value = 0
Port B: Read Address = 0, Read Value = 0
45 Port A: Read Address = 1, Read Value = 0
Port B: Read Address = 1, Read Value = 0
65 Port A: Read Address = 2, Read Value = 0
Port B: Read Address = 2, Read Value = 0
85 Port A: Read Address = 3, Read Value = 0
Port B: Read Address = 3, Read Value = 0
105 Port A: Read Address = 4, Read Value = 0
Port B: Read Address = 4, Read Value = 0
125 Port A: Read Address = 5, Read Value = 0
Port B: Read Address = 5, Read Value = 0
145 Port A: Read Address = 6, Read Value = 0
Port B: Read Address = 6, Read Value = 0
165 Port A: Read Address = 7, Read Value = 0
Port B: Read Address = 7, Read Value = 0
185 Port A: Read Address = 8, Read Value = 0
Port B: Read Address = 8, Read Value = 0
205 Port A: Read Address = 9, Read Value = 0
Port B: Read Address = 9, Read Value = 0
225 Port A: Read Address = 10, Read Value = 0
Port B: Read Address = 10, Read Value = 0
245 Port A: Read Address = 11, Read Value = 0
Port B: Read Address = 11, Read Value = 0
265 Port A: Read Address = 12, Read Value = 0
Port B: Read Address = 12, Read Value = 0
285 Port A: Read Address = 13, Read Value = 0
Port B: Read Address = 13, Read Value = 0
305 Port A: Read Address = 14, Read Value = 0
Port B: Read Address = 14, Read Value = 0
325 Port A: Read Address = 15, Read Value = 0
Port B: Read Address = 15, Read Value = 0
345 Now we will write to each register location some random data.
365 Write Address = 0, Written Value = 13604
385 Write Address = 1, Written Value = 24193
405 Write Address = 2, Written Value = 54793
425 Write Address = 3, Written Value = 22115
445 Write Address = 4, Written Value = 31501
465 Write Address = 5, Written Value = 39309
485 Write Address = 6, Written Value = 33893
505 Write Address = 7, Written Value = 21010
525 Write Address = 8, Written Value = 58113
545 Write Address = 9, Written Value = 52493
565 Write Address = 10, Written Value = 61814
585 Write Address = 11, Written Value = 52541
605 Write Address = 12, Written Value = 22509
625 Write Address = 13, Written Value = 63372
645 Write Address = 14, Written Value = 59897
665 Write Address = 15, Written Value = 9414
685 Now we will read from each register location the data that we have written from each port.
725 Port A: Read Address = 0, Read Value = 13604
Port B: Read Address = 0, Read Value = 13604
745 Port A: Read Address = 1, Read Value = 24193
```

Port B: Read Address = 1, Read Value = 24193
765 Port A: Read Address = 2, Read Value = 54793
Port B: Read Address = 2, Read Value = 54793
785 Port A: Read Address = 3, Read Value = 22115
Port B: Read Address = 3, Read Value = 22115
805 Port A: Read Address = 4, Read Value = 31501
Port B: Read Address = 4, Read Value = 31501
825 Port A: Read Address = 5, Read Value = 39309
Port B: Read Address = 5, Read Value = 39309
845 Port A: Read Address = 6, Read Value = 33893
Port B: Read Address = 6, Read Value = 33893
865 Port A: Read Address = 7, Read Value = 21010
Port B: Read Address = 7, Read Value = 21010
885 Port A: Read Address = 8, Read Value = 58113
Port B: Read Address = 8, Read Value = 58113
905 Port A: Read Address = 9, Read Value = 52493
Port B: Read Address = 9, Read Value = 52493
925 Port A: Read Address = 10, Read Value = 61814
Port B: Read Address = 10, Read Value = 61814
945 Port A: Read Address = 11, Read Value = 52541
Port B: Read Address = 11, Read Value = 52541
965 Port A: Read Address = 12, Read Value = 22509
Port B: Read Address = 12, Read Value = 22509
985 Port A: Read Address = 13, Read Value = 63372
Port B: Read Address = 13, Read Value = 63372
1005 Port A: Read Address = 14, Read Value = 59897
Port B: Read Address = 14, Read Value = 59897
1025 Port A: Read Address = 15, Read Value = 9414
Port B: Read Address = 15, Read Value = 9414
1025 End of Simulation.

As can be seen from the simulation the register file works as expected. After reset is asserted we see that from time 20 ns to 320 ns all the values read are 0's. Then until time 680 random values are inserted to each location in the register file and in the remaining simulation time those values are read from both ports correctly.

2.3. Register File Behavioral (Parameterized)

2.3.1. Design

There is not much to say for the design of this. It is pretty straightforward. Interested readers may refer to the file called "regfileparam_behav.v" for further insight.

2.3.2. Simulation

Here I have included the XOR of the read ports from the behavioral and structural representations of the register file. Basically if they are correct, the XOR should always return all 0's. I have added additional lines to the simulation during read operations that also outputs the comparison of read ports. The trace is as follows:

Simulator is doing circuit initialization process.
0 Simulation is started.
Finished circuit initialization process.
10 Reset is Asserted
45 Port A: Read Address = 0, Read Value = 0
Port B: Read Address = 0, Read Value = 0

Compare Port A: 0000000000000000, Compare Port B: 0000000000000000
65 Port A: Read Address = 1, Read Value = 0
Port B: Read Address = 1, Read Value = 0
Compare Port A: 0000000000000000, Compare Port B: 0000000000000000
85 Port A: Read Address = 2, Read Value = 0
Port B: Read Address = 2, Read Value = 0
Compare Port A: 0000000000000000, Compare Port B: 0000000000000000
105 Port A: Read Address = 3, Read Value = 0
Port B: Read Address = 3, Read Value = 0
Compare Port A: 0000000000000000, Compare Port B: 0000000000000000
125 Port A: Read Address = 4, Read Value = 0
Port B: Read Address = 4, Read Value = 0
Compare Port A: 0000000000000000, Compare Port B: 0000000000000000
145 Port A: Read Address = 5, Read Value = 0
Port B: Read Address = 5, Read Value = 0
Compare Port A: 0000000000000000, Compare Port B: 0000000000000000
165 Port A: Read Address = 6, Read Value = 0
Port B: Read Address = 6, Read Value = 0
Compare Port A: 0000000000000000, Compare Port B: 0000000000000000
185 Port A: Read Address = 7, Read Value = 0
Port B: Read Address = 7, Read Value = 0
Compare Port A: 0000000000000000, Compare Port B: 0000000000000000
205 Port A: Read Address = 8, Read Value = 0
Port B: Read Address = 8, Read Value = 0
Compare Port A: 0000000000000000, Compare Port B: 0000000000000000
225 Port A: Read Address = 9, Read Value = 0
Port B: Read Address = 9, Read Value = 0
Compare Port A: 0000000000000000, Compare Port B: 0000000000000000
245 Port A: Read Address = 10, Read Value = 0
Port B: Read Address = 10, Read Value = 0
Compare Port A: 0000000000000000, Compare Port B: 0000000000000000
265 Port A: Read Address = 11, Read Value = 0
Port B: Read Address = 11, Read Value = 0
Compare Port A: 0000000000000000, Compare Port B: 0000000000000000
285 Port A: Read Address = 12, Read Value = 0
Port B: Read Address = 12, Read Value = 0
Compare Port A: 0000000000000000, Compare Port B: 0000000000000000
305 Port A: Read Address = 13, Read Value = 0
Port B: Read Address = 13, Read Value = 0
Compare Port A: 0000000000000000, Compare Port B: 0000000000000000
325 Port A: Read Address = 14, Read Value = 0
Port B: Read Address = 14, Read Value = 0
Compare Port A: 0000000000000000, Compare Port B: 0000000000000000
345 Port A: Read Address = 15, Read Value = 0
Port B: Read Address = 15, Read Value = 0
Compare Port A: 0000000000000000, Compare Port B: 0000000000000000
345 Now we will write to each register location some random data.
385 Write Address = 0, Written Value = 13604
405 Write Address = 1, Written Value = 24193
425 Write Address = 2, Written Value = 54793
445 Write Address = 3, Written Value = 22115
465 Write Address = 4, Written Value = 31501
485 Write Address = 5, Written Value = 39309
505 Write Address = 6, Written Value = 33893
525 Write Address = 7, Written Value = 21010
545 Write Address = 8, Written Value = 58113
565 Write Address = 9, Written Value = 52493
585 Write Address = 10, Written Value = 61814
605 Write Address = 11, Written Value = 52541
625 Write Address = 12, Written Value = 22509
645 Write Address = 13, Written Value = 63372
665 Write Address = 14, Written Value = 59897
685 Write Address = 15, Written Value = 9414
685 Now we will read from each register location the data that we have written from each port.

725 Port A: Read Address = 0, Read Value = 13604
Port B: Read Address = 0, Read Value = 13604
Compare Port A: 0000000000000000, Compare Port B: 0000000000000000
745 Port A: Read Address = 1, Read Value = 24193
Port B: Read Address = 1, Read Value = 24193
Compare Port A: 0000000000000000, Compare Port B: 0000000000000000
765 Port A: Read Address = 2, Read Value = 54793
Port B: Read Address = 2, Read Value = 54793
Compare Port A: 0000000000000000, Compare Port B: 0000000000000000
785 Port A: Read Address = 3, Read Value = 22115
Port B: Read Address = 3, Read Value = 22115
Compare Port A: 0000000000000000, Compare Port B: 0000000000000000
805 Port A: Read Address = 4, Read Value = 31501
Port B: Read Address = 4, Read Value = 31501
Compare Port A: 0000000000000000, Compare Port B: 0000000000000000
825 Port A: Read Address = 5, Read Value = 39309
Port B: Read Address = 5, Read Value = 39309
Compare Port A: 0000000000000000, Compare Port B: 0000000000000000
845 Port A: Read Address = 6, Read Value = 33893
Port B: Read Address = 6, Read Value = 33893
Compare Port A: 0000000000000000, Compare Port B: 0000000000000000
865 Port A: Read Address = 7, Read Value = 21010
Port B: Read Address = 7, Read Value = 21010
Compare Port A: 0000000000000000, Compare Port B: 0000000000000000
885 Port A: Read Address = 8, Read Value = 58113
Port B: Read Address = 8, Read Value = 58113
Compare Port A: 0000000000000000, Compare Port B: 0000000000000000
905 Port A: Read Address = 9, Read Value = 52493
Port B: Read Address = 9, Read Value = 52493
Compare Port A: 0000000000000000, Compare Port B: 0000000000000000
925 Port A: Read Address = 10, Read Value = 61814
Port B: Read Address = 10, Read Value = 61814
Compare Port A: 0000000000000000, Compare Port B: 0000000000000000
945 Port A: Read Address = 11, Read Value = 52541
Port B: Read Address = 11, Read Value = 52541
Compare Port A: 0000000000000000, Compare Port B: 0000000000000000
965 Port A: Read Address = 12, Read Value = 22509
Port B: Read Address = 12, Read Value = 22509
Compare Port A: 0000000000000000, Compare Port B: 0000000000000000
985 Port A: Read Address = 13, Read Value = 63372
Port B: Read Address = 13, Read Value = 63372
Compare Port A: 0000000000000000, Compare Port B: 0000000000000000
1005 Port A: Read Address = 14, Read Value = 59897
Port B: Read Address = 14, Read Value = 59897
Compare Port A: 0000000000000000, Compare Port B: 0000000000000000
1025 Port A: Read Address = 15, Read Value = 9414
Port B: Read Address = 15, Read Value = 9414
Compare Port A: 0000000000000000, Compare Port B: 0000000000000000
1025 End of Simulation

As can be seen from the simulation trace, everything looks okay.

3. ALU Implementation

First I start with the implementation of the 16-bit flag register. The next part is implementation of the ALU structurally. Note that just the BITSIZE of the ALU is parameterized but the arithmetic and logic operations are not (I am not sure even if that is possible). Again I have shown the simulation results with a few different inputs for each function of the ALU. Lastly I

have implemented the behavioral ALU and again compared its output with the structural one just as I did with the register file.

3.1. 16-bit Flag Register

The flag outputs of the ALU may be used in the current clock cycle. In this case we won't need a register to hold the flags, but if we plan to use the flags of some previous instruction we need a register to hold those values. For example let's assume we have the following assembly instructions.

"cmp reg1, reg2;"

"jne label1;"

Depending on the implementation of the microprocessor, you may need more than a single clock cycle to do the two things. In this case "cmp" instruction will set the "zero" flag to 0 or 1, and on the next clock cycle the "jne" instruction will check that same "zero" flag and decide whether to branch to "label1" or not accordingly.

3.2. ALU (Parameterized)

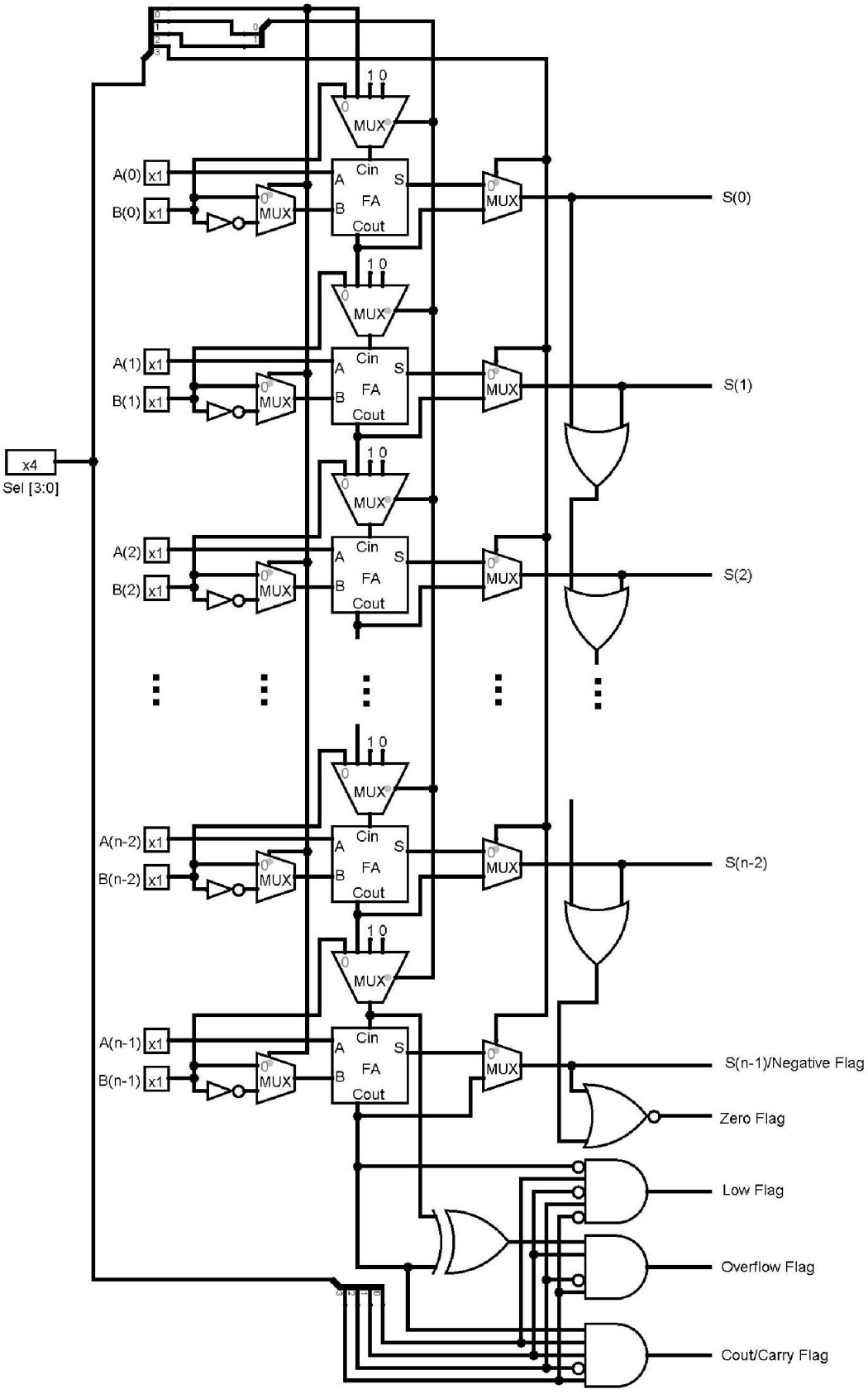
3.2.1. Design

The ALU is implemented according to the specifications given in the assignment. One parameter is passed to the design. It is the size of the inputs A and B. If you for example pass 16 as parameter, it will generate a 16-bit ALU. The main ALU part is carry ripple adder based and is implemented as shown on the assignment sheet. In addition to those the 5 flags that are mandatory were added. Those are as follows.

1. Carry Flag: This is directly the carry out of the last full adder in the circuit.
2. Negative Flag: This flag shows whether the result is negative if the calculation is interpreted as a signed calculation. The sum result from the last adder determines this flag.
3. Overflow Flag: The overflow flag checks whether there has been an overflow during the arithmetic operation. The XOR of the carry in and carry out of the last adder can be used to determine this value.
4. Zero Flag: The zero flag turns on if all of the sum values from the full adders are 0. This means that we need a cascading or circuit and at the end negate the result to determine this flag. Normally one might go for a tree-shaped or circuit in this example, but since our ALU is already carry ripple adder based, there is no benefit from using such an approach.
5. Low Flag: The low flag turns on when the signed representation of the first operand of the ALU is less than the second operand. This is set on "cmp" operations. The basic

"cmp" operation does subtraction and therefore on signed arithmetic we expect to see no carry on the last output if the second operand is larger.

Carry Flag and Overflow Flag are set to 0 for all logical operations. I also decided to set carry flag for subtraction to 0, since we are doing 2's complement subtraction and the carry flag has no meaning in signed arithmetic operations. Also Low Flag is only set when doing a subtraction operation.


x4
Sel [3:0]
A(0)
B(0)
A(1)
B(1)
A(2)
B(2)
A(n-2)
B(n-2)
A(n-1)
B(n-1)
x1
MUX
Cin
A
B
FA
S
Cout
S(0)
S(1)
S(2)
S(n-2)
S(n-1)/Negative Flag
Zero Flag
Low Flag
Overflow Flag
Cout/Carry Flag

3.2.2. Simulation

Here I have tested each function with a couple of different inputs that check all the possible combinations for the flags. I have both checked the results of the operation and the flags resulting from the operation. Again I have used the "$display" function to create the output. For logical operations the inputs A, B and the outputs Result and Flags is shown in binary format. For addition and subtraction the inputs A, B and the output Result is shown in decimal format for readability. The flag remains in binary format. The trace is as follows:

Simulator is doing circuit initialization process.
0 Start of Simulation.
Flags: xXXXXXXXXNZOXXLXC
Finished circuit initialization process.

0 XOR Operations Can Only Set Zero or Negative Flags.

20 Input A: 0011010100100100, Input B: 0011010100100100, Result: 0000000000000000, Flags: 0000000001000000, OP: XOR ON SAME INPUT

40 Input A: 0101111010000001, Input B: 1010000101111110, Result: 1111111111111111, Flags: 0000000010000000, OP: XOR ON ALL DIFFERENT INPUT

60 Input A: 1101011000001001, Input B: 0101011001100011, Result: 1000000001101010, Flags: 0000000010000000, OP: XOR ON RANDOM INPUT

60 XNOR Operations Can Only Set Zero or Negative Flags.

80 Input A: 0111101100001101, Input B: 0111101100001101, Result: 1111111111111111, Flags: 0000000010000000, OP: XNOR ON SAME INPUT

100 Input A: 1001100110001101, Input B: 0110011001110010, Result: 0000000000000000, Flags: 0000000001000000, OP: XNOR ON ALL DIFFERENT INPUT

120 Input A: 1000010001100101, Input B: 0101001000010010, Result: 0010100110001000, Flags: 0000000000000000, OP: XNOR ON RANDOM INPUT

120 NOT Operations Can Only Set Zero or Negative Flags.

140 Input A: 0000000000000000, Input B: 1110001100000001, Result: 1111111111111111, Flags: 0000000010000000, OP: NOT A ON ALL 0's

160 Input A: 1111111111111111, Input B: 1100110100001101, Result: 0000000000000000, Flags: 0000000001000000, OP: NOT A ON ALL 1's

180 Input A: 1111000101110110, Input B: 1100110100111101, Result: 0000111010001001, Flags: 0000000000000000, OP: NOT A ON RANDOM INPUT

180 AND Operations Can Only Set Zero or Negative Flags.

200 Input A: 0101011111101101, Input B: 0101011111101101, Result: 0101011111101101, Flags: 0000000000000000, OP: AND ON ALL SAME INPUT

220 Input A: 1111011110001100, Input B: 0000100001110011, Result: 0000000000000000, Flags: 0000000001000000, OP: AND ON ALL DIFFERENT INPUT

240 Input A: 1110100111111001, Input B: 0010010011000110, Result: 0010000011000000, Flags: 0000000000000000, OP: AND ON RANDOM INPUT

260 Input A: 1000010011000101, Input B: 1101001010101010, Result: 1000000010000000, Flags: 0000000010000000, OP: AND ON RANDOM INPUT WITH FIRST BITS = 1

260 OR Operations Can Only Set Zero or Negative Flags.

280 Input A: 1111011111100101, Input B: 1111011111100101, Result: 1111011111100101, Flags: 0000000010000000, OP: OR ON ALL SAME INPUT

300 Input A: 0111001001110111, Input B: 1000110110001000, Result: 1111111111111111, Flags: 0000000010000000, OP: OR ON ALL DIFFERENT INPUT

320 Input A: 0000000000000000, Input B: 0000000000000000, Result: 0000000000000000, Flags: 0000000001000000, OP: OR ON ALL 0's

320 ADD Operations Can Set Zero, Carry, Overflow and Negative Flags.

340 Input A: -10734, Input B: -9329, Result: -20063, Flags: 0000000010000001, OP: ADD ON RANDOM INPUT (SIGNED)

360 Input A: 32767, Input B: 32767, Result: -2, Flags: 0000000010100000, OP: ADD THAT CREATES OVERFLOW (SIGNED)

380 Input A: -30000, Input B: -20000, Result: 15536, Flags: 0000000000100001, OP: ADD THAT CREATES OVERFLOW (SIGNED)

400 Input A: 65535, Input B: 45055, Result: 45054, Flags: 0000000010000001, OP: ADD THAT CREATES CARRY (UNSIGNED)

420 Input A: 4095, Input B: 4095, Result: 8190, Flags: 0000000000000000, OP: ADD THAT DOES NOT CREATE CARRY (UNSIGNED)

440 Input A: 30000, Input B: -30000, Result: 0, Flags: 0000000001000001, OP: ADD THAT CHECKS ZERO FLAG (SIGNED)

440 SUB Operations Can Set Zero, Low, Overflow and Negative Flags.

460 Input A: 27122, Input B: -26930, Result: -11484, Flags: 0000000010100100, OP: SUB ON RANDOM INPUT

480 Input A: 30000, Input B: -20000, Result: -15536, Flags: 0000000010100100, OP: SUB THAT CREATES OVERFLOW

500 Input A: -20000, Input B: 30000, Result: 15536, Flags: 0000000000100000, OP: SUB THAT CREATES OVERFLOW

520 Input A: 100, Input B: 2000, Result: 63636, Flags: 0000000010000100, OP: SUB THAT SETS LOW FLAG (UNSIGNED)

540 Input A: 256, Input B: 65535, Result: 257, Flags: 0000000000000100, OP: SUB THAT SETS LOW FLAG (UNSIGNED)

560 Input A: 1000, Input B: 500, Result: 500, Flags: 0000000000000000, OP: SUB THAT RESETS LOW FLAG (UNSIGNED)

580 Input A: 65535, Input B: 16, Result: 65519, Flags: 0000000010000000, OP: SUB THAT RESETS LOW FLAG (UNSIGNED)

600 Input A: -30000, Input B: -30000, Result: 0, Flags: 0000000001000000, OP: SUB THAT CREATES ZERO

600 End of Simulation.

Here we see that all the results are correct. And the flags are also set accordingly. At the beginning of the simulation the location of the flags is given as XXXXXXXXNZOXXLXC. Here C corresponds to carry, L to low, O to overflow, Z to zero and N to negative (sign) flags.

3.3. ALU Behavioral (Parameterized)

3.3.1. Design

There is not much to say for the design of this. It is pretty straightforward. Interested readers may refer to the file called “aluparam_behav.v” for further insight. I have used the case statement for each arithmetic or logical operation, and depending on the type of the operation some signed number conversions are made.

3.3.2. Simulation

Here similar to the testing of the register file I have again included the XOR of the outputs and flags from the behavioral and structural representations of the register file. Basically if they are correct, the XOR should always return all 0’s. The trace is as follows:

Simulator is doing circuit initialization process.
0 Start of Simulation.
Flags: xXXXXXXXXNZOXXLXC
Finished circuit initialization process.

0 XOR Operations Can Only Set Zero or Negative Flags.

20 Input A: 0011010100100100, Input B: 0011010100100100, Result: 0000000000000000,
Flags: 0000000001000000, OP: XOR ON SAME INPUT
Compare Flags: 0000000000000000, Compare Result: 0000000000000000

40 Input A: 0101111010000001, Input B: 1010000101111110, Result: 1111111111111111,
Flags: 0000000010000000, OP: XOR ON ALL DIFFERENT INPUT
Compare Flags: 0000000000000000, Compare Result: 0000000000000000

60 Input A: 1101011000001001, Input B: 0101011001100011, Result: 1000000001101010,
Flags: 0000000010000000, OP: XOR ON RANDOM INPUT
Compare Flags: 0000000000000000, Compare Result: 0000000000000000

60 XNOR Operations Can Only Set Zero or Negative Flags.

80 Input A: 0111101100001101, Input B: 0111101100001101, Result: 1111111111111111,
Flags: 0000000010000000, OP: XNOR ON SAME INPUT
Compare Flags: 0000000000000000, Compare Result: 0000000000000000

100 Input A: 1001100110001101, Input B: 0110011001110010, Result: 0000000000000000,
Flags: 0000000001000000, OP: XNOR ON ALL DIFFERENT INPUT
Compare Flags: 0000000000000000, Compare Result: 0000000000000000

120 Input A: 1000010001100101, Input B: 0101001000010010, Result: 0010100110001000,
Flags: 0000000000000000, OP: XNOR ON RANDOM INPUT
Compare Flags: 0000000000000000, Compare Result: 0000000000000000

120 NOT Operations Can Only Set Zero or Negative Flags.

140 Input A: 0000000000000000, Input B: 1110001100000001, Result: 1111111111111111,
Flags: 0000000010000000, OP: NOT A ON ALL 0's
Compare Flags: 0000000000000000, Compare Result: 0000000000000000

160 Input A: 1111111111111111, Input B: 1100110100001101, Result: 0000000000000000,
Flags: 0000000001000000, OP: NOT A ON ALL 1's
Compare Flags: 0000000000000000, Compare Result: 0000000000000000

180 Input A: 1111000101110110, Input B: 1100110100111101, Result: 0000111010001001,
Flags: 0000000000000000, OP: NOT A ON RANDOM INPUT

Compare Flags: 0000000000000000, Compare Result: 0000000000000000

180 AND Operations Can Only Set Zero or Negative Flags.

200 Input A: 0101011111101101, Input B: 0101011111101101, Result: 0101011111101101, Flags: 0000000000000000, OP: AND ON ALL SAME INPUT
Compare Flags: 0000000000000000, Compare Result: 0000000000000000

220 Input A: 1111011110001100, Input B: 0000100001110011, Result: 0000000000000000, Flags: 0000000001000000, OP: AND ON ALL DIFFERENT INPUT
Compare Flags: 0000000000000000, Compare Result: 0000000000000000

240 Input A: 1110100111111001, Input B: 0010010011000110, Result: 0010000011000000, Flags: 0000000000000000, OP: AND ON RANDOM INPUT
Compare Flags: 0000000000000000, Compare Result: 0000000000000000

260 Input A: 1000010011000101, Input B: 1101001010101010, Result: 1000000010000000, Flags: 0000000010000000, OP: AND ON RANDOM INPUT WITH FIRST BITS = 1
Compare Flags: 0000000000000000, Compare Result: 0000000000000000

260 OR Operations Can Only Set Zero or Negative Flags.

280 Input A: 1111011111100101, Input B: 1111011111100101, Result: 1111011111100101, Flags: 0000000010000000, OP: OR ON ALL SAME INPUT
Compare Flags: 0000000000000000, Compare Result: 0000000000000000

300 Input A: 0111001001110111, Input B: 1000110110001000, Result: 1111111111111111, Flags: 0000000010000000, OP: OR ON ALL DIFFERENT INPUT
Compare Flags: 0000000000000000, Compare Result: 0000000000000000

320 Input A: 0000000000000000, Input B: 0000000000000000, Result: 0000000000000000, Flags: 0000000001000000, OP: OR ON ALL 0's
Compare Flags: 0000000000000000, Compare Result: 0000000000000000

320 ADD Operations Can Set Zero, Carry, Overflow and Negative Flags.

340 Input A: -10734, Input B: -9329, Result: -20063, Flags: 0000000010000001, OP: ADD ON RANDOM INPUT (SIGNED)
Compare Flags: 0000000000000000, Compare Result: 0000000000000000

360 Input A: 32767, Input B: 32767, Result: -2, Flags: 0000000010100000, OP: ADD THAT CREATES OVERFLOW (SIGNED)
Compare Flags: 0000000000000000, Compare Result: 0000000000000000

380 Input A: -30000, Input B: -20000, Result: 15536, Flags: 0000000000100001, OP: ADD THAT CREATES OVERFLOW (SIGNED)
Compare Flags: 0000000000000000, Compare Result: 0000000000000000

400 Input A: 65535, Input B: 45055, Result: 45054, Flags: 0000000010000001, OP: ADD THAT CREATES CARRY (UNSIGNED)
Compare Flags: 0000000000000000, Compare Result: 0000000000000000

420 Input A: 4095, Input B: 4095, Result: 8190, Flags: 0000000000000000, OP: ADD THAT DOES NOT CREATE CARRY (UNSIGNED)
Compare Flags: 0000000000000000, Compare Result: 0000000000000000

440 Input A: 30000, Input B: -30000, Result: 0, Flags: 0000000001000001, OP: ADD THAT CHECKS ZERO FLAG (SIGNED)
Compare Flags: 0000000000000000, Compare Result: 0000000000000000

440 SUB Operations Can Set Zero, Low, Overflow and Negative Flags.

460 Input A: 27122, Input B: -26930, Result: -11484, Flags: 0000000010100100, OP: SUB ON RANDOM INPUT

Compare Flags: 0000000000000000, Compare Result: 0000000000000000

480 Input A: 30000, Input B: -20000, Result: -15536, Flags: 0000000010100100, OP: SUB THAT CREATES OVERFLOW
Compare Flags: 0000000000000000, Compare Result: 0000000000000000

500 Input A: -20000, Input B: 30000, Result: 15536, Flags: 0000000000100000, OP: SUB THAT CREATES OVERFLOW
Compare Flags: 0000000000000000, Compare Result: 0000000000000000

520 Input A: 100, Input B: 2000, Result: 63636, Flags: 0000000010000100, OP: SUB THAT SETS LOW FLAG (UNSIGNED)
Compare Flags: 0000000000000000, Compare Result: 0000000000000000

540 Input A: 256, Input B: 65535, Result: 257, Flags: 0000000000000100, OP: SUB THAT SETS LOW FLAG (UNSIGNED)
Compare Flags: 0000000000000000, Compare Result: 0000000000000000

560 Input A: 1000, Input B: 500, Result: 500, Flags: 0000000000000000, OP: SUB THAT RESETS LOW FLAG (UNSIGNED)
Compare Flags: 0000000000000000, Compare Result: 0000000000000000

580 Input A: 65535, Input B: 16, Result: 65519, Flags: 0000000010000000, OP: SUB THAT RESETS LOW FLAG (UNSIGNED)
Compare Flags: 0000000000000000, Compare Result: 0000000000000000

600 Input A: -30000, Input B: -30000, Result: 0, Flags: 0000000001000000, OP: SUB THAT CREATES ZERO
Compare Flags: 0000000000000000, Compare Result: 0000000000000000

600 End of Simulation.

Again we see that every compare flag and compare result is all 0's. So the outputs of the structural and behavioral code match exactly. Although this does not guarantee that the results are correct, it greatly enhances the confidence in the design that two different approaches yield the same result.

4. Synthesized Simulations

Here you can see that the synthesized models work just as expected. We compare the results of the simulation with the results from the previous two chapters. Inside the "Verilog Code" folder I have included an additional folder called "Post-Synthesis" which contains the post synthesis simulation models of the register file and the ALU. I will only talk about the changes made to see the simulation. The simulation traces are **exactly the same** as in the previous section. So I am not going to repeat the traces in this section again. It will just take too much space. The interested reader may refer to the following subsections to make the necessary changes and observe the same result. Since I am not experienced in this, I am not sure whether the post-simulation synthesis models work without any changes on another computer (Logically as long as the same target device is selected there should be no problems). If that is not the case the reader may open the original structural designs and generate post-synthesis simulation models on their own by using Xilinx 14.7.

4.1. Register File

For simulating the register file post synthesis model only the following changes have to be made.

1. Add the post-synthesis model file to the project.
2. Change the following code inside the test bench file.
    a. regfileparam #(.BITSIZE(16), .ADDSIZE(4)) uut (
    b. .adat(adat),
    c. .bdat(bdat),
    d. .ra(ra),
    e. .rb(rb),
    f. .rw(rw),
    g. .wdat(wdat),
    h. .wren(wren),
    i. .clk(clk),
    j. .rst(rst)
    k.);
3. The changed code should be as follows:
    a. regfileparamsynthesis uut (
    b. .adat(adat),
    c. .bdat(bdat),
    d. .ra(ra),
    e. .rb(rb),
    f. .rw(rw),
    g. .wdat(wdat),
    h. .wren(wren),
    i. .clk(clk),
    j. .rst(rst)
    k.);

And as expected the simulation will run and generate the same result. This means that the simulation model matches our behavioral design (which I earlier tested against the structural design and got the same results).

4.2. ALU

For simulating the ALU post synthesis model again the same thing has to be done. The following change inside the test bench for the ALU has to be made.

1. Add the post synthesis simulation model to the project.
2. Change the following code inside the test bench file:

```
aluparam_behav #(.BITSIZE(16)) uut (
        .Y(Y),
        .flags(flags),
        .A(A),
        .B(B),
        .sel(sel)
);
```

3. The changed code should be as follows:

```
aluparamsynthesis uut (
        .Y(Y),
        .flags(flags),
        .A(A),
        .B(B),
        .sel(sel)
);
```

And as expected the simulation will run and generate the same result. This means that the simulation model matches our structural (this time) design (which I earlier tested against the behavioral design and got the same results).

5. Synthesis Outputs

	Register File	**Register File (Behavioral)**	**ALU**	**ALU (Behavioral)**
Advanced HDL Synthesis	• 256 x FF • 30 x 16 bit 2to1 MUX	• 256 x FF • 16 x 16 bit 2to1 MUX • 2 x 16 bit 16to1 MUX	• 48 x 1 bit 2to1 MUX • 33 x 1 bit XOR2	• 1 x 16 bit carry in adder • 1 x 17 bit adder • 1 x 16 bit comparator (>) • 4 x 32 bit comparator (>) • 1 x 16 bit XOR2
Number of Slice Registers	256	256	0	0

Number of Slice LUTs	149	385	52	101
Number used as Logic	149	385	52	101
Number of I/O	63	63	68	68
Total Delay	6.871 ns	6.871 ns	23.014 ns	10.100 ns

- Obviously as can be seen from the results the carry ripple adder based ALU performs poorly in times of total delay.
- Also the slice utilization of the behavioral code is bad versus the structural one. This is mainly due to the non-optimized code. Behavioral code is just used for verification purposes. Although I used synthesizable constructs, I did not utilize optimization methods such as operator and functionality sharing which can again be seen by the number of comparators or two different adders the behavioral ALU design is using.
- As expected the number of registers used in the ALU is 0 since this is just a combinational system. The number of slice register used in the register file is exactly equal to the number required by our design which is 256. This means we have a latch free Verilog code.
- In the following assignments behavioral ALU code will be optimized and used instead of the structural one.

Other Stuff:

The tree-shaped multiplexer design uses the non-standard two dimensional array construct. Xilinx 14.7 does support this, but if the reader uses another compiler that doesn't allow that, the reader may need to flatten the array before running synthesis.

References:

- BOUN - EE 540 - Spring 2015 - Assignment 2
- RTL Hardware Design Using VHDL, Pong P. Chu
- The Verilog Hardware Description Language, Thomas & Moorby
- http://teaching.idallen.com/dat2343/10f/notes/040_overflow.txt
- http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-884-complex-digital-systems-spring-2005/related-resources/parameter_models.pdf
- Digital Circuits and Systems, Video Lectures by Prof. S. Srinivasan
- www.bing.com