Lab Report #6

Creating the Single Cycle Processor

2220 01 – Microprocessor Design

Benjamin Philipose

06/07/2022

Processor Design

The Processor design is a basic design of a RISC-V Processor where I utilized components from previous labs which include Register Bank, Data Memory, and ALU module. Within this lab I designed the Control unit, PC counter, as well as branch, 2x1 mux, and immediate gen capabilities. The CPU after being fully assembled is able to support 16 separate instructions, two of which are LW and SW for data memory, and another 2 for branching (BEQ BNE).

The instruction RAM is essentially a linked list of 32 bits which contain a separate instruction in each 'node'. Due to this structure, each node is treated as a line that holds an instruction in assembly code. This allows for Branching capabilities to jump around in the instructions and to loop back to a certain instruction.

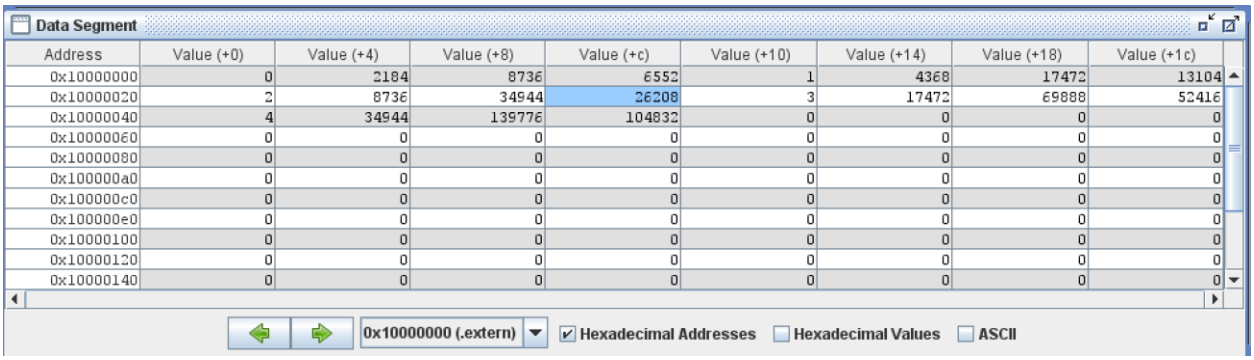

Figure 1 - RARS Expected Result

When implementing the Control logic, I made a rough truth table with values that I though conceptually made sense, from there I implemented it using a with-select-when statement where each output signal took 1 bit from the with-select-when's output. Note that the order for the output bit with the with-select-when is in the same order as the rough truth table. I also put X's for values that wouldn't mean anything and put 0's in place of the X's when including it in my implementation. Only exception is for funct7, opcode, and funt3 values, where if they had X's it was not checked for that particular instruction within the with-select-when statement.

```
identity <= funct3 & opcode;
--Instruction  Branch  MemRead  MemtoReg  ALUCtrl  MemWrite  ALUSrc  RegWrite  ImmGen  funct7    funct 3   opcode   Type
--add          00      1        0         00000    0         0       1         00      0000000   000       0110011  R-Type
--sub          00      1        0         00001    0         0       1         00      0100000   000       0110011  R-Type
--addi         00      1        0         00010    0         1       1         10      xxxxxxx   000       0010011  I-Type
--and          00      1        0         00011    0         0       1         00      0000000   111       0110011  R-Type
--or           00      1        0         00101    0         0       1         00      0000000   110       0110011  R-Type
--sll          00      1        0         00111    0         0       1         00      0000000   001       0110011  R-Type
--srl          00      1        0         01001    0         0       1         00      0000000   101       0110011  R-Type
--lw           00      0        1         00010    0         1       1         10      xxxxxxx   010       0000011  S-Type
--sw           00      1        0         00010    1         1       0         11      xxxxxxx   010       0100011  S-Type
--beq          10      1        0         00001    0         0       0         01      xxxxxxx   000       1100011  B-Type
--bne          01      1        0         00001    0         0       0         01      xxxxxxx   001       1100011  B-Type
--lui          00      1        0         xxxxx    0         1       1         00      xxxxxxx   xxx       0110111  U-Type
--andi         00      1        0         00100    0         1       1         10      xxxxxxx   000       0010011  I-Type
--ori          00      1        0         00110    0         1       1         10      xxxxxxx   110       0010011  I-Type
--slli         00      1        0         01000    0         1       1         10      0000000   001       0010011  I-Type
--srli         00      1        0         01010    0         1       1         10      0000000   101       0010011  I-Type
```

Figure 2 - Rough Truth Table for Control Logic

```
resultbit <= ("00"&"1"&"0"&"00000"&"0"&"0"&"1"&"00") WHEN (funct7 = "0000000" AND identity = "0000110011") ELSE
             ("00"&"1"&"0"&"00001"&"0"&"0"&"1"&"00") WHEN (funct7 = "0100000" AND identIty = "0000110011") ELSE
             ("00"&"1"&"0"&"00010"&"0"&"1"&"1"&"10") WHEN (identity = "0000010011") ELSE
             ("00"&"1"&"0"&"00011"&"0"&"0"&"1"&"00") WHEN (funct7 = "0000000" AND identity = "1110110011") ELSE
             ("00"&"1"&"0"&"00101"&"0"&"0"&"1"&"00") WHEN (funct7 = "0000000" AND identity = "1100110011") ELSE
             ("00"&"1"&"0"&"00111"&"0"&"0"&"1"&"00") WHEN (funct7 = "0000000" AND identity = "0010110011") ELSE
             ("00"&"1"&"0"&"01001"&"0"&"0"&"1"&"00") WHEN (funct7 = "0000000" AND identity = "1010110011") ELSE
             ("00"&"0"&"1"&"00010"&"0"&"1"&"1"&"10") WHEN (identity = "0100000011") ELSE
             ("00"&"1"&"0"&"00010"&"1"&"1"&"0"&"11") WHEN (identity = "0100100011") ELSE
             ("10"&"1"&"0"&"00001"&"0"&"0"&"0"&"01") WHEN (identity = "0001100011") ELSE
             ("01"&"1"&"0"&"00001"&"0"&"0"&"0"&"01") WHEN (identity = "0011100011") ELSE
             ("00"&"1"&"0"&"00000"&"0"&"1"&"1"&"00") WHEN (identity(6 DOWNTO 0) = "0110111") ELSE
             ("00"&"1"&"0"&"00100"&"0"&"1"&"1"&"10") WHEN (identity = "0000010011") ELSE
             ("00"&"1"&"0"&"00110"&"0"&"1"&"1"&"10") WHEN (identity = "1100010011") ELSE
             ("00"&"1"&"0"&"01000"&"0"&"1"&"1"&"10") WHEN (funct7 = "0000000" AND identity = "0010010011") ELSE
             ("00"&"1"&"0"&"01010"&"0"&"1"&"1"&"10") WHEN (funct7 = "0000000" AND identity = "1010010011") ELSE
             "00100000000000";

Branch <= resultbit(13 DOWNTO 12);
MemRead <= resultbit(11);
MemtoReg <= resultbit(10);
ALUCtrl <= resultbit(9 DOWNTO 5);
MemWrite <= resultbit(4);
ALUSrc <= resultbit(3);
RegWrite <= resultbit(2);
ImmGen <= resultbit(1 DOWNTO 0);
```

Figure 3 - Finalized Control Logic

When implementing the Immediate generate logic I had a few issues there with regards to how many bits each instruction selects and from where. Majority of the time each instruction type had their own immediate generate which conveyed which orientation and selection of bits from the instruction would be utilized for the immediate bits. “00” is for U/R-type instructions, R-type doesn’t use immediate bits so any immediate generate number would work. “10” is for I-Type and Load Word instructions, “11” is used for Store word instruction, and lastly, “01” is used for B-type instructions which are the two branch instructions required for implementation. Figure 3’s table does have a good representation of

each instruction immediate generate statement. For selecting the immediate bits in Processor.vhdl there were 4 standards of siphoning the required immediate bits as show in Figure 5.

For sign extending the immediate values siphoned from instruction, sign extend was based on the 31st bit of the instruction due to it being the most significant bit from the extracted immediate generated bit for all the incorporated instruction types.

```
--For immidiate select
with instruction1(31) select
signExtend <= (others => '1') when '1',
(others => '0') when others;

with immGen select
immidiatepart <= (signExtend(19 downto 0) & instruction1(31 downto 20)) when "10", --I-type
(signExtend(18 downto 0) & instruction1(31) & instruction1(7) & instruction1(30 downto 25) & instruction1(11 downto 8) & '0') when "01", --B-type
(signExtend(19 downto 0) & instruction1(31 downto 25) & instruction1(11 downto 7)) when "11", --S-type
(instruction1(31 downto 12) & "000000000000") when others; --U-type/R-type

--Branching Logic
```

Figure 4 - ImmGen Logic

Processor Testing

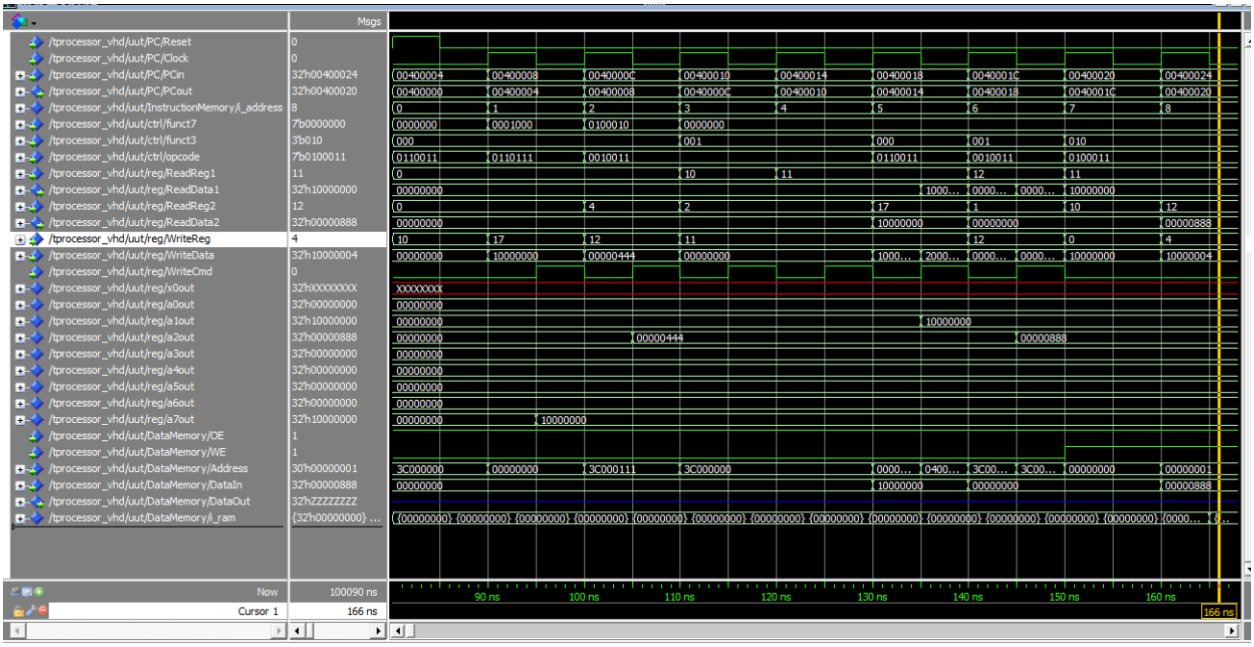

Figure 5 – Processor at 1st Checkpoint(1st iteration)

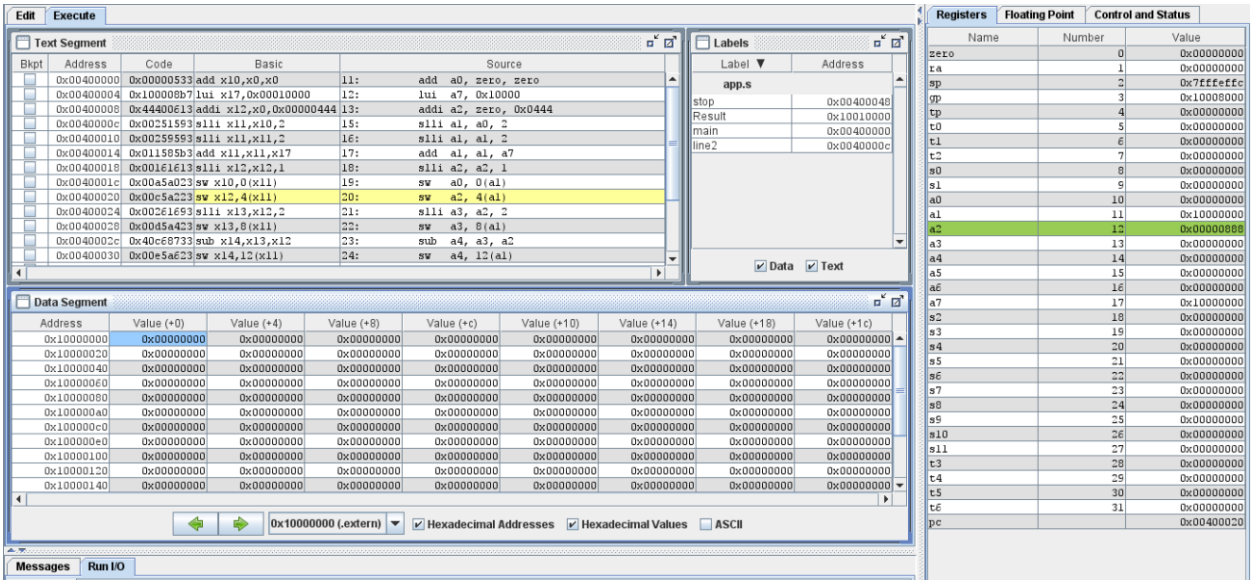

Figure 6 - RARS Results at first checkpoint

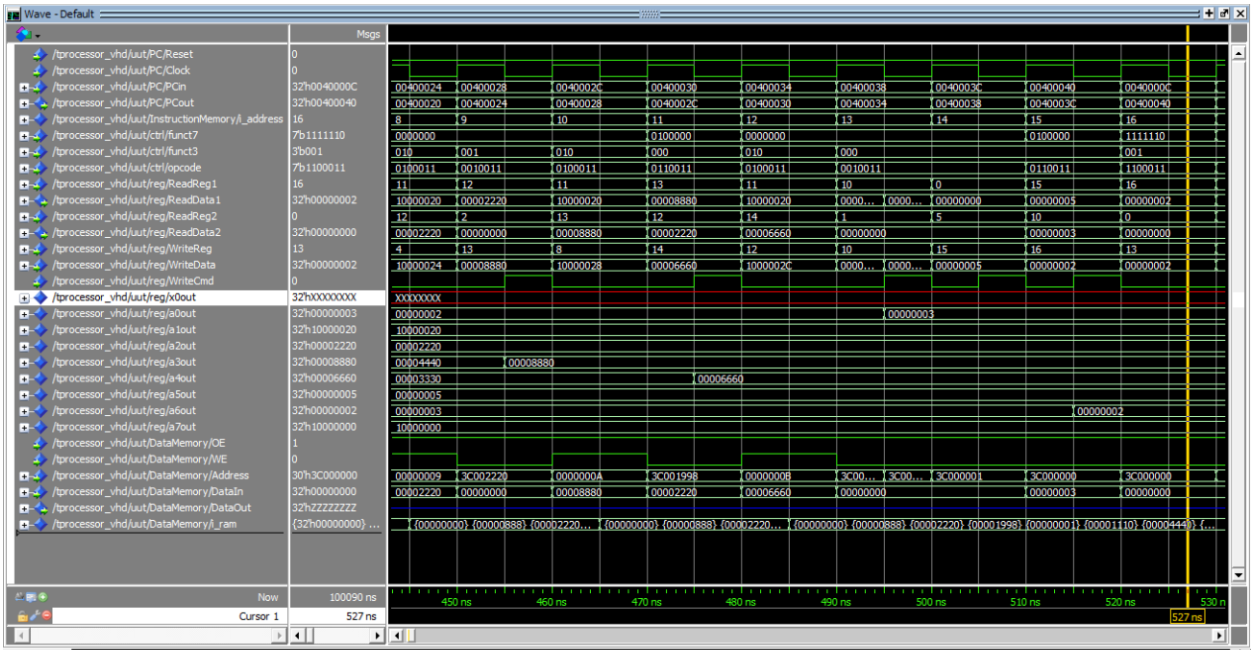

Figure 7 - Checkpoint 2 (end of 3rd iteration)

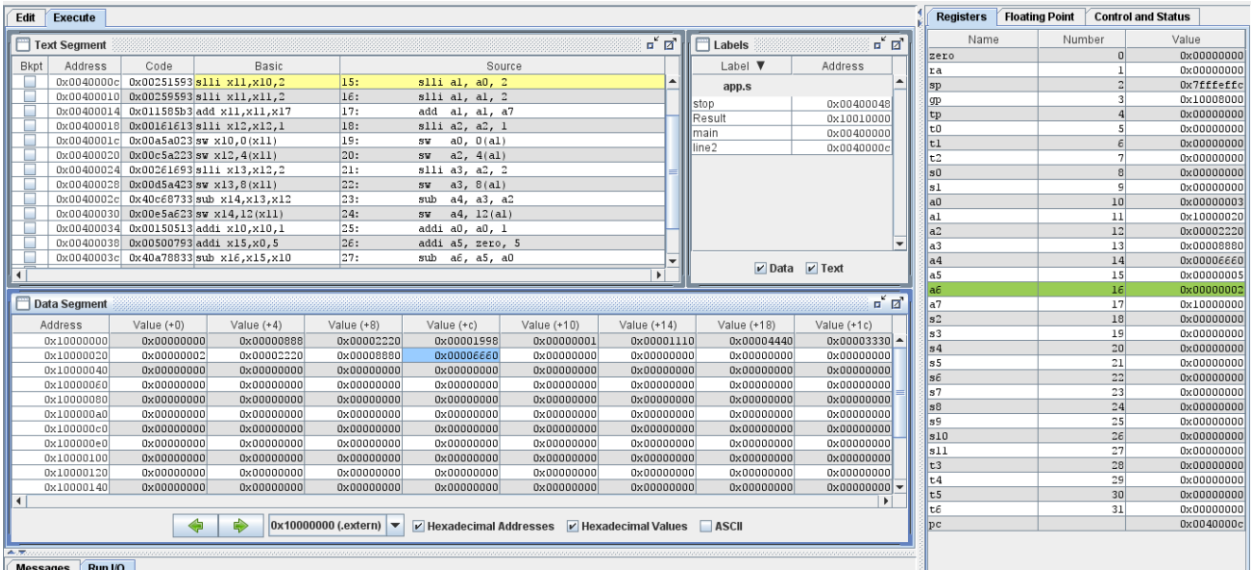

Figure 8 - RARS Result at second checkpoint

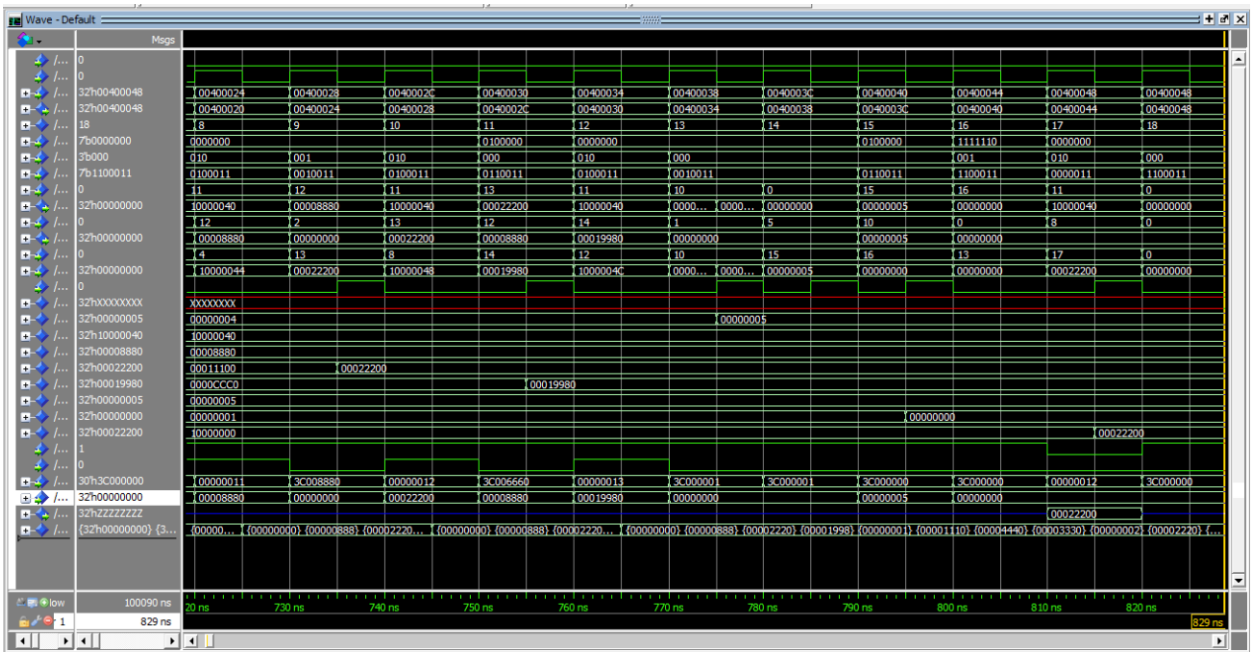

Figure 9 - Processor result at Final Checkpoint (steady state at end of program)

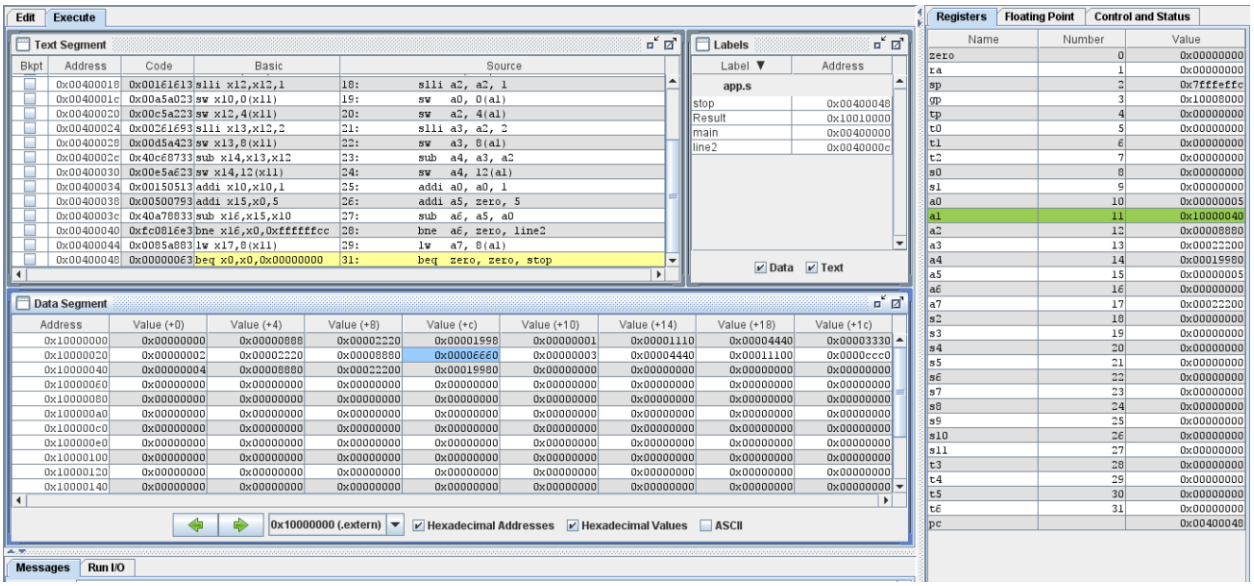

Figure 10 - RARS Result at final checkpoint

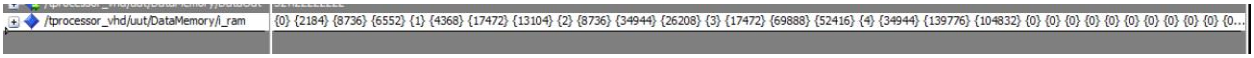

Figure 11 - Lab #6 final DRAM data

To Define what the correct expected results for the data memory is, I simulated it through RARS utilizing the app.s file within lab 6 shown in Figure 1. When comparing the final Data memory values where it reaches a steady state from looping the final branch instruction with respect to RARS final data memory values (Figure 1). I switched both programs' data representation to Unsigned fixed number for better readability and comparison. And after comparing the values between the two I confirmed that my processors data memory matched that of RARS.

Registers | Floating Point | Control and Status

Name	Number	Value
zero	0	0x00000000
ra	1	0x00000000
sp	2	0x7fffeffc
gp	3	0x10008000
tp	4	0x00000000
t0	5	0x00000000
t1	6	0x00000000
t2	7	0x00000000
s0	8	0x00000000
s1	9	0x00000000
a0	10	0x00000005
a1	11	0x10000040
a2	12	0x00008880
a3	13	0x00022200
a4	14	0x00019980
a5	15	0x00000005
a6	16	0x00000000
a7	17	0x00022200
s2	18	0x00000000
s3	19	0x00000000
s4	20	0x00000000
s5	21	0x00000000
s6	22	0x00000000
s7	23	0x00000000
s8	24	0x00000000
s9	25	0x00000000
s10	26	0x00000000
s11	27	0x00000000
t3	28	0x00000000
t4	29	0x00000000
t5	30	0x00000000
t6	31	0x00000000
pc		0x00400048

Figure 12 - Final Register Values through RARS

Figure 13 - Final Register Values from Processor

When comparing the final register values between Figure 12(RARS) and Figure 13(Lab Processor) I can conclude that they do match values for each register. Disclaimer, x0 out shows no value and that is how it is designed. In a previous lab I designed x0 to be a 'phantom' register that cant be written to and when called on would automatically output 32 bits of 0's not from the 32 bit register component. The x0 32 bit register component is merely a visual place holder to show in simulations.

GitHub Repo Link: https://github.com/bphilipose1/ECEGR2022/tree/main/Lab_6