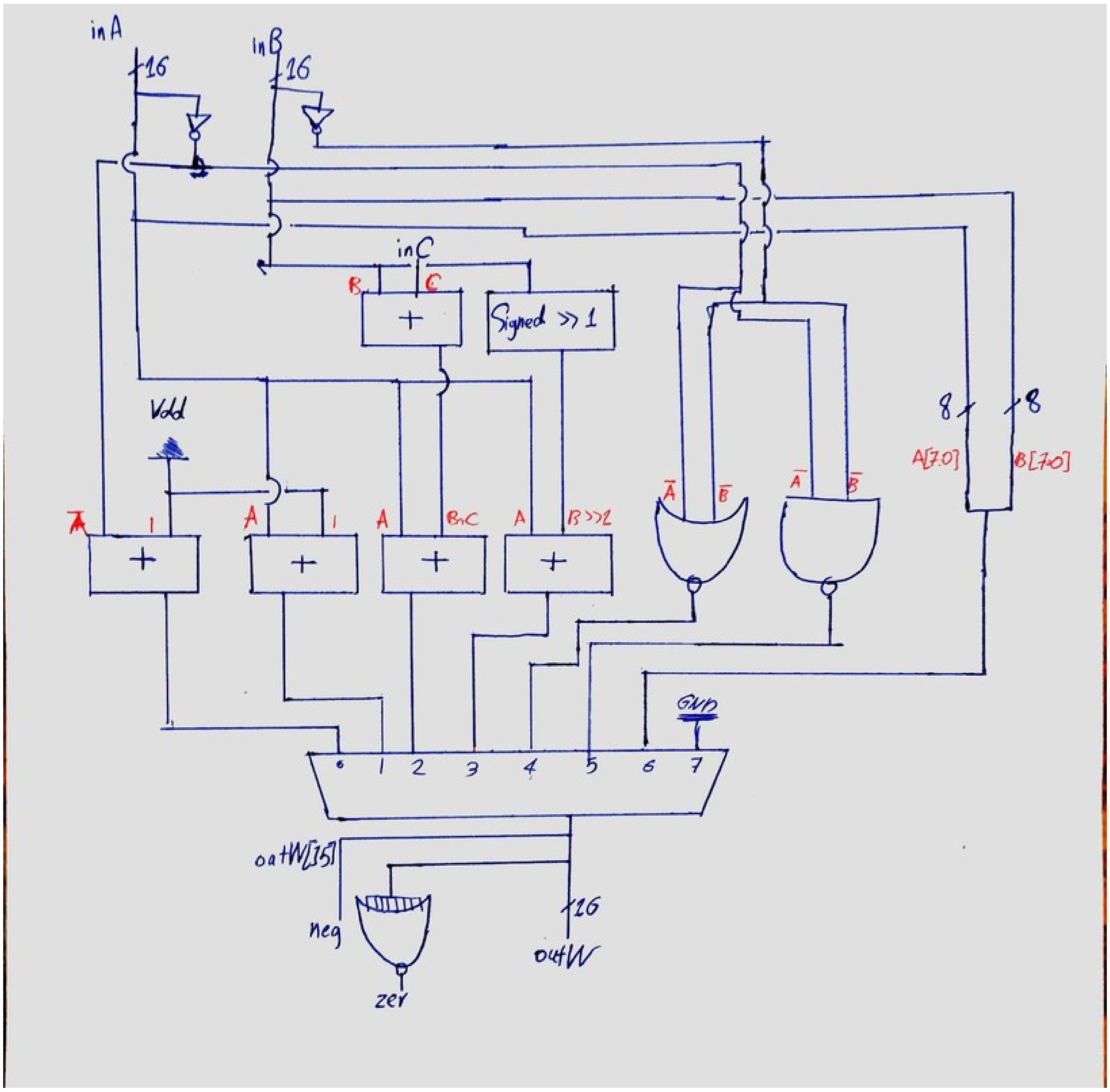
CA3

Question 1

Part B

For this question, we design our ALU without caring about optimizations:



Note: Since our input numbers and output are treated in 2's complement method, we won't have to change anything in add operation. The only thing that we should do is fix sign bit when shifting `inB` to right which is an easy fix:

```
reg [15:0] shiftedB;
assign shiftedB = inB >> 1;
outW = inA + ({inB[15], shiftedB[14:0]});
```

After synthesizing our Verilog description using Yosys, this is the report that Yosys gives us:

```
=== ALUL1 ===

Number of wires:                1213
Number of wire bits:            1274
Number of public wires:            8
Number of public wire bits:       69
Number of memories:                0
Number of memory bits:             0
Number of processes:               0
Number of cells:                 703
     NAND                        207
     NOR                         366
     NOT                         130
```

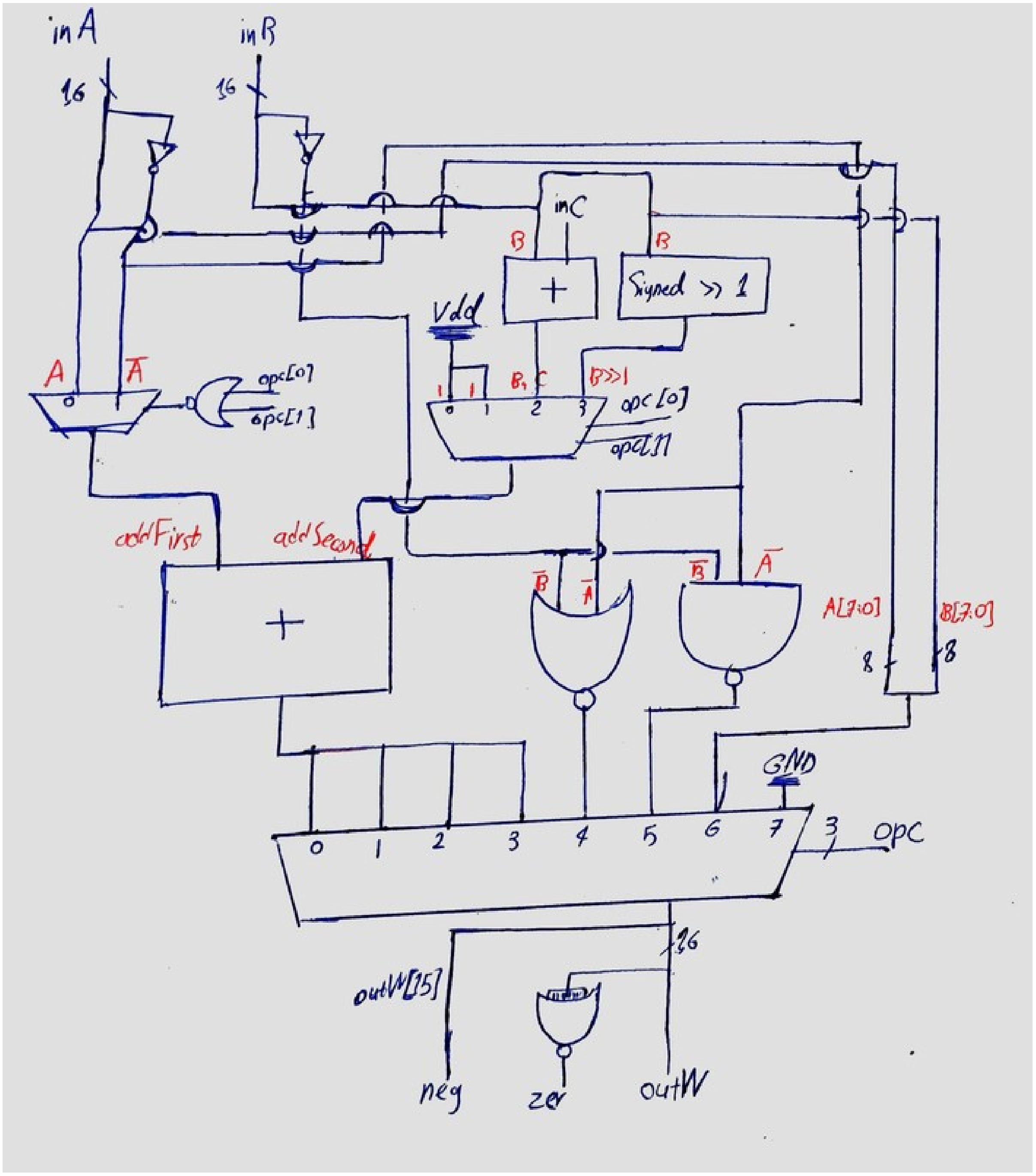
Our behavioral description was transformed to another Verilog description which has 703 cells. This is because we are using 5 adders which has a lot of delay and space.

Part C

Our `1-alu.v` file (the description that we have written) is less than 30 lines and has assign statements and always statements without delays. But `1-syn.v` file has more than 4000 lines of code and consists of thousands of wires and gates. So simulating synthesized file should take more time.

Question 2

The Verilog description for second question can be found in `2-alu.v` file. We'll discuss what we did here in question 3. For now, let's just take a look a t the diagram for our description:



Let's see how many gates have we saved:

```
=== ALUL2 ===

Number of wires:                940
Number of wire bits:           1001
Number of public wires:           8
Number of public wire bits:      69
Number of memories:               0
Number of memory bits:            0
Number of processes:              0
Number of cells:                563
     NAND                       171
     NOR                        279
     NOT                        113
```

Great, we have reduced the number of cells by 140.

Question 3

Optimizations

To see what we did in question 2, let's take a look at our ALU function first:

Operation Code	Function
000	2sCompl(inA)
001	inA + 1
010	inA + inB + inC
011	inA + inB * 0.5
100	inA & inB
101	inA or inB
110	{inA[7:0], inB[7:0]}
111	No Operation

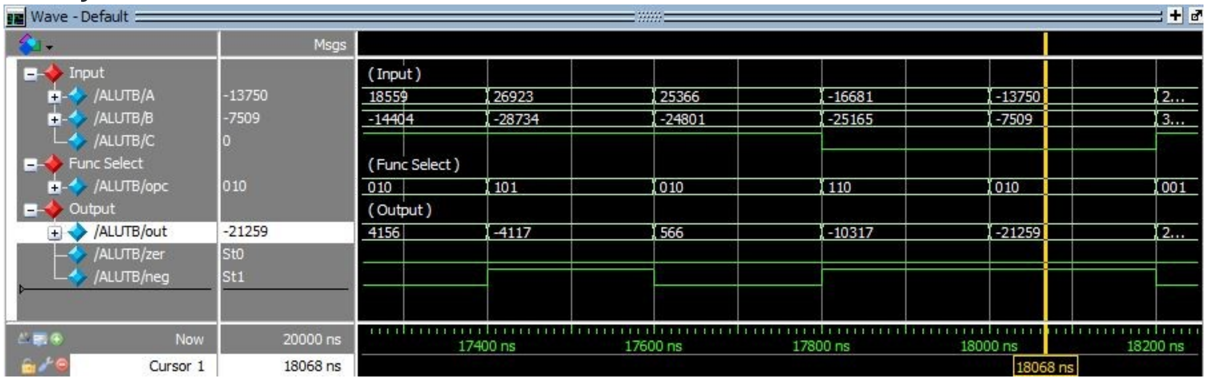
We have 5 add operations and for each one of them, Yosys is creating an adder component. We can reduce the number of adders with reusing adder components.

Note: in `opc: 010`, there are 2 add operations, even though `inC` is one bit carry input. We could create this functionality with only one adder with designing an adder component that supports carry input, but I guess it's not a goal in this assignment. To reuse our adder component, we should change it's inputs based on `opc`. Another thing we can do to reduce the number of cells, is to put addition to `default` arm of our switch case statement.

Waveforms

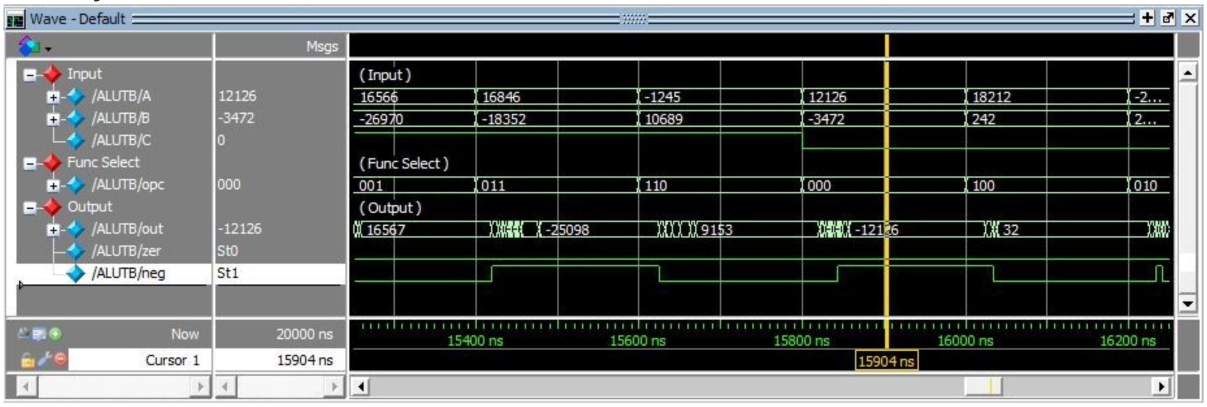
Question 1

Pre-Synthesis



```
Wave - Default
                    Msgs
Input
  /ALUTB/A          -13750
  /ALUTB/B          -7509
  /ALUTB/C          0
Func Select
  /ALUTB/opc        010
Output
  /ALUTB/out        -21259
  /ALUTB/zer        St0
  /ALUTB/neg        St1
Now                 20000 ns
Cursor 1            18068 ns
```

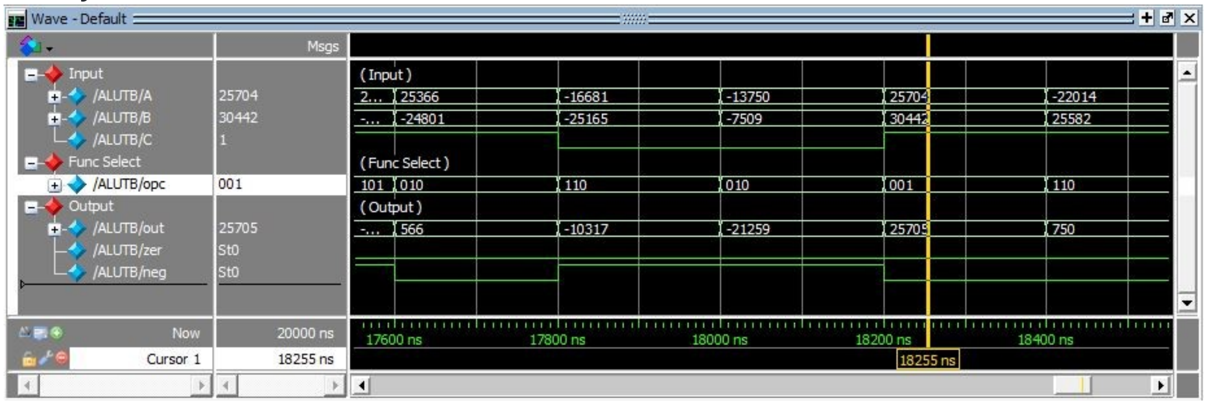
Post-Synthesis



```
Wave - Default
                    Msgs
Input
  /ALUTB/A          12126
  /ALUTB/B          -3472
  /ALUTB/C          0
Func Select
  /ALUTB/opc        000
Output
  /ALUTB/out        -12126
  /ALUTB/zer        St0
  /ALUTB/neg        St1
Now                 20000 ns
Cursor 1            15904 ns
```

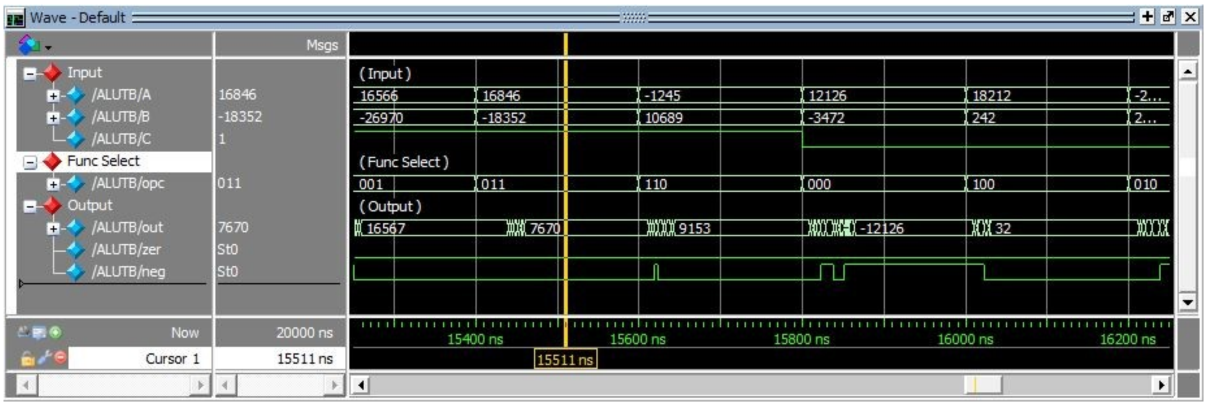
Question 2

Pre-Synthesis:



```
Wave - Default
                    Msgs
Input
  /ALUTB/A          25704
  /ALUTB/B          30442
  /ALUTB/C          1
Func Select
  /ALUTB/opc        001
Output
  /ALUTB/out        25705
  /ALUTB/zer        St0
  /ALUTB/neg        St0
Now                 20000 ns
Cursor 1            18255 ns
```

Post-Synthesis



Note: Since we didn't specified any delay values for our pre-synthesis Verilog description, there is no delay in pre-synthesis waveforms. However, the gates in `mycells.v` and `mycells.lib` do have a delay associated with them, so post-synthesis waveforms do have delays in them.

Timing and delays

Let's hand-calculate worst case delay for a 16 bit adder using given delays in library file:

NAND	NOR	NOT
5	5	3

A longest path in a single full adder is consists of two `NAND`s:

```
5 + 5 = 10 => 16 * 10 = 160
```

Also, two `XOR`s are required for the last sum result to be calculated.

We don't have an `XOR` module in our library, an `XOR` can be created with a two-level `NAND` circuit: `5 + 5 = 10` is the calculated delay for an `XOR` gate and `20` ns for two serial `XOR`s.

Therefore the delay for a 16 bit adder is `160 + 20 = 180ns`.

The worst case delay for such a complex circuit is hard to find, but the average delay time for my circuit is around `200` ns. So there are a few possiblites:

1. Yosis implemented an adder using full adders with a carry in input, so delay of

the longest path in my circuit would be `3To8Mux + 16BitAdder` that would match `200` ns delay.

2. Yosis used two serial adders for `inA + inB + inC`, but implemented the adders using fast adders.
3. Same as #2, but Yosis implemented the adders with a two-level nand login or something like that which I suppose has less delay but in trade-off is not cascadable and violates RTL standards.