CS232 Project 8: Assembler

November 18th, 2024

The purpose of this project is to follow up on the previous; write a 2-pass assembler that converts a mnemonic assembly language into the machine code for a CPU

Task 1: Assembler

Top Level Design:

The assembler is implemented using a two-pass design to ensure that labels and instructions are processed efficiently and correctly:

Pass 1: Label Resolution

1. **Objective**: Identify and assign memory addresses to all labels in the assembly code.
2. **Implementation**:
    - Iterate through the tokenized lines of the input file.
    - Track the memory address of each instruction as you progress.
    - If a line contains a label (identified by a colon `:` at the end), store the label's name and its corresponding memory address in a dictionary called labels.
    - Ignore instruction details during this pass, focusing solely on labels and their positions.
3. **Result**: A labels dictionary mapping label names to their memory addresses.

Pass 2: Instruction Encoding

1. **Objective**: Convert each assembly instruction into binary machine code.
2. **Implementation**:
    - Iterate through the tokenized lines again, now using the resolved labels dictionary for branch instructions.
    - Use an opcode table to map instruction mnemonics to their corresponding binary representations.
    - For each instruction:
        - Extract the opcode and operands.
        - Encode immediate values, register numbers, or label addresses into binary.
        - Handle each instruction format (e.g., movei, add, bra) with specific logic to ensure proper binary encoding.
    - Append the fully encoded instruction to the output list.
3. **Result**: A list of binary machine instructions ready for use in the target CPU.

Fibonacci:

This is based on the code from the last project, to generate the first 10 numbers of the fibonacci sequence. As a repeat from that project, here's how my instructions worked:

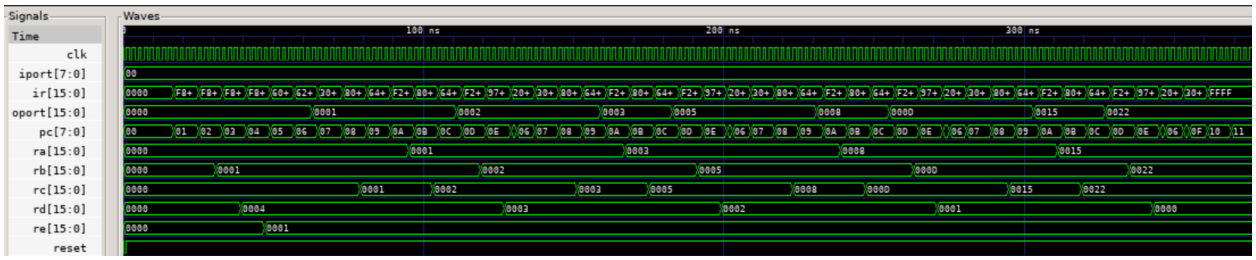
The first 10 numbers of the Fibonacci sequence are first stored temporarily through register C, then displayed via the output as they come up. The program stores the first 2 fibonacci numbers (0, 1) and stores them in registers RA and RB respectively. The program also uses RD and RE as a loop counter 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.

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 14 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
```

Here is the working GTKWave:



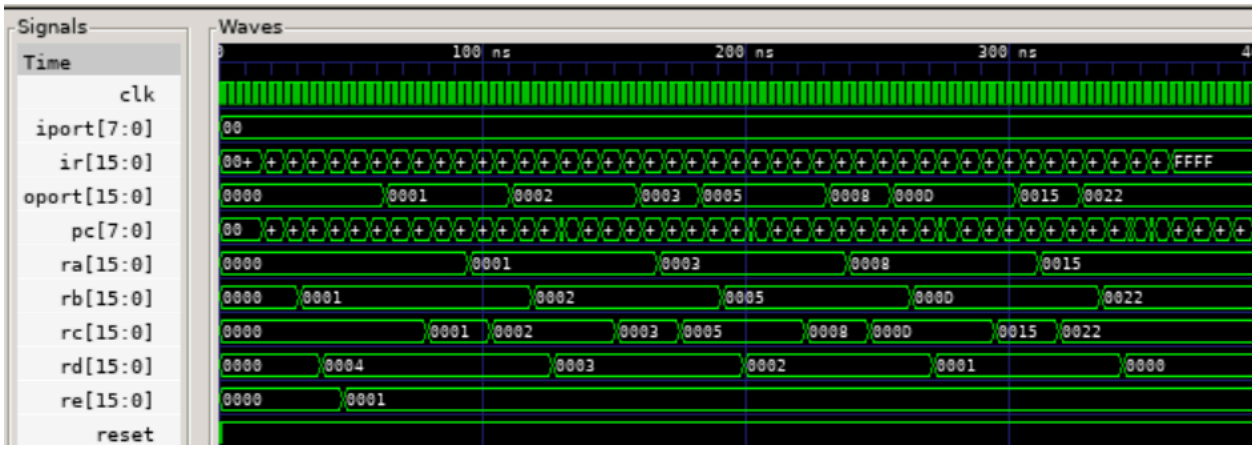
Fibonacci Assembly Code: (fibonacci.mif)

```
MOVEI 0 RA
MOVEI 1 RB
MOVEI 4 RD
MOVEI 1 RE
OPORT RA
OPORT RB
LOOP_START:
BRAZ END
ADD RA RB RC
OPORT RC
MOVE RC RA
ADD RA RB RC
OPORT RC
MOVE RC RB
SUB RD RE RD
BRA LOOP_START
END:
```

Fibonacci Instructions Based on Assembly Code: (machinefibonacci.mif)

```
-- Program Memory File
DEPTH = 256;
WIDTH = 16;
ADDRESS_RADIX = HEX;
DATA_RADIX = BIN;
CONTENT
BEGIN
00 : 1111100000000000;
01 : 1111100000001001;
02 : 1111100000100011;
03 : 1111100000001100;
04 : 0110000000000000;
05 : 0110001000000000;
06 : 0011000000001111;
07 : 1000000001000010;
08 : 0110010000000000;
09 : 1111001000000000;
0A : 1000000001000010;
0B : 0110010000000000;
0C : 1111001000000001;
0D : 1001011100000011;
0E : 0010000000000110;
[0F..FF] : 1111111111111111;
END
```

As shown, it matches the code aforementioned, but to show the accuracy, here is the GTKWave run on this file.



Recursion:

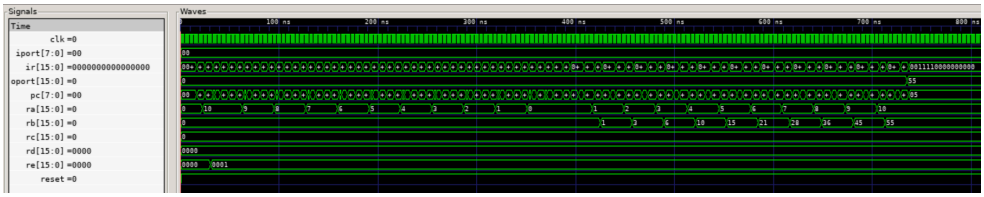
The purpose of this program is meant to sum the numbers 1 to N through recursion, and works as follows:

1. Initializes N with a value by storing it into Register A (for this example, 10).
2. Initializes a decrementer 1 in Register E.
3. Pushes N onto the stack as the argument for the recursive function.
4. Calls the recursive function SUM to calculate the result.
5. Pops the final result into a register and outputs it using the output port.
6. Ends the program with a HALT instruction.
7. **Recursive Function (SUM)**:
    a. Pushes the current value of N onto the stack to save it.
    b. Checks if N = 0 (base case) using the BRAZ instruction.
    c. If not, decrements N, and recursively calls SUM.
    d. After the recursive call, pops N into Register B and calculates N + SUM(N−1).
    e. Returns to either the next sum of numbers or the beginning of the program.
8. **Base Case**:
    a. If N = 0, pushes 0 onto Register B and returns.

Demonstration of Execution:

Given N=10, the recursive process computes: SUM(10) = 10 + SUM(9) = 10 + (9 + SUM(8)) = ··· = 10 + 9 + 8 + ··· + 1=55

Here is the working GTKWave:



Fibonacci Assembly Code: (recursion.a)

```
MOVEI 10 RA
MOVEI 1 RE
CALL SUM
OPORT RB
HALT
SUM:
PUSH RA
BRAZ BASECASE
SUB RA RE RA
CALL SUM
POP RA
ADD RA RB RB
RETURN
BASECASE:
POP RA
ADD RA RB RB
RETURN
```

Fibonacci Instructions Based on Assembly Code: (recursion.mif)

```
-- Program Memory File
DEPTH = 256;
WIDTH = 16;
ADDRESS_RADIX = HEX;
DATA_RADIX = BIN;
CONTENT
BEGIN
00 : 1111100001010000;
01 : 1111100000001100;
02 : 0011010000000101;
03 : 0110001000000000;
04 : 0011110000000000;
05 : 0100000000000000;
06 : 0011000000001100;
07 : 1001000100000000;
08 : 0011010000000101;
09 : 0101000000000000;
0A : 1000000001000001;
0B : 0011100000000000;
0C : 0101000000000000;
0D : 1000000001000001;
0E : 0011100000000000;
[0F..FF] : 1111111111111111;
END
```

As shown, the GTKWave increments as more numbers from the stack reappear and are added together. The final output lines up as well. To account for higher dimensions, all that's needed is to change the first instruction from 10 to N. (Up to 255)

Extensions

Error Detection

The code is modified to detect invalid inputs before executing, utilizing the try and except python commands. It allows the program to give custom errors, in which case indicating which line has the error and the correct format to follow. It also detects invalid Register symbols.

Testing the CPU's capabilities (within all.a and all.mif)

This code tests all of the assembler and thus CPU commands to make one large operation.

Assembly Code: (all.a)

```
START:
MOVEI 5 RA
MOVEI 3 RB
ADD RA RB RC
SUB RA RB RD
SHIFTL RA RC
SHIFTR RB RD
XOR RA RB RE
AND RA RB RE
OR RA RB RE
OPORT RC
STORE RC 10
LOAD RD 10
OPORT RD
MOVEI 0 RA
BRAZ BRANCH_SKIP
MOVEI 99 RE
OPORT RE
BRANCH_SKIP:
MOVEI 42 RE
OPORT RE
PUSH RA
PUSH RE
POP RC
OPORT RC
POP RD
OPORT RD
CALL SUBROUTINE
OPORT RA
IPORT RB
OPORT RB
HALT
SUBROUTINE:
MOVEI 99 RA
RETURN
```

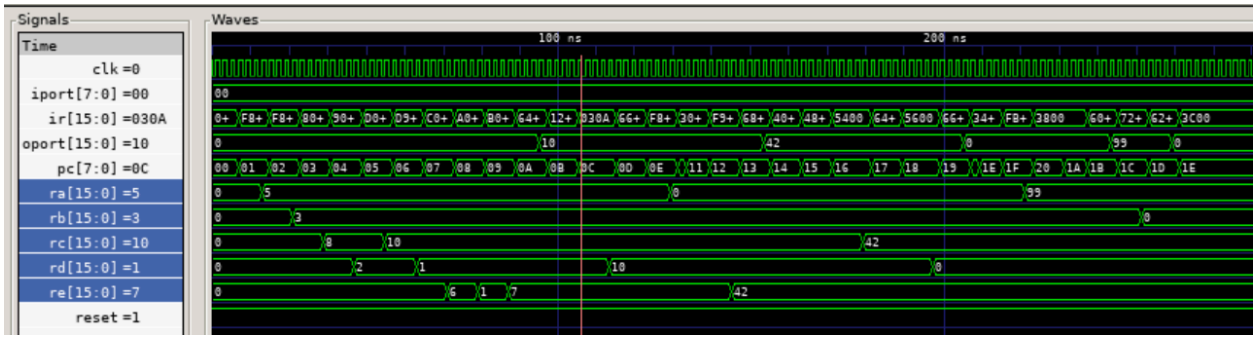
Instructions Based on Assembly Code: (all.mif)

```
-- Program Memory File
DEPTH = 256;
WIDTH = 16;
ADDRESS_RADIX = HEX;
DATA_RADIX = BIN;
CONTENT
BEGIN
00 : 1111100000101000;
01 : 1111100000011001;
02 : 1000000001000010;
03 : 1001000001000011;
04 : 1101000000000010;
05 : 1101100100000011;
06 : 1100000001000100;
07 : 1010000001000100;
08 : 1011000001000100;
09 : 0110010000000000;
0A : 0001001000001010;
0B : 0000001100001010;
0C : 0110011000000000;
0D : 1111100000000000;
0E : 0011000000010001;
0F : 1111101100011100;
10 : 0110100000000000;
11 : 1111100101010100;
12 : 0110100000000000;
13 : 0100000000000000;
14 : 0100100000000000;
15 : 0101010000000000;
16 : 0110010000000000;
17 : 0101011000000000;
18 : 0110011000000000;
19 : 0011010000011110;
1A : 0110000000000000;
1B : 0111001000000000;
1C : 0110001000000000;
1D : 0011110000000000;
1E : 1111101100011000;
1F : 0011100000000000;
[20..FF] : 1111111111111111;
END
```

If everything is correct, we should get the outputs as:

{10, 10, 42, 42, 0, 99, 0}

GTKWave:



As we can see, it follows just that! (10 and 42 overlap because GTKWave is weird like that, but it does print twice)

Acknowledgements

For once, I actually didn't get much help for this project. Thanks to the TAs and Professor Li for helping on the way anyways!