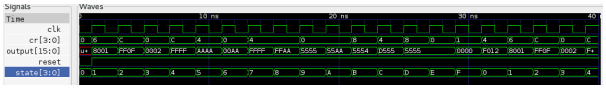
Zena's CS232 Project 7

For this week's [very long] project, we built a CPU using the provided CPU design template. The CPU imported data from a ROM, a RAM, and an ALU.
Tasks 1-3 were to make a project in quartus, then download the provided MIF files for the ROM and RAM. This was done in the same manner as in the last project.

Task 4 defined the signals that we had to use in our top level file, which I called "Project7", in accordance with my "Project#" theme. We wanted a clock, a reset button, and all the register outputs to be included so we could see them in the gtkwaves to come. We also needed to make port map statements and such for the ROM and RAM.

Task 5 was to make an ALU using the provided template. The ALU needed 4 signals: the first and second operands, the operation-specifying input, the condition register output, and the actual output value/result of the operation. We needed to define what to do for each operation, and since one condition was asking if there was a carried one, we needed to use 17 bits to get the answer, then map the low 16 bits to the actual answer that would be the output. We pretended like we used 16 the whole time; the sole purpose of the 17th bit was to see if there was a carried one for the condition checks at the bottom of the file. The condition register has 4 bits, and bit index 0 would be 1 if all zeroes in the result was true, for example. The gtkwave for the ALU is shown below:



The next task was to make the skeleton of the CPU, which has 9 states: startup, execute setup, execute process, execute wait, execute write, halt, and return pause 1 and 2. Each would enter the next state in order, except halt, which went nowhere because it was meant to be a "quit" function. Steps 7 and 8 were to set up all of the signals for the CPU as well as the port maps for the ALU, ROM, and RAM. We would need things like the two ALU sources, a program counter, and registers, as well as the overall output called oport, which is what our final result of each instruction is, when sent to the outport.

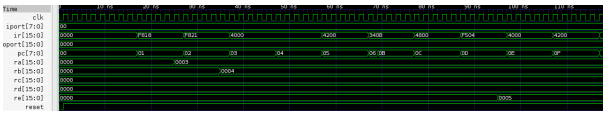
Task 9 was to make the process for the CPU. The result of pressing the reset button, the function of the states, etc. were defined here. The execute setup state was also set up, which was...fun. We needed to define the sources for all of the instructions as corresponding to the table. For instance, load from ram takes IR(15 downto 12), and within that, IR(11) tells us whether to add register E or not, and IR(7 downto 0) tells us the address value. This was a long process of following what the table told us. The table is as follows:

Instruction	opcode	Remaining bits
Load from RAM	0000	R-DDD-AAAAAAAA: DDD = dest (table B), R = add register E to immediate value, AAAAAAAA - immediate address value
Store to RAM	0001	R-SSS-AAAAAAAA: SSS = src (table B), R = add register E to immediate value, AAAAAAAA - immediate address value
Unconditional Branch	0010	UUUU-AAAAAAAA: U = unused, AAAAAAAA = immediate address value.

Conditional Branch	0011	00-CC-AAAAAAAA: CC = condition (00 = zero, 01 = overflow, 10 = negative, 11 = carry), U = unused, AAAAAAAA = immediate address value.
Call	0011	01-UU-AAAAAAAA: the cpu should push the PC and CR on the stack and then jump to the immediate address value AAAAAAAA.
Return	0011	10-UU-UUUUUUUU: the cpu should pop the CR and then pop the PC to continue execution at the stored address.
Exit	0011	11-UU-UUUUUUUU: The CPU should enter a halt state and not leave until the user resets the circuit.
Push	0100	SSS-UUUUUUUUU: SSS = src (table C) U = unused, operation puts the value into memory at location SP and increments SP.
Pop	0101	SSS-UUUUUUUUU: SSS = dest (table C) U = unused, operation reads the value from memory at location SP-1 and decrements SP.
Store to Output	0110	SSS-UUUUUUUUU: SSS = dest (table D) U = unused.
Load from Input	0111	SSS-UUUUUUUUU: SSS = dest (table B) U = unused.
Add	1000	SSS-TTT-UUU-DDD: SSS = srcA (table E), DDD = dest (table B), TTT = srcB (table E)
Subtract	1001	SSS-TTT-UUU-DDD: SSS = srcA (table E), DDD = dest (table B), TTT = srcB (table E)
And	1010	SSS-TTT-UUU-DDD: SSS = srcA (table E), DDD = dest (table B), TTT = srcB (table E)
Or	1011	SSS-TTT-UUU-DDD: SSS = srcA (table E), DDD = dest (table B), TTT = srcB (table E)
Exclusive-or	1100	SSS-TTT-UUU-DDD: SSS = srcA (table E), DDD = dest (table B), TTT = srcB (table E)
Shift	1101	R-SSS-UUUUU-DDD: SSS = srcA (table E), DDD = dest (table B), R = direction bit '0' = left, '1' = right, maintains sign bit
Rotate	1110	R-SSS-UUUUU-DDD: SSS = srcA (table E), DDD = dest (table B), R = direction bit '0' = left, '1' = right
Move	1111	T-(IIIIIIII or SSS-UUUUU)-DDD: DDD = dest (table B), T = if '1', treat next 8 bits as a sign-extended immediate value, else SSS = source location (table D)

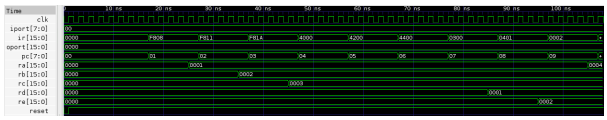
Execute write was set up here, as well. It was also a long process of following the table to decide on destinations, concatenating zeroes and taking slices when necessary. The execute return pause states are needed for the waiting time on return instructions. The execute wait state is meant to give wait time, as well, and does nothing. The execute process state tells the CPU when it can enable writing, which is when the instructions are store, push, or call.

Here is the result of the first cpu test bench:

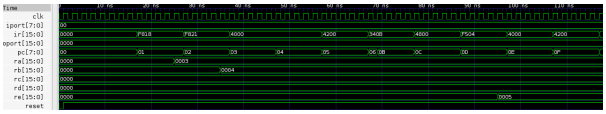


Task 10 was to test two provided MIF files, used as ROMs.

The first, test push, is pictured below:



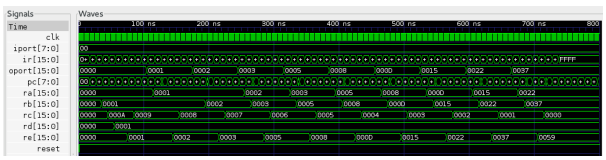
The second, test call, is pictured here:



The last task was to make the fibonacci sequence as a gtkwave simulation. This meant making an MIF file to give instructions, which was fairly straightforward (except the part where I spent literally 9 hours trying to find my mistake, which turned out to be a colon instead of a semicolon in my MIF...). My logic for this is shown below:

```
-- program memory file for fibonacci ROM
DEPTH = 256;
WIDTH = 16;
ADDRESS_RADIX = HEX;
DATA_RADIX = BIN;
CONTENT
BEGIN

00 : 1111100000000000; --MOVE 0 to RA
01 : 1111100000001001; --MOVE 1 to RB
02 : 1111100001010010; --MOVE 10 to RC
03 : 1111100000001011; --MOVE 1 to RD
04 : 0010000000000101; --Branch to 05

05 : 1000000001000100; --RE = RA + RB
06 : 1001010011000010; --RC = RC - RD
07 : 0011000000001100; --Branch to 0C(12) if RC = 0
08 : 0110100000000000; --Store RE to output
09 : 1111000100000000; --MOVE RB to RA
0A : 1111010010000001; --MOVE RE to RB
0B : 0010000000000101; --Branch to 05

0C : 0011110000000000; --Exit

[0D..FF] : 1111111111111111;
END
```

As you can see, I use RA and RB as the indicators for what I want to add. RC is the counter that has RD (which is always 1) subtracted from it each round in the loop, meaning it goes from 10 to 0, and when it's 0, the loop stops, which is how we get the first ten numbers only. The loop is to have the sum of RA and RB get put into RE, then output unless RC reaches 0. Then, RB is passed on to RA and RE is passed on to RB, which, when looped, effectively creates the sequence "0, 1, 1, 2, 3, 5, 8, 13, 21, 34", as seen in the oport row below:



No extensions this week because it's high time I started Project 8. Sorry!

This project taught me how the ALU, ROM, and RAM are implemented into the CPU, and it gave me practice interpreting and making instructions. It also made me comment my code more carefully as well as be more open to asking others for help.

Thanks to: Melody, Professor Maxwell, Tatsuya, Bumblebee, and anyone else who may have helped me through this long, difficult journey.