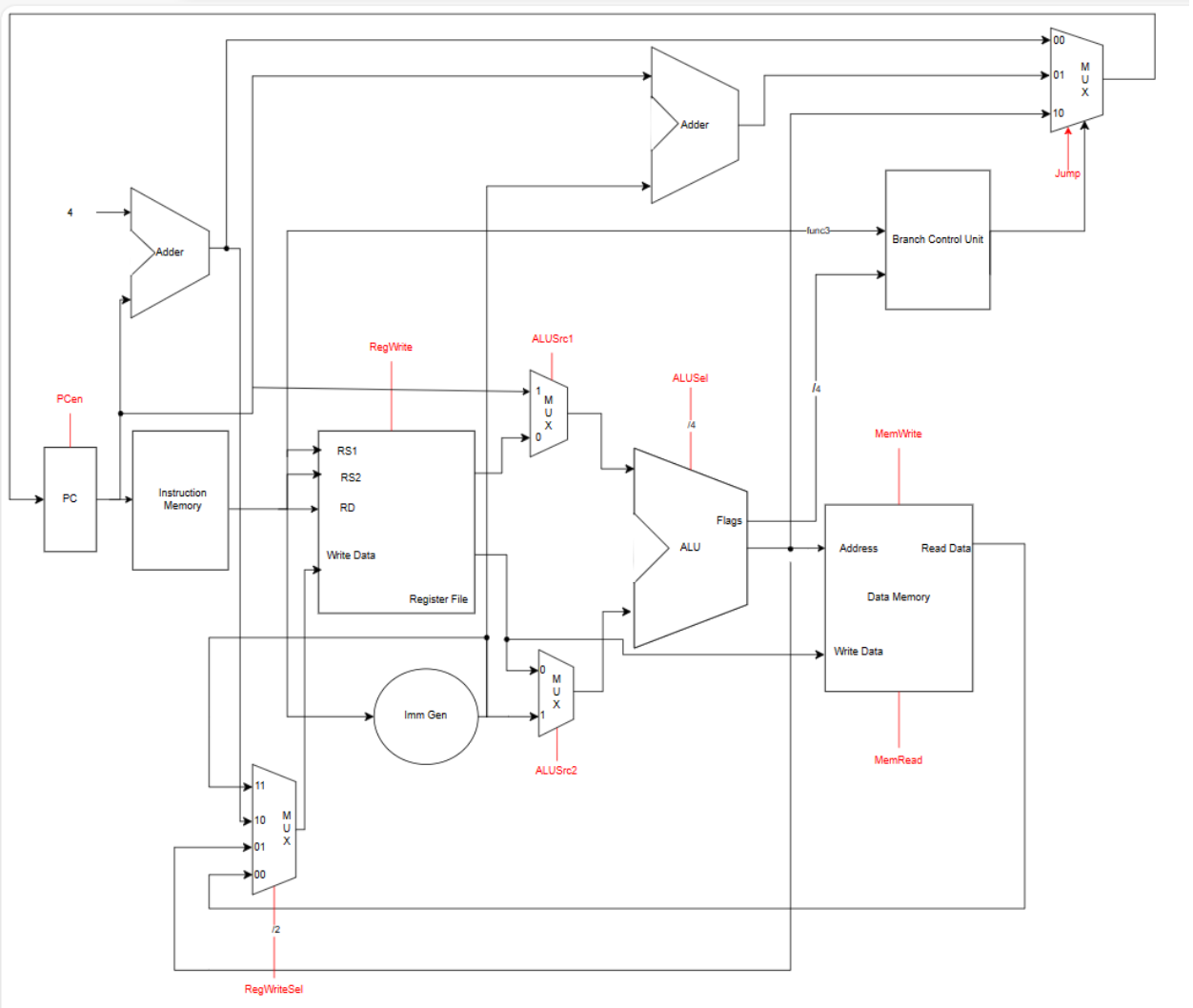
CSCE 3301 – Computer Architecture
Project 1: SINGLE CYCLE CPU
Nour Kasaby - 900211955
Nadine Safwat - 900212508

1. Project Objectives
   The main aim of this milestone is to create a fully functional RV32I single cycle processor which supports the 40 instructions including FENCE, ECALL and EBREAK where FENCE and ECALL act as nops and EBREAK acts as a halting instruction.

2. Design of the project



3. Implementation
   - PC: Implemented as a register that increments by 4 if the PCen signal is on, it will also check the BranchAnd signal and the Jump signal to either add the immediate of the ALU_out to the PC instead of adding 4 (To branch or jump).

   - Instruction Memory: A very simple register file of size 256 (for testing purposes) with each row of size 32 bits. Seeing as we created a single cycle processor we made this memory for instructions only. It will receive the address of the instruction from the PC and will read it.

   - RegisterFile: We pass in the read and write sources which we can extract from the Instruction we get from the InstMem module. We will reset all registers to 0 if the rst signal is on, otherwise, using the given inputs we will always read the two sources but only write to the RD if the WriteReg signal is on.
   - ALU: The ALU takes in the chosen ALU_A as A and ALU_B as B and ALUSel as S. The ALU will carry out arithmetic or logical operations on the two inputs A and B according to

the ALUSel provided. It will also assign all the flags which will be used in the Branch Control Unit (All flags are generated by subtracting B from A).

- DataMemory: Same implementation as the Instruction memory, except it receives the ALU_out as the address to be read or written from. Because this is a single ported memory, it will read the data in the given address if the readMem signal is on or it will write data to the given address if the WriteMem signal is on. We also pass in the funct3 to know the number of bytes that must be read or written (In case of store or load instructions).

- Control Units: (Not displayed in datapath for simplicity)
  - Control Unit: The control unit assigns the control signals of each instruction based on the last bits of the opcode.
  - ALU Control Unit: The ALU control unit will receive func3 of every instruction and the ALUOp and accordingly it will assign the ALUSel which would decide which ALU operation to be carried out in the ALU.
  - Branch Control Unit: The branch control unit will take in func3 and flags of the branch instructions and will assign the BranchAnd according to the specified flag comparisons for each branch type.

4. Difficulties Encountered and Solutions

Seeing as we worked on the project at home and had no access to vivado, our biggest obstacle was attempting to write our modules without an efficient syntax checker. However, after writing all our modules we were able to go to the lab in uni and test all of our code and most syntax and logic errors were discovered easily using the simulation tool in vivado.

When it came to testing, we attempted to write our programs on ears and the dump into a hex file, but somehow the dump always resulted in an empty file, therefore we had to use the decoder recommended to us on the Architecture forum to translate our instructions into machine code that we then pasted directly into our memories

5. Testing Procedure

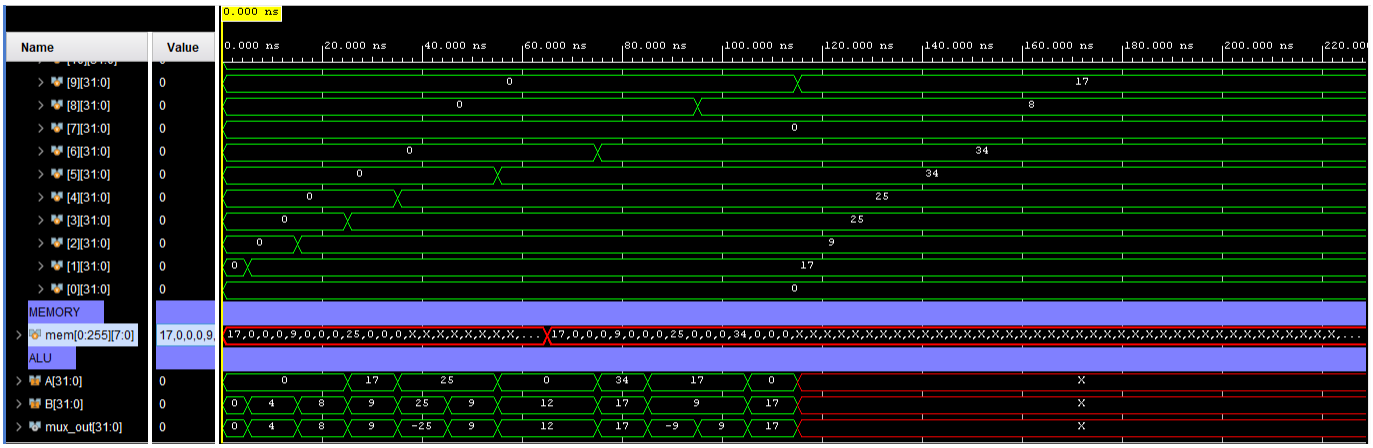
We split the testing of the 40 instructions on a couple of test programs, each test program testing a few different instructions. We did 4 test programs in total aside from the one provided from the lab, three of which were programs that we wrote simply to test whether the instructions work or not rather than being an actual function which does a certain operation and one test program was a function that writes an array in memory and swaps it.

- Test program (provided from the lab):

```
(INST)
000000000000_00000_010_00001_0000011 ; //lw x1, 0(x0)
000000000100_00000_010_00010_0000011 ; //lw x2, 4(x0)
000000001000_00000_010_00011_0000011 ; //lw x3, 8(x0)
0000000_00010_00001_110_00100_0110011 ; //or x4, x1, x2
0_000000_00011_00100_000_0100_0_1100011; //beq x4, x3, 4
0000000_00010_00001_000_00011_0110011 ; //add x3, x1, x2
0000000_00010_00011_000_00101_0110011 ; //add x5, x3, x2
0000000_00101_00000_010_01100_0100011; //sw x5, 12(x0)
000000001100_00000_010_00110_0000011 ; //lw x6, 12(x0)
0000000_00001_00110_111_00111_0110011 ; //and x7, x6, x1
0100000_00010_00001_000_01000_0110011 ; //sub x8, x1, x2
```

```
0000000_00010_00001_000_00000_0110011 ; //add x0, x1, x2
0000000_00001_00000_000_01001_0110011 ; //add x9, x0, x1

(DATA)
00000000000000000000000000010001
00000000000000000000000000001001
00000000000000000000000000011001
```

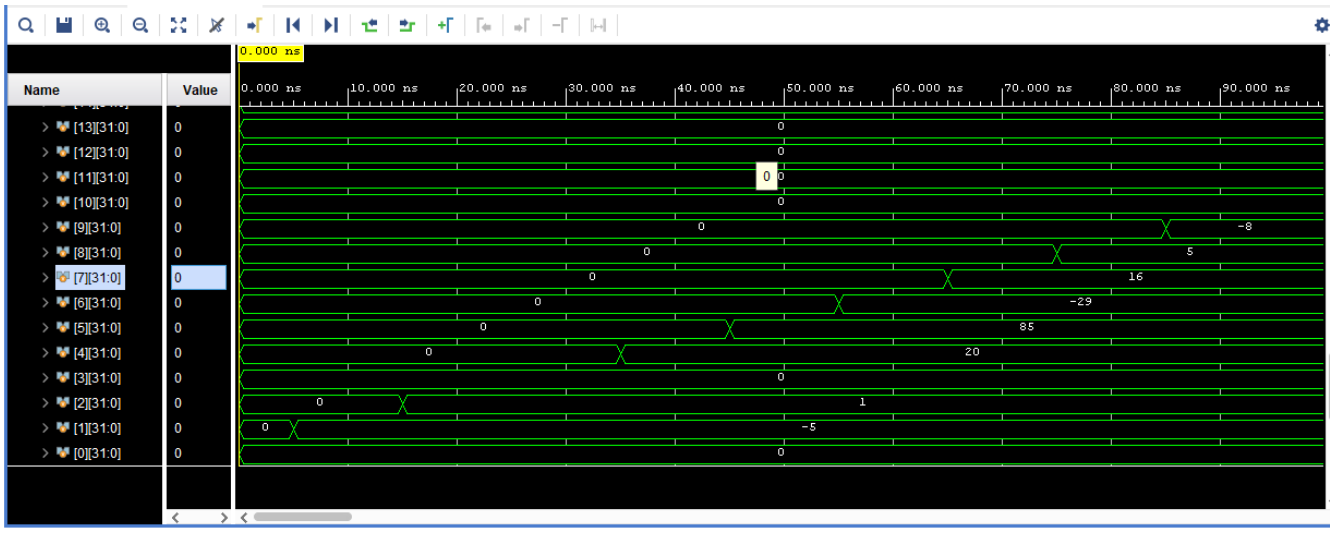


- Test program 1:
  - Assembly:

```
addi x1, x0, -5 #X1 = -5
slti x2, x1, 3 #x2 = 1
sltiu x3,x1, 3 #x3 = 0
xori x4, x3, 20 #x4 = 20
ori x5, x4, 85 #x5 = 85
andi x6, x1, -25 #x6 = -29
slli x7, x2, 4 #X7 = 16
srli x8, x4, 2 #x8 = 5
srai x9, x6, 2 #x9 = -8
```

  - Machine Code:

```
11111111101100000000000010010011
00000000001100001010000100010011
00000000001100001011000110010011
00000001010000011100001000010011
00000101010100100110001010010011
11111110011100001111001100010011
00000000010000010001001110010011
00000000001000100101010000010011
01000000001000110101010010010011
```



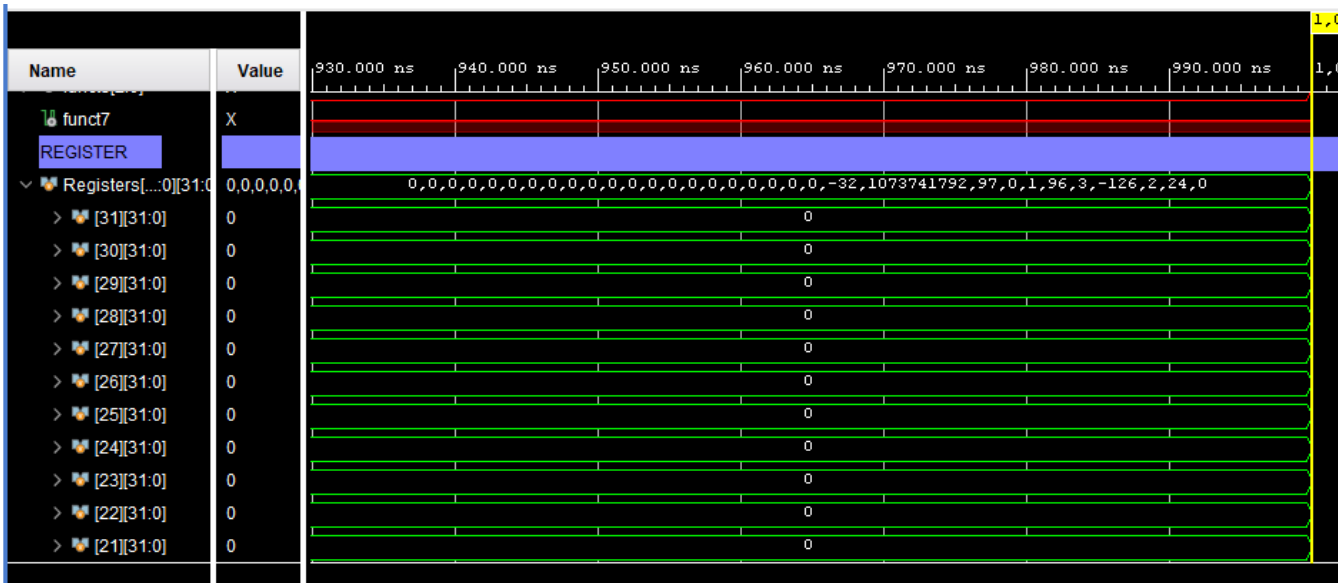
- Test Program 2:
  - Assembly:

```
lbu x1, 0(x0) #x1 = 24
lh x2, 4(x0) #x2 = 2
lb x3, 8(x0) #x3 = -126
lhu x4, 12(x0) #x4 = 3
sll x5, x1, x2 #x5 = 96
slt x6, x3, x1 #x6 = 1
sltu x7, x3, x1 #x7 = 0
xor x8, x6, x5 #x8 = 97
srl x9, x3, x2 #x9 = 1073741792
sra x10,x3, x2 #x10 = -32
```

  - Machine Code:

```
(INST)
00000000000000000100000010000011
00000000010000000001000100000011
00000000100000000000000110000011
00000000110000000101001000000011
00000000000100000100100101100011
00000000000010001101000110011011
00000000000010001101100111011011
00000000010100110100010000110011
00000000000100001110101001011011
01000000001000011101010100110011
(DATA)

00000000000000000000000000011000
00000000000000000000000000000010
00000000000000000000000010000010
00000000000000000000000000000011
```



- Test 3:
  - Assembly:

```
lui x1, 2 #x1 = 8192
auipc x2, 5 #x2 = 20484
jal x3, 8 # x3 = 12 ; PC => 16
ebreak          #Halt PC
bne x1, x2, 8 #PC => 24
ecall
addi x4, x0, -56    #x4 = -56
blt x1, x2, 8 #PC => 36
ecall
sb x2, 0(x0)   #mem[0] = 4
bge x2, x1, 8 #PC => 48
ecall
bltu x1, x4, 8 #PC => 56
ecall
bgeu x4, x2, 8 #PC => 64
ecall
sh x4, 1(x0)    #mem[2:1] = -56
ecall           #nop
fence           #nop
fence.i         #nop
jalr x0, x3, 0 #PC => 12
```

  - Machine Code:

```
00000000000000000010000010110111
00000000000000000010100010001 0111
00000000100000000000000111101111
00000000000100000000000001110011
00000000001000001001010001100011
00000000000000000000000001110011
11111100100000000000001000010011
00000000001000001100010001100011
00000000000000000000000001110011
00000000001000000000000000100011
00000000000100010101010001100011
00000000000000000000000001110011
```

```
00000000010000001110010001100011
00000000000000000000000001110011
00000000001000100111010001100011
00000000000000000000000001110011
00000000010000000001000010100011
00000000000000000000000001110011
00001111111100000000000000001111
00000000000000000001000000001111
00000000000000011000000001100111
```

- Test 4

```
00000000000100000000001010010011; //addi t0, zero, 1 # x = 1
00000000000000000000001100010011; //addi t1, zero, 0 # i = 0
00000000011000000000001110010011; //addi t2, zero, 6 # y = 6
00000000011100101000101001100011; //beq t0, t2, endStore # i == y?
00000000010100110010000000100011; //sw t0, 0(t1) # mem[i] = x
00000000010000110000001100010011; //addi t1, t1, 4 # i+=4
00000000000010010100000010010011; //addi t0, t0, 1 # x++
11111110000000000000100011100011; //beq zero, zero, store # loop back
00000000000000000000001010010011; //addi t0, zero, 0 # i = 0
00000000000000000000001100010011; //addi t1, zero, 0 # *i
```

```
00000001000000000000000001110010011; //addi t2, zero, 16 # *(4-i)
00000000001100000000111000010011; //addi t3, zero, 3 # x = 3
00000011110000010100000100110011; //beq t0, t3, endSwap # i == x?
00000000000000011001011110100000011; //lw t4, 0(t1) # temp1 = mem[i]
00000000000000011101011111000000011; //lw t5, 0(t2) # temp2 = mem[4-i]
00000001110100111010000000100011; //sw t4, 0(t2) # mem[4-i] = temp1
00000001111000110010000000100011; //sw t5, 0(t1) # mem[i] = temp2
00000000000100101000001010010011; //addi t0, t0, 1 # i++
00000000010000110000001100010011; //addi t1, t1, 4 # *(i+1)
11111111110000111000001110010011; //addi t2, t2, -4 # *(4-i-1)
11111110000000000000000011100011; //beq zero, zero, swap # loop back
```