Implementation of Instruction Fetch and Data Memory Logic

# Instruction Fetch

Before we implemented actual instruction fetch circuit we divided the task in multiple subtasks of different single circuit elements:

1. Adder
2. Left shifter by 2
3. Binary multiplexer
4. Program Counter
5. Instruction memory

And then after successfully testing each element of instruction fetch, we merged them together using descriptive VHDL.

## Adder

For adder we just used a “ieee.numeric\_std.all” library to do simple mathematical operations with STD\_LOGIC\_VECTORs. Literally we are just summing to vectors asynchronously.

For implementation please refer to “Adder/adder.vhd”

### Simulation and Conclusion

For simulation waveform results please refer to “Adder/adder\_waveform.pdf”.

For Test bench code please refer to “Adder/adder\_tb.vhd”.

The results from simulation were just like we expected, we tried summing up 0x28(40d) and 0x0a(10d) as a result we got 0x32(50d). Afterwards we summed up 0xbf883(784515d) and 0xae2b(44587d), and again we got correct result, which was 0xc036ae(12596910d). We can conclude that our simple adder implementation is working fine.

## Left Shifter by 2

Second component which we implemented was left shifter. At first we thought of making a n bit shifter, where our component could be reused for different amounts of shift, which would be specified by the user (engineer). However, after looking at the design schematic we concluded that for our task is not necessary to implement multi-purpose bit shifter. In the schematic we have to bit shifters twice, and both of the times we have to shift bits to the left by two.

For implementation of “Left shifter by 2” please refer to “Shift/shift\_left.vhd”

### Simulation & Conclusion

For simulation waveform results please refer to “shift/waveform.pdf”

For Test bench code please refer to “Adder/shift\_left\_tb.vhd”.

After putting some random numbers in the test bench code we can confirm that our “Left shifter by two” works exactly like expected. However we could have implemented slightly different as well. Instead of using “numeric\_all” library, we could have just specified the bits which we want to extract from the std\_logic\_vector and the code would look something like this:

Output <= input(29:0) & “00”;

Essentially that is the same thing, as far as we are concerned it doesn’t matter how you do the bit shifting, but our current approach allows us to improve the shifter to multi-purpose bit shifter, where user could specify n bits to be shifted, left or right.

## Binary multiplexer

Third component which we implemented was binary multiplexer also called 2 to 1 multiplexer. For VHDL implementation please refer to “BinaryMUX/binaryMUX.vhd”.

### Simulation & Conclusion

For simulation waveform results please refer to “BinaryMUX/waveform.pdf”

For Test bench code please refer to “BinaryMUX/binaryMUX\_tb.vhd”.

From simulation results we can confirm that by applying “0xaaaaaaaa” to inputA and “0xbbbbbbbb” to inputB. By triggering “sel” we get value of inputA in output when sel equals 0 and inputB when sel equals 1.

## Program Counter

We decided that we won’t merge program counter (PC) in our instruction memory component. We found this way more convenient and easier to debug, since by separating the PC we were following the provided circuit diagram. Basically our PC on rising edge of the clock forwards the “input” value to “output”.

For VHDL implementation please refer to “Program\_Counter/Program\_counter.vhd”.

### Simulation & Conclusion

For simulation waveform results please refer to “Program\_Counter /waveform.pdf”

For Test bench code please refer to “Program\_Counter /Program\_Counter\_tb.vhd”.

After testing we can confirm that our implementation of PC works like expected. We successfully simulated counting up by value of 4 on every rising clock edge.

## Instruction Memory

For instruction memory we mostly based our code on the sample provided in the lecture notes. However, we did some small modification:

1. We made address as only a input from PC, in the sample code it was acting like a PC
2. Extended the DATA WIDTH to 32 bits, instead of 8
3. With simple bit manipulation we are accessing sequentially the array address by a step of 4. Instead of skipping 3 array elements on every iteration.

For VHDL implementation please refer to “InstrMem32bit/InstrMem32bit.vhd”.

### Simulation & Conclusion

For simulation waveform results please refer to “InstrMem32bit/waveform.pdf”

For Test bench code please refer to “InstrMem32bit/InstrMem32bit\_tb.vhd”.

We simulated a PC connected to the “address” and we were successful in reading sequentially the program memory from our component. Now we have all the necessary single components to implement fully functional “Instruction Fetcher”.

## Instruction Fetch

To implement fully functional instruction fetcher we decided to use descriptive VHDL instead of wiring up circuit diagram. For our structural VHDL implementation code please refer to “Instruction\_Fetch/Instruction\_Fetch.vhd”

### Simulation & Conclusion

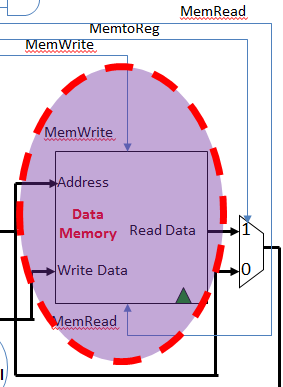
For simulation waveform results please refer to “Instruction\_Fetch /waveform.pdf”

For Test bench code please refer to “Instruction\_Fetch/Instruction\_Fetch\_tb.vhd”.

We can confirm that our final instruction fetcher implementation works exactly like expected, including branching and jumping. First we simulated 3 sequential steps for our PC. Afterwards we tested branching function, however we didn’t use bits (15:0) from our memory, since we don’t have sign extender yet, we just assigned a branching value to our “branch\_in” vector. With branching we made our PC to skip 3 instructions (125 ns), then when PC has reached memory location 11 (155 ns), we made it to execute jump instruction. In our memory location we have a value of 0x00000001, which means our jump instruction should set our PC to 4, and basically start executing the commands in sequence again on every rising clock edge.

# Testing the Data Memory

After we successfully implement the Instruction Fetch we implemented the Data Memory.



We wrote our VHDL code to represent the Data in, Address bus, the Output, the MemWrite and the MemRead. The memory is set to be a 256 array of 32 bits each array. Below is the simulation output of our test bench.

### Simulation Output

For waveform please refer to „Simple\_Ram/waveform.pdf”

We can see that in the first nanoseconds the simulation is still on reset mode. After 100ns we start testing our implementation.

1. Fist cycle MemWrite is set to 1, address is set to 0, and data in is 12345678h. Data out is stayed on zero since MemRead is set to 0.
2. Second cycle MemWrite is set to 0, address is still 0, data in is still 12345678h but since MemWrite is 0 they will not be written in memory. Data out now is changed to 12345678h since MemRead is set to 1.
3. Third cycle MemWrite is set to 1, address is set to 1, and data is 55555555h. Data out is still 12345678h since MemRead is set to 0.
4. Fourth cycle MemWrite is set to 0, address is set to 0, and data in is still 55555555h. However, Data out is still 12345678h since we now reading from address 0 which is 12345678h.
5. Fifth cycle MemWrite is set to 0 again, address is set to 1, and data In is still 55555555h. Now, data Out is changed to 55555555h since address 1 was set to that number 2 cycles ago and MemRead was set to 1.
6. Sixth cycle MemWrite is set to 1, address is set to 0, and data In to 33333333h. Data Out is still 55555555h since MemRead is set to 0.
7. Seventh cycle MemWrite is set to 0, address is set to 0m and data in is still 33333333h. Data out is now 33333333h since MemRead is set to 1.

### Conclusion

From our test bench going through each step we can confirm that out Data Memory is behaving as it should. We can put data in and choose write or read accordingly and get from data out the data that we want.

# To Sum Up

We managed to implement each of the components successfully without any setbacks. In the process we managed learn some important skills for descriptive VHDL, which will definitely be useful later in our course.