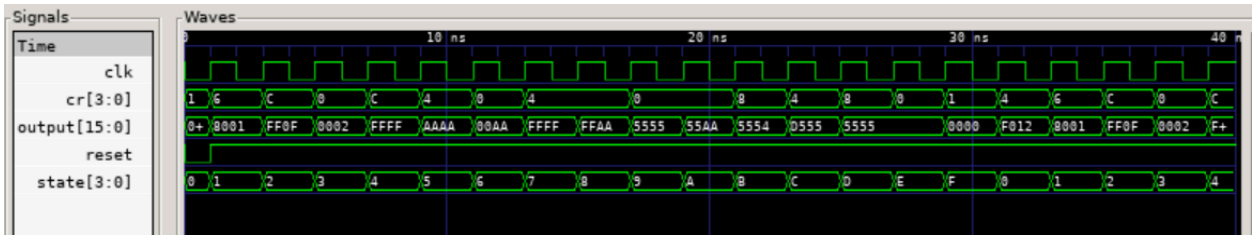
CS232 Project 7: CPU

November 4th, 2024

The purpose of this project is to quite literally, simulate the processes of a CPU.

Task 1: ALU

An essential component of the CPU.



Task 2: CPU

Top Level Design:

The top-level design centers on a state machine controlling the RISC CPU, which coordinates data flow among key components like the ALU, ROM, and RAM.

CPU State Machine

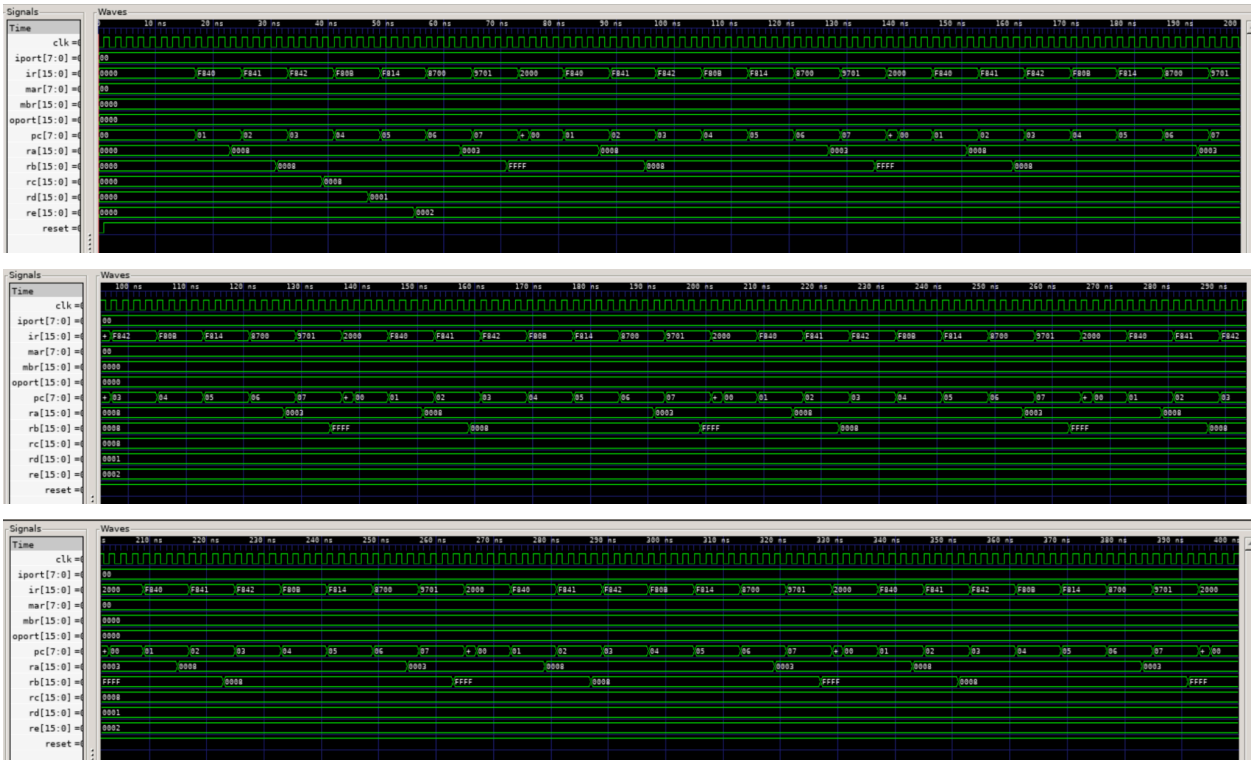
The state machine operates in nine distinct states: **Start, Fetch, Execute-Setup, Execute-ALU, Execute-MemoryWait, Execute-Write, Execute-ReturnPause1, Execute-ReturnPause2, and Halt**. Each state performs a specific task essential for instruction execution:

- **Start**: The CPU begins here, waiting for memory initialization and then transitions to the **Fetch** state.
- **Fetch**: The CPU retrieves an instruction from ROM, increments the Program Counter (PC), and proceeds to **Execute-Setup**.
- **Execute-Setup**: Prepares the CPU based on the instruction type:
    - **Load/Store/Push/Pop/Call/Return**: Sets up memory addresses, MBR, MAR, and adjusts the stack pointer as needed.
    - **ALU Operations**: Prepares source operands for the ALU.
    - **Branching**: Adjusts the PC if necessary.
    - **Exit**: Leads to the Halt state.
- **Execute-ALU**: Operates on inputs using the ALU for arithmetic/logical instructions or moves values directly. If memory access is needed, it goes to **Execute-MemoryWait**; otherwise, it advances to **Execute-Write**.
- **Execute-MemoryWait**: Pauses to allow memory data to be read or written, then moves to **Execute-Write**.
- **Execute-Write**: Finalizes execution by writing data to registers, memory, or the output port. Calls and returns are managed here as well, with **Return** going through **Execute-ReturnPause1** and **Execute-ReturnPause2** before returning to Fetch.

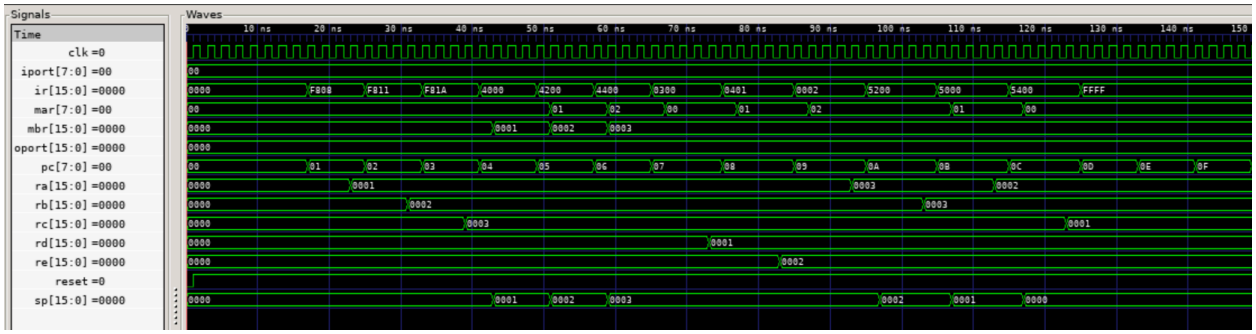
Component Collaboration

- **ALU**: Receives two operands and an opcode to perform arithmetic or logical operations. Its result goes to the destination register or the output port. The ALU also updates the 4-bit Condition Register, but only when required by specific instructions.
- **ROM**: Stores the program instructions, which the CPU fetches in the Fetch state.
- **RAM**: Acts as the main data memory for the CPU, storing intermediate values and the stack data (for calls, pushes, and pops).
- **Registers**: Registers A-E hold operands for operations, while the **Stack Pointer** manages stack operations, and the **Program Counter** (PC) keeps track of the instruction sequence. The **Condition Register** holds flags (e.g., zero, negative) set by the ALU to control conditional branching.

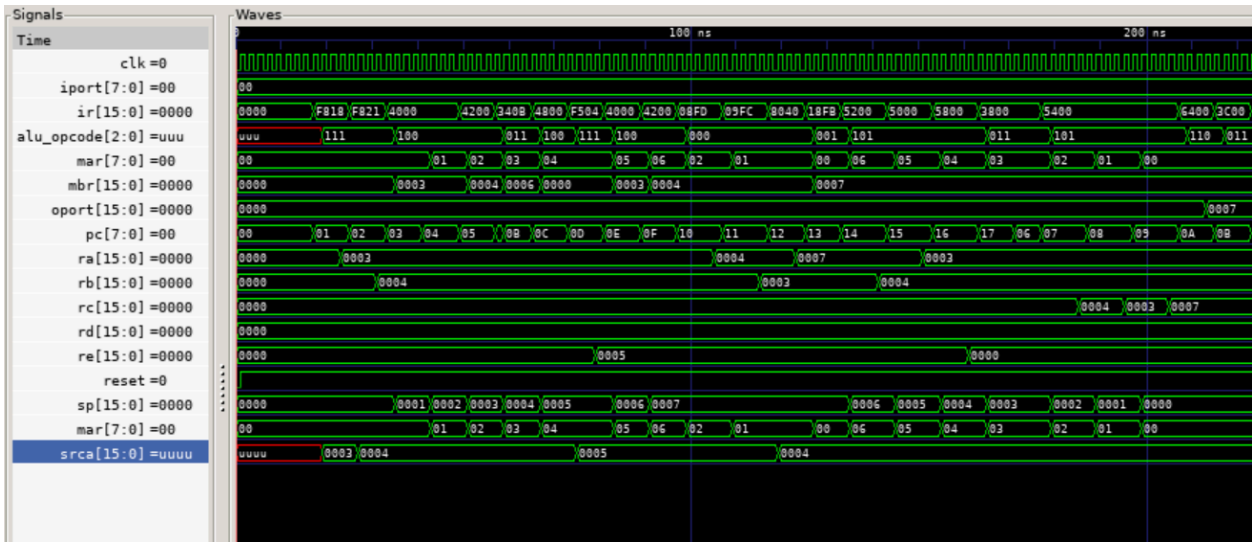
CPU Test:



Push Test:



Call Test:



Fibonacci Code: (fibonacci.mif)

-- program memory file for the fibonnaci sequence

DEPTH = 256;

WIDTH = 16;

ADDRESS_RADIX = HEX;

DATA_RADIX = BIN;

CONTENT

BEGIN

00 : 1111100000000000; -- Move 0 to RA (first Fibonacci number)

01 : 1111100000001001; -- Move 1 to RB (second Fibonacci number)

02 : 1111100000100011; -- Move 4 to RD (10 Fibonacci Values = 4 Loops + 2 numbers)

03 : 1111100000001100; -- Move 1 to RE (Loop Decrementer)

04 : 0110000000000000; -- Store RA (result) to output (1st Fibonacci)

05 : 0110001000000000; -- Store RB (result) to output (2nd Fibonacci)

-- Loop start

06 : 0011000000001111; -- Branch to 15 if RD (multiplier) is zero to end

07 : 1000000001000010; -- Add RA and RB and put the output in RC

08 : 0110010000000000; -- Store RC (result) to output (Display)

09 : 1111001000000000; -- Move RC to RA (next Fibonacci number)

0A : 1000000001000010; -- Add RA and RB and put the output in RC

0B : 0110010000000000; -- Store RC (result) to output (Display)

0C : 1111001000000001; -- Move RC to RB (next Fibonacci number)

0D : 1001011100000011; -- SUB 1 (RE) from RD (decrease counter)

0E : 0010000000000110; -- Jump back to the beginning of the loop

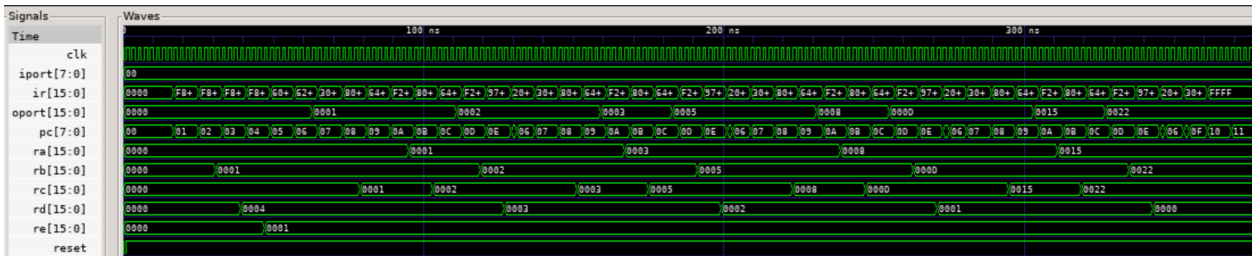
-- End

[0F..FF] : 1111111111111111; -- Nothing lol

END

Fibonacci Test:

The first 10 numbers of the Fibonacci sequence are first stored temporarily through register C, then displayed via the output as they come up. The program stores the first 2 fibonacci numbers (0, 1) and stores them in registers RA and RB respectively. The program also uses RD and RE as a loop counter and loop decrementer respectively. Because the program already directly outputs RA and RB at the beginning, and for each loop RC updates RA and RB one at a time, (so the program generates 2 fibonacci numbers per loop) the loop runs 4 times. Note: It looks like there are only 9 numbers, but that's because 1 appears twice in the sequence, and GTKWave doesn't acknowledge multiple outputs.



Extensions

Extension 1: Unsigned Multiply Program (multi.mif)

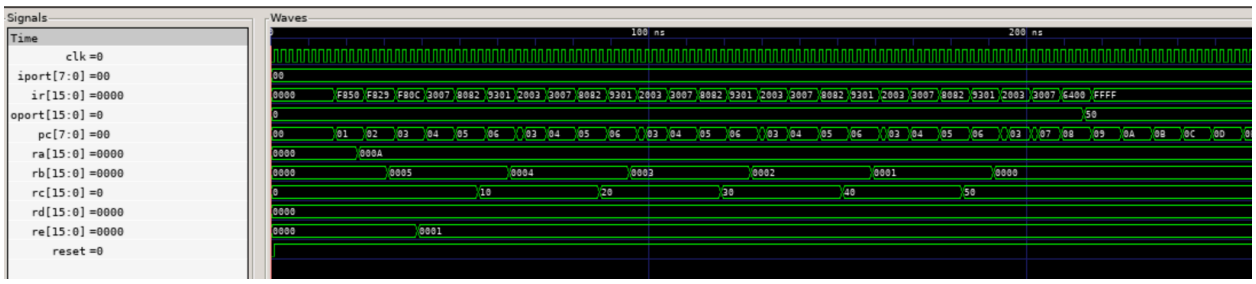
To get used to writing programs on the CPU, I will write more programs! This first program multiplies two numbers (10 and 5) and stores the result, which requires repeated addition and a loop. This involves repeatedly

adding the former number multiple times to a temporary register while using the second number as a tally of how many times the former number has been added.

Program:

```
DEPTH = 256;
WIDTH = 16;
ADDRESS_RADIX = HEX;
DATA_RADIX = BIN;
CONTENT
BEGIN
-- Load two numbers to multiply
00 : 1111100001010000; -- move 10 to RA (Multiplicand)
01 : 1111100000101001; -- move 5 to RB (Multiplier)
02 : 1111100000001100; -- move 1 to RE (Decrementer)

-- Loop to perform multiplication by repeated addition
03 : 0011000000000111; -- Branch to 07 if RB (multiplier) is zero to end
04 : 1000000010000010; -- ADD RA (multiplicand) to RC (result)
05 : 1001001100000001; -- SUB 1 (RE) from RB (decrement multiplier)
06 : 0010000000000011; -- Unconditional branch to line 03 (repeat)

-- End of multiplication
07 : 0110010000000000; -- Store RC (result) to output

[08..FF] : 1111111111111111; -- Halt
END
```

GTKWave:



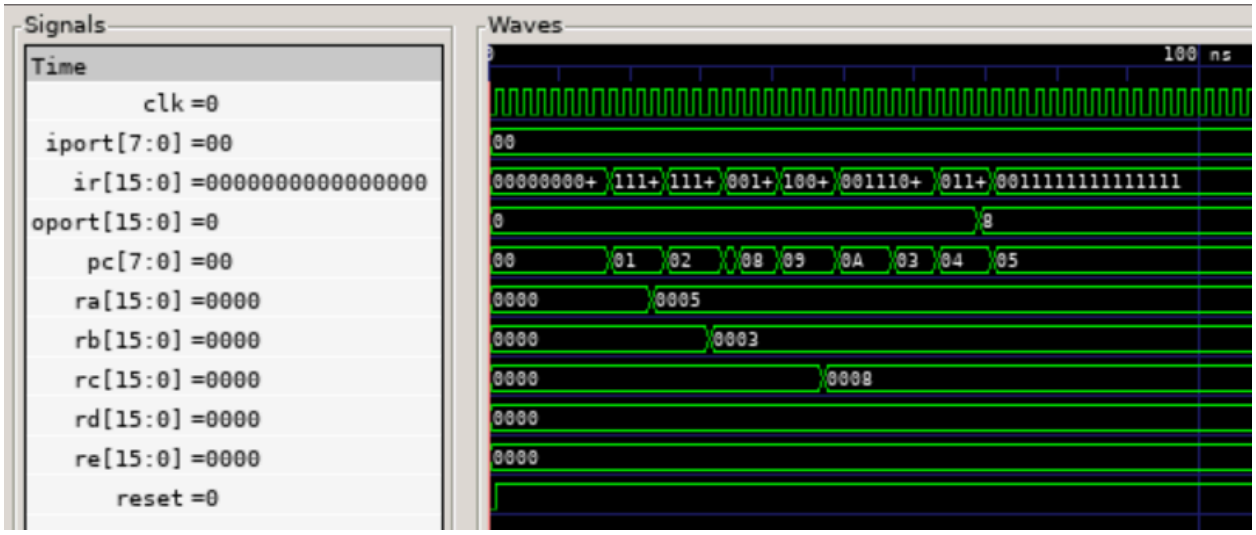
Extension 2: Call/Return Program (callreturn.mif)

This program demonstrates a function that adds two numbers using CALL and RETURN. In this case, add 5 and 3 and display 8.

Program:

```
DEPTH = 256;
WIDTH = 16;
ADDRESS_RADIX = HEX;
DATA_RADIX = BIN;
CONTENT
BEGIN
-- Main Program
00 : 1111100000101000; -- move 5 to RA (First operand)
01 : 1111100000011001; -- move 3 to RB (Second operand)
02 : 0011010000001000; -- CALL to address 08 (subroutine)

-- Output Result
03 : 0110010000000000; -- Store RC (result of addition) to output
04 : 0011111111111111; -- Exit/Halt

-- Subroutine to Add RA and RB, store in RC
08 : 1000000001000010; -- ADD RA and RB, store in RC
09 : 0011101000000000; -- RETURN to main program

[0A..FF] : 1111111111111111; -- Halt
END
```

GTKWave:

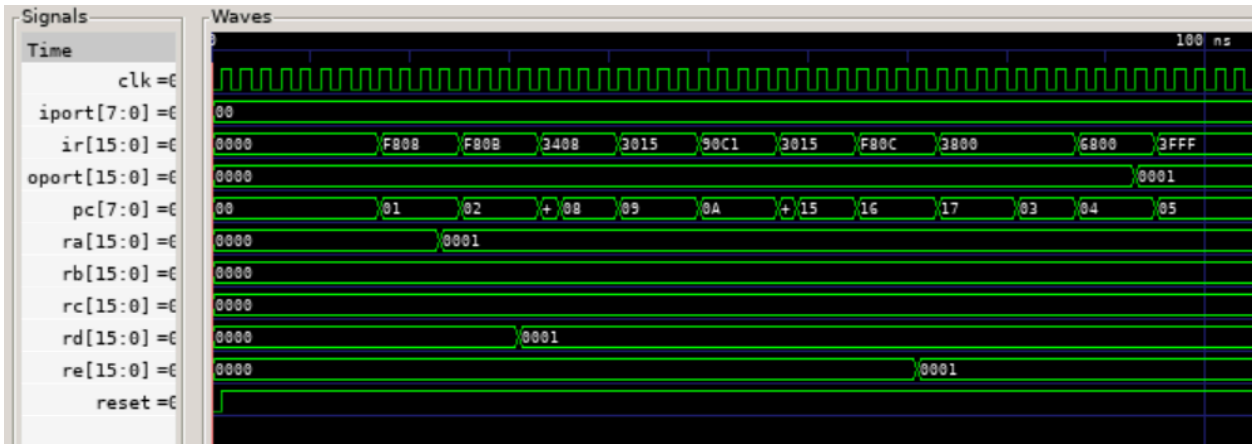


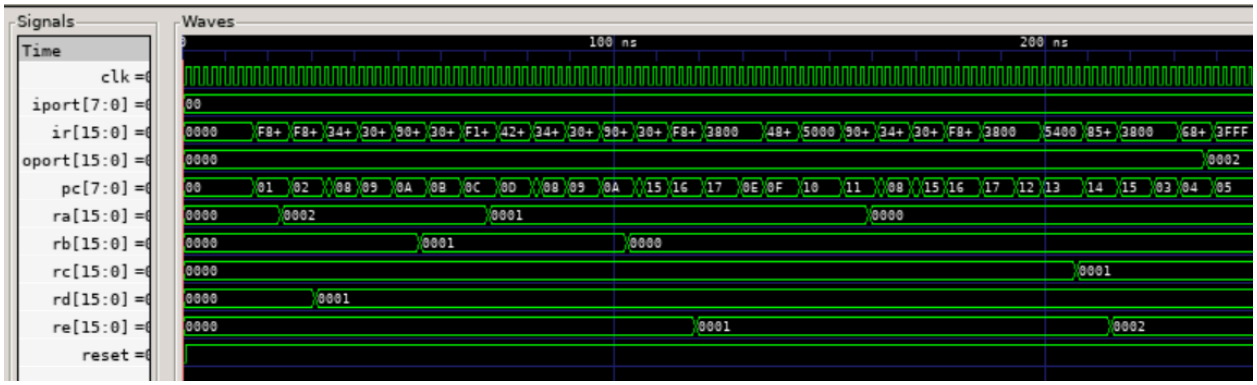
Extension 3: Fibonacci Recursion (fibonaccirec.mif)

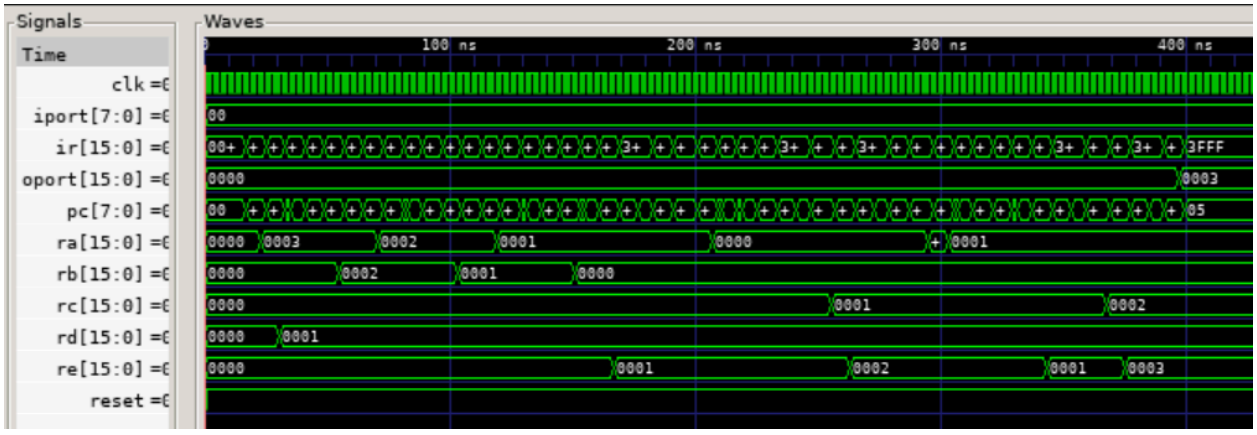
This program recursively calls the fibonacci sequence by using this overly complicated code. The first instruction determines what index of the fibonacci sequence to output.

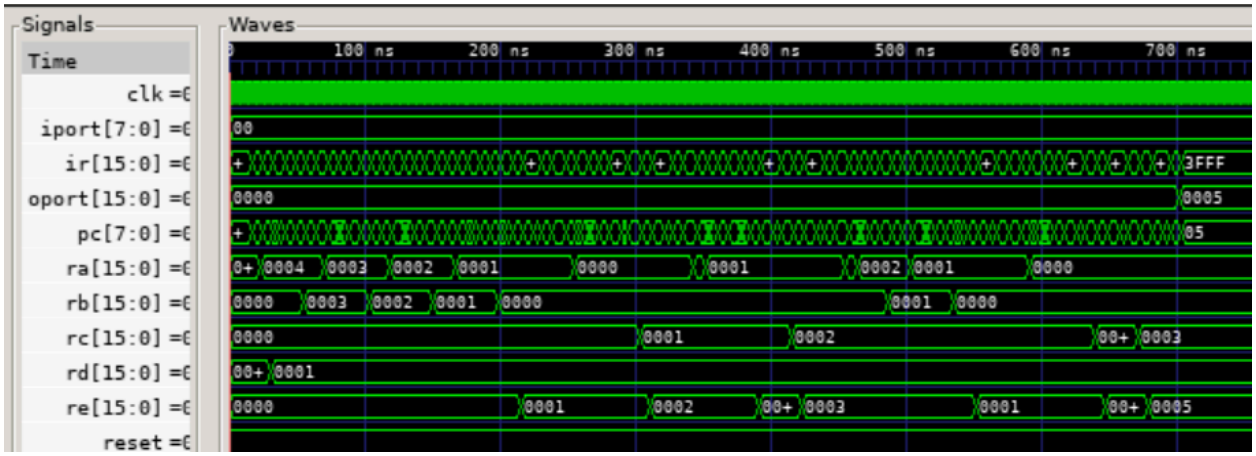
```
DEPTH = 256;
WIDTH = 16;
ADDRESS_RADIX = HEX;
DATA_RADIX = BIN;
CONTENT
BEGIN
-- Main Program: Calculate Fibonacci(5)
00 : 1111100000100000; -- move 3 to RA (Fibonacci index)
01 : 1111100000001011; -- move 1 to RD (Decrementer)
02 : 0011010000001000; -- CALL to address 08 (Fibonacci subroutine)

-- end
03 : 0110100000000000; -- Store RE (result) to output (display Fibonacci(5))
04 : 0011111111111111; -- Exit/Halt

-- Fibonacci Subroutine
-- Check if n = 0 or n = 1
08 : 0011000000010101; -- Conditional branch to address 15 if RA == 0 (base case F(0) = 0)
09 : 1001000011000001; -- SUB 1 from RA, store result in RB (RB = RA - 1 (RD))
0A : 0011000000010101; -- Conditional branch to address 15 if RA == 0 (base case F(0) = 0)

-- Recursive Case
-- Save the original RA value (because it will get lost as you iterate further) for F(n-2) and calculate F(n-1) recursively
0B : 1111000100000000; -- Move RB (n-1) to RA (prepare to call F(n-1))
0C : 0100001000000000; -- Push RB (n-1) to stack (for N-2)
0D : 0011010000001000; -- CALL Fibonacci subroutine for F(n-1)
0E : 0100100000000000; -- Push RE (result of F(n-1)) to stack

-- Pop the other n value, and Find F(n-2)
0F : 0101000000000000; -- Pop n-1 to RA (for N-2)
10 : 1001000011000000; -- SUB 1 from RA, store result in RA (RB = RA - 1 (RD)) (prepare to call F(n-2))
11 : 0011010000001000; -- CALL Fibonacci subroutine for F(n-2)
12 : 0101010000000000; -- Pop F(n-1) result from stack into RC
13 : 1000010100000100; -- ADD RC (F(n-1)) to RE (F(n-2)), store result in RE
14 : 0011100000000000; -- RETURN

-- Base Cases
15 : 1111100000001100; -- Move 1 to RE (base case F(0) OR F(1) = 0)
16 : 0011100000000000; -- RETURN

[19..FF] : 1111111111111111; -- Halt
END
```

Fibonacci 1-3 (F(1) = 1, F(2) = 2, F(3) = 3, F(4) = 5): (NOTE: the sequence starts with 1 , 1 instead of 0, 1)


Signals
Waves
Time
clk
iport[7:0]
ir[15:0]
oport[15:0]
pc[7:0]
ra[15:0]
rb[15:0]
rc[15:0]
rd[15:0]
re[15:0]
reset
100 ns


Signals
Waves
Time
clk
iport[7:0]
ir[15:0]
oport[15:0]
pc[7:0]
ra[15:0]
rb[15:0]
rc[15:0]
rd[15:0]
re[15:0]
reset
100 ns
200 ns


Signals
Waves
Time
clk
iport[7:0]
ir[15:0]
oport[15:0]
pc[7:0]
ra[15:0]
rb[15:0]
rc[15:0]
rd[15:0]
re[15:0]
reset
100 ns
200 ns
300 ns
400 ns



Acknowledgements

This program was extremely long, and the problem with these types of long programs is that it's easy and frustrating to get lost in your code, especially when troubleshooting in VHDL. Thank you to Professor Li who basically had to look over the mess that was my base code and fix what I messed up, because I was literally scratching my head for ages. She also helped me figure out how to code the fibonacci sequence recursion for my last extension.