Project 3: ISA Simulation

ECE 366 - FA18
Professor Rao
November 8th, 2018

GROUP 12

Yihua Pu - ypu5
Singee Nguyen - snguye38
Jenshin Chen - chen172

Part A - ISA Intro

Name: KISS, Keep It Short and Sweet

For our ISA, we decided to focus on pinpointed performance rather than versatility. Our instructions are powerful for use in this project specifically, but would not fare well in other programs. For example, our branch instructions, jpu1 and jpu2, utilize muxes to deliberately branch to specific addresses, rather than jumping to an instruction relative to PC+1. This allows us to keep our programs short and simple, as our ISA name implies.

Part B - Answers to Questions

1. The best part about our ISA is how direct it is. The programs are very short, and the hardware implementation isn't overly difficult, although it seems that any multi-cycle circuits require a fair amount of work. Our ISA's main limitation is that it really can't do anything besides the two programs that we've written for it, and this is probably the biggest compromise we made to make things work. It was designed specifically for the purpose of our class, but that's it!
2. Since the base number for program 1 is six, it meant that we would always be multiplying by six. We took advantage of this being equivalent to multiplying a number by 4 and by 2 separately using logical shifts, and then summing them together, and this both lowered our DIC since we wouldn't have to do repeated addition for multiplication and simplified our hardware implementation since we wouldn't need to build a multiplier circuit.
   One way that we could have improved the multi-cycle FSM would be to optimize lw and sw by switching around XX and YY in the opcode, or something equivalent in its function. That way, we could share the same bubble as add and inc in our FSM design.
3. (a). We learned that pipeline circuits are very tedious to implement. Multi-cycle circuits already require us to keep track of a large amount of information in the form of control signals, and pipelined ISA designs would clearly require many more signals, in addition to the memory management that is necessary for the pipeline to work efficiently. This was also the best/worst part of the project. It was effective in showing us glimpses into how to design a pipelined ISA, but it did so using brute force. With multi-cycle circuits, we were able to avoid dealing with any control hazards, as well.
   (b). Write neatly and be organized; it's very easy to lose track of what values your signals are. Start as soon as possible, too! Try to work on it as a group if possible so that everybody is on the same page as you move forward.
   (c) . It was useful practice in keeping track of a bunch of little things, and we learned how these little things could be put together in a specific fashion to accomplish some task. In addition, we also learned how important it can be to be able to keep track of the details instead of being too caught up in the bigger picture and losing focus. Regarding the content of the project itself, we were able to gain a close understanding of how different CPU design techniques can process instructions for greater speed and efficiency, with tradeoffs in time or money.

1. Instruction list

Instruction	PC	Coding	Functionality	Example	
init Rx, imm	PC++	000 xx ii	Rx = MUX[imm] Imm will go into a MUX to select specific hardwrited number(0,1,6,108)	init R1, 6	000 01 10
ld Rx, Ry	PC++	001 xx yy	Rx = Mem[Ry]	ld R3, R2	001 11 10
st Rx, Ry	PC++	010 xx yy	Mem[Ry] = Rx	st R0, R3	010 00 11
add Rx, Ry	PC++	011 xx yy	Rx = Rx + Ry	add R1, R1	011 01 01

jpu1 Rx,Ry,imm	if Rx < Ry:
	PC = MUX(imm)
	else:
	PC++
		100 x y ii	Rx ∈ {R0, R1}
			x = 0 1
			Ry ∈ {R2, R3}
			y = 0 1
			Imm number will go into a MUX to select specific jumps
				jpu1 R0, R2, 6
				MUX
				00 9
				01 6
				10 24
				11 18
					100 0 0 01

jpu2 Rx,Ry,imm	if Rx < Ry:
	PC = MUX(imm)
	else:
	PC++
		101 x y ii	Rx ∈ {R2, R3}
			Ry ∈ {R0, R1}
			Same as jpu1, but the decoding of registers changed.
				jpu2, R1, R3, 8
				MUX
				00 14
				01 8
				10 27
					101 1 1 01

subR3 Rx	PC++	11100 xx	R3 = R3 - Rx	subR3 R2	11100 10
inc Rx	PC++	11101 xx	Rx = Rx + 1	Inc R3	11101 11
R3x6	PC++	1111110	R3 = R3 * 6	R3x6	11111110
score	PC++	1111111	R3 = the match score of R3 and R1. This function is done using logic circuit.	Score	1111111

2. Register Design

Register Name	Number
R0	00
R1	01
R2	10
R3	11

3. Control Flow

Since there are total 7 branches used in our Program1 and Program2, the instruction address of all these branches are constant. We save all these address into MUXs, and use the immediate number from machine code to directly select values from MUX.
Accordingly, there is no need for us to calculate the target addresses.

4. Memory Model

4.1 Data Memory

- 16-bit double-byte addressable
- 128memory units in total
- using 7-bit address.

Address	Memory
000 0000	Mem[0]
000 0001	Mem[1]
...	...
111 1111	Mem[127]

4.2 Instruction Memory

- 8-bit byte addressable, PC is initialized at 0

- 64 memory units in total
- using 6-bit address.

Address	Memory
00 0000	Mem[0]
00 0001	Mem[1]
...	...
11 1111	Mem[63]

Part C - Simulation Results

1. Execution Results

Sample Data Pattern A	Dynamic Instr Count: 98 Total Cycle Count: 347 Registers R0-R3: [9, 2, 9, 11] Data Memory : Addr 0: HEX:00000009 DEC: 9 Addr 1: HEX:00000011 DEC: 17 Addr 2: HEX:0000000b DEC: 11 Addr 3: HEX:00000000 DEC: 0 Addr 4: HEX:00000000 DEC: 0 Addr 5: HEX:00000000 DEC: 0 ******** Simulation starts ********* ******** Simulation finished ********* Dynamic Instr Count: 1048 Total Cycle Count: 3850 Registers R0-R3: [5, 10, 7, 108] Data Memory : Addr 0: HEX:00000009 DEC: 9 Addr 1: HEX:00000011 DEC: 17
Sample Data Pattern B	Dynamic Instr Count: 3004 Total Cycle Count: 10776 Registers R0-R3: [267, 2, 267, 2415] Data Memory : Addr 0: HEX:0000010b DEC: 267 Addr 1: HEX:00001003 DEC: 4099 Addr 2: HEX:0000096f DEC: 2415 Addr 3: HEX:00005555 DEC: 21845 Addr 4: HEX:00000000 DEC: 0 Addr 5: HEX:00000000 DEC: 0 ******** Simulation starts ********* ******** Simulation finished ********* Dynamic Instr Count: 1104 Total Cycle Count: 4015 Registers R0-R3: [5, 12, 4, 108] Data Memory : Addr 0: HEX:0000010b DEC: 267 Addr 1: HEX:00001003 DEC: 4099 Addr 2: HEX:0000096f DEC: 2415 Addr 3: HEX:00005555 DEC: 21845 Addr 4: HEX:0000000c DEC: 12 Addr 5: HEX:00000004 DEC: 4

Our Data Pattern C	Dynamic Instr Count: 41 Total Cycle Count: 145 Registers R0-R3: [6, 2, 6, 0] Data Memory : Addr 0: HEX:00000006 DEC: 6 Addr 1: HEX:0000b640 DEC: 46656 Addr 2: HEX:00000000 DEC: 0 Addr 3: HEX:00000000 DEC: 0 Addr 4: HEX:00000000 DEC: 0 Addr 5: HEX:00000000 DEC: 0 ******** Simulation starts ********* ******** Simulation finished ********* Dynamic Instr Count: 1056 Total Cycle Count: 3871 Registers R0-R3: [5, 12, 6, 108] Data Memory : Addr 0: HEX:00000006 DEC: 6 Addr 1: HEX:0000b640 DEC: 46656 Addr 2: HEX:00000000 DEC: 0 Addr 3: HEX:00000000 DEC: 0 Addr 4: HEX:0000000c DEC: 12 Addr 5: HEX:00000006 DEC: 6
Our Data Pattern D	Dynamic Instr Count: 690852 Total Cycle Count: 2482998 Registers R0-R3: [65021, 2, 65021, 53730] Data Memory : Addr 0: HEX:0000fdfd DEC: 65021 Addr 1: HEX:0000f33f DEC: 62271 Addr 2: HEX:0000d1e2 DEC: 53730 Addr 3: HEX:0000e60d DEC: 58893 Addr 4: HEX:0000dfff DEC: 57343 Addr 5: HEX:0000bedc DEC: 48860 ******** Simulation starts ********* ******** Simulation finished ********* Dynamic Instr Count: 1032 Total Cycle Count: 3800 Registers R0-R3: [5, 13, 1, 108] Data Memory : Addr 0: HEX:0000fdfd DEC: 65021 Addr 1: HEX:0000f33f DEC: 62271 Addr 2: HEX:0000d1e2 DEC: 53730 Addr 3: HEX:0000e60d DEC: 58893 Addr 4: HEX:0000000d DEC: 13 Addr 5: HEX:00000001 DEC: 1

2. Execution Progress of the Target Programs

	Program 1	Program 2

Sample Data Pattern A	0000000
	init R0, 0
	
	0011000
	ld R2, R0
	
	0000001
	init R0, 1
	
	0010100
	ld R1, R0
	
	0001110
	init R3, 6
	
	Registers R0-R3: [1, 17, 9, 6]
	Program Counter : 5
		0000001
		init R0, 1
		
		1110100
		
		inc R0
		
		1110100
		
		inc R0
		
		0010100
		ld R1, R0
		
		0001010
		
		init R2,6
		
		Registers R0-R3:
		[3, 0, 6, 0]
		Program Counter :
		5
		Press any key to continue

Sample Data Pattern B	0000000
	init R0, 0
	
	0011000
	ld R2, R0
	
	0000001
	init R0, 1
	
	0010100
	ld R1, R0
	
	0001110
	init R3, 6
	
	Registers R0-R3: [1, 4099, 267, 6]
	Program Counter : 5
	Press any key to continue
		0000001
		init R0, 1
		
		1110100
		inc R0
		
		1110100
		inc R0
		
		0010100
		ld R1, R0
		
		0001010
		init R2,6
		
		Registers R0-R3: [3, 21845, 6, 0]
		Program Counter : 5
		Press any key to continue

Our Data Pattern C	0000000
	init R0, 0
	
	0011000
	ld R2, R0
	
	0000001
	init R0, 1
	
	0010100
	ld R1, R0
	
	0001110
	init R3, 6
	
	Registers R0-R3: [1, 46656, 6, 6]
	Program Counter : 5
	Press any key to continue
		0000001
		init R0, 1
		
		1110100
		inc R0
		
		1110100
		inc R0
		
		0010100
		ld R1, R0
		
		0001010
		init R2,6
		
		Registers R0-R3: [3, 0, 6, 0]
		Program Counter : 5
		Press any key to continue

Our Data Pattern D	0000000
	init R0, 0
	
	0011000
	ld R2, R0
	
	0000001
	init R0, 1
	
	0010100
	ld R1, R0
	
	0001110
	init R3, 6
	
	Registers R0-R3: [1, 62271, 65021, 6]
	Program Counter : 5
	Press any key to continue
		0000001
		init R0, 1
		
		1110100
		inc R0
		
		1110100
		inc R0
		
		0010100
		ld R1, R0
		
		0001010
		init R2,6
		
		Registers R0-R3: [3, 58893, 6, 0]
		Program Counter : 5
		Press any key to continue

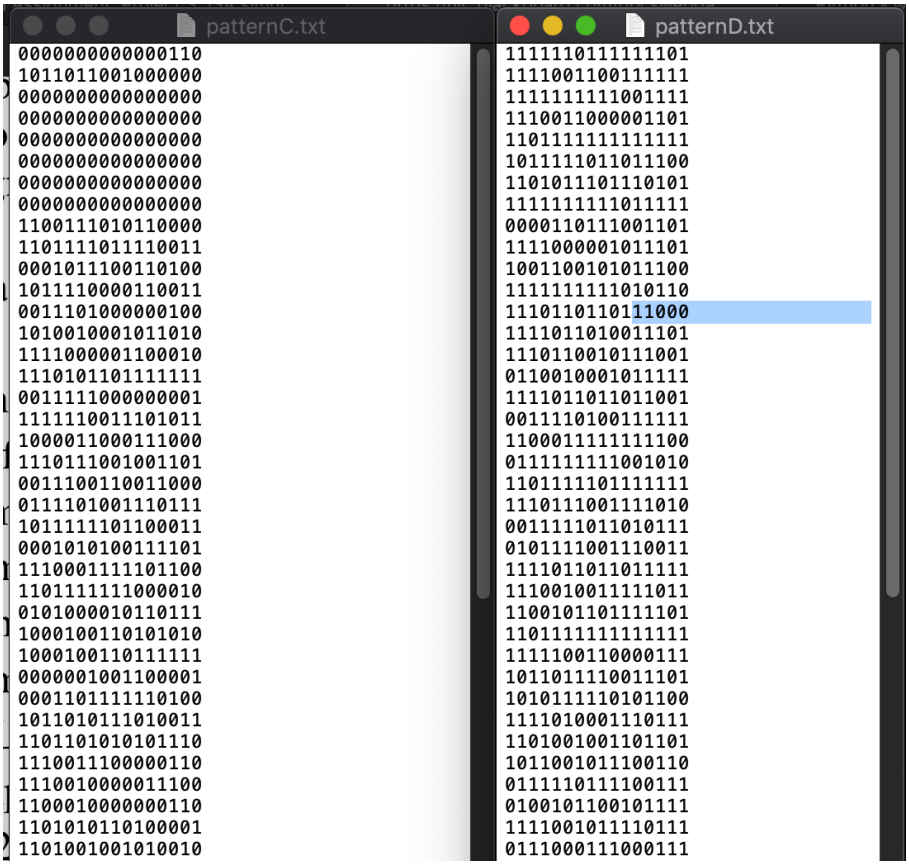
Part D. ISA package

1. Algorithms & Machine Code

```
# Program 1 by Group12 for Project3
# 28, Oct, 2018

00000000              # init R0, 0
00011000              # ld R2, R0           # R2 = P
10000001              # init R0, 1
00010100              # ld R1, R0           # R1 = Q
10001110              # init R3, 6          # R3 = MUX[3] = 6, the result of calculation
10000001              # init R0, 1          # the exponential counter
L1:                   # jpu1 MUX[01] = 6 (instruction mem address)
01111110              # R3x6                # R3 = R3 * 6
01110100              # inc R0
10111101              # add R3, R1          # R3 = R3 + Q
L2:                   # jpu1 MUX[00] = 9 (instruction mem address)
01110001              # subR3 R1            # R3 = R3 - Q
11001100              # jpu1 R1, R3, 9      # jump to L2 if R1 < R3 (Q < Result) there should be Q<= Result
01000001              # jpu1 R0, R2, 6      # jump to L1 if R0 < R2 (counter < P)
Finish:
01011100              # jpu2 R3, R1, 14     # R3 should <= R1(Q) here, if R3 < R1 (Result < Q) jump to END
01110001              # subR3 R1            # PC goes here only if R3 = R1 (Result = Q), when should subtract Q one more time
END:                  # jpu2 MUX[00] = 14 (instruction mem address)
00000101              # init R1, 1
11110101              # inc R1              # R1 = 2
00101101              # st R3, R1
```

```
# Program 2 by Group12 for Project3
# 28, Oct, 2018

10000001               # init R0, 1
01110100               # inc R0
01110100               # inc R0              # R0 = 3
00010100               # ld R1, R0           # R1 = Target Pattern
00001010               # init R2,6
11110110               # inc R2
11110110               # inc R2              # R2 = 8, Memory Pointer
00000011               # init R0, 108        # R1 = 108, exit memory address
GetScore:              # jpu2 MUX[01] = 8 (instruction mem address)
00011110               # ld R3, R2           # R3 = Current pattern
11111111               # score               # calculate the number of same bits between R1 and R3
00101110               # st R3, R2
11110110               # inc R2
11010001               # jpu2 R2, R0, 8
10000010               # init R0, 6
01110100               # inc R0
01110100               # inc R0              # R0 = 8, Memory Pointer
10000100               # init R1, 0          # the best score
10001000               # init R2, 0          # the best score counter
FindBest:              # jpu1 MUX[11] = 18 (instruction mem address)
10011100               # ld R3, R0
11011110               # jpu2 R3, R1, 27     # if (pointed score) < (best score), jump to LessThanBest
01001110               # jpu1 R1, R3, 24     # if (best score) < (curretn score), jump to BiggerThanBest
11110110               # inc R2              # pc arrives here -> R1 = R3
00001100               # init R3, 0
01011010               # jpu2 R3, R0, 27     # R3=0 MUST < R0, jump condition is met, jump to LessThanBest
BiggerThanBest:        # jpu1 MUX[10] = 24 (instruction mem address)
10000100               # init R1, 0
10110111               # add R1, R3          # reset best score
00001001               # init R2, 1          # reset counter = 1
LessThanBest:          # jpu2 MUX[11] = 27 (instruction mem address)
01110100               # inc R0
00001111               # init R3, 108
01000111               # jpu1 R0, R3, 18     # if (address pointer) < 108, jump back; else go out
Out:
10000001               # init R0, 1
01110100               # inc R0
01110100               # inc R0
01110100               # inc R0              # R0 = 4
00100100               # st R1, R0           # store the best score
01110100               # inc R0              # R0 = 5
00101000               # st R2, R0           # store the counter
```

2. Pattern C & Pattern D



Pattern C:

- For program1, Pattern C makes sure that in 6^P % Q, even if the remainder is 0, our program can still get the answer correctly.
- For program2, target pattern is all 0, the patterns need to be graded are random numbers produce by a Binomial distribution with p = 0.5.

Pattern D:

- All the numbers in Pattern D are random numbers produced by Binomial distribution with p = 0.667, to make sure that our both programs can work on arbitrary data.

3. Python Simulator Code

```python
# Simulator and Assembler by ECE366 Project3 Group12
# take instructions as input, convert it to machine code
# take machine code as input to run the simulator


def simulate(MC, Instr, Nsteps, debug_mode, Memory):
    PC = 0              # Program-counter
    DIC = 0
    TotalCycle = 0
    Reg = [0, 0, 0, 0]     # 4 registers, init to all 0
    print("******** Simulation starts *********")
    finished = False
    while not finished:
        fetch = MC[PC]
        fetch = fetch[1:]
        DIC += 1
        if debug_mode:
            print(fetch)
            print(Instr[PC])
            print()
        if fetch[0:3] == "000":  # init
            MUX = [0, 1, 6, 108]
            R = int(fetch[3:5],2)
            MUXindex = int(fetch[5:7],2)
            imm = MUX[MUXindex]
            Reg[R] = imm
            PC += 1
            TotalCycle += 2
        elif fetch[0:3] == "001":  # ld
            Rx = int(fetch[3:5], 2)
            Ry = int(fetch[5:7], 2)
            Reg[Rx] = Memory[Reg[Ry]]
            PC += 1
            TotalCycle += 5
        elif fetch[0:3] == "010":  # st
            Rx = int(fetch[3:5], 2)
            Ry = int(fetch[5:7], 2)
            Memory[Reg[Ry]] = Reg[Rx]
            PC += 1
            TotalCycle += 4
        elif fetch[0:3] == "011":  # add
            Rx = int(fetch[3:5], 2)
            Ry = int(fetch[5:7], 2)
            Reg[Rx] = Reg[Rx] + Reg[Ry]
            PC += 1
            TotalCycle += 4
        elif fetch[0:3] == "100":  # jpu1
            MUX = [9, 6, 24, 18]
            Rx = int(fetch[3], 2)
            Ry = 2 + int(fetch[4], 2)
```

```
            imm = MUX[int(fetch[5:7], 2)]
            if Reg[Rx] < Reg[Ry]:
                PC = imm
            else:
                PC += 1
            TotalCycle += 3
        elif fetch[0:3] == "101":  # jpu2
            MUX = [14,8,27]
            Rx = 2 + int(fetch[3],2)
            Ry = int(fetch[4],2)
            imm = MUX[int(fetch[5:7],2)]
            if Reg[Rx] < Reg[Ry]:
                PC = imm
            else:
                PC += 1
            TotalCycle += 3
        elif fetch[0:5] == "11100":   # subR3
            Ry = int(fetch[5:7],2)
            Reg[3] = Reg[3] - Reg[Ry]
            PC += 1
            TotalCycle += 4
        elif fetch[0:5] == "11101":   # inc
            Rx = int(fetch[5:7],2)
            Reg[Rx] = Reg[Rx] + 1
            PC += 1
            TotalCycle += 4
        elif fetch == "1111110":      # R3x6
            Reg[3] = Reg[3]*6
            PC += 1
            TotalCycle += 4
        elif fetch == "1111111":      # score
            Rx = Reg[3]
            Ry = Reg[1]
            binRx = bin(Rx).replace('0b','')
```

```
            binRy = bin(Ry).replace('0b','')

            some0 = ''
            for i in range(16-len(binRx)):
                some0 += '0'
            binRx = some0 + binRx

            some0 = ''
            for i in range(16-len(binRy)):
                some0 += '0'
            binRy = some0 + binRy

            score = 0
            for i in range(0, 16):
                if binRx[i] == binRy[i]:
                    score += 1
            Reg[3] = score
            PC += 1
            TotalCycle += 4

        if PC == len(MC):
            finished = True
        if debug_mode:
            if (DIC % Nsteps) == 0:  # print stats every Nsteps
                print("Registers R0-R3: ", Reg)
                print("Program Counter : ", PC)
                input("Press any key to continue")
                print()
        else:
            continue
```

```
115
116         print("******** Simulation finished *********")
117         print("Dynamic Instr Count: ", DIC)
118         print("Total Cycle Count:   ", TotalCycle)
119         print("Registers R0-R3:     ", Reg)
120         print("Data Memory :")
121         for i in range(0,6):
122             print('Addr '+str(i)+": HEX:"+format(Memory[i], "016b")+"   DEC: "+str(Memory[i]))
123
124
125     def assemble(I, program_dupe):...
247
248     def main():
249         Memory = []
250         debug_mode = False  # is machine in debug mode?
251         Nsteps = 3          # How many cycle to run before output statistics
252         Instruction = []    # all instructions will be stored here
253         machineInstruction = []
254
255         print("Welcome to ECE366 Project3 Group12 ISA Simulator!")
256
257         # Read in instr and convert to machine code
258         print("type 1 for program 1")
259         print("type 2 for program 2")
260         print("type 3 for both program 1 and program 2")
261         program = int(input("Enter which program to run:"))
262         if program != 1 and program != 2 and program != 3:
263             print("wrong program selection!")
264             exit()
```

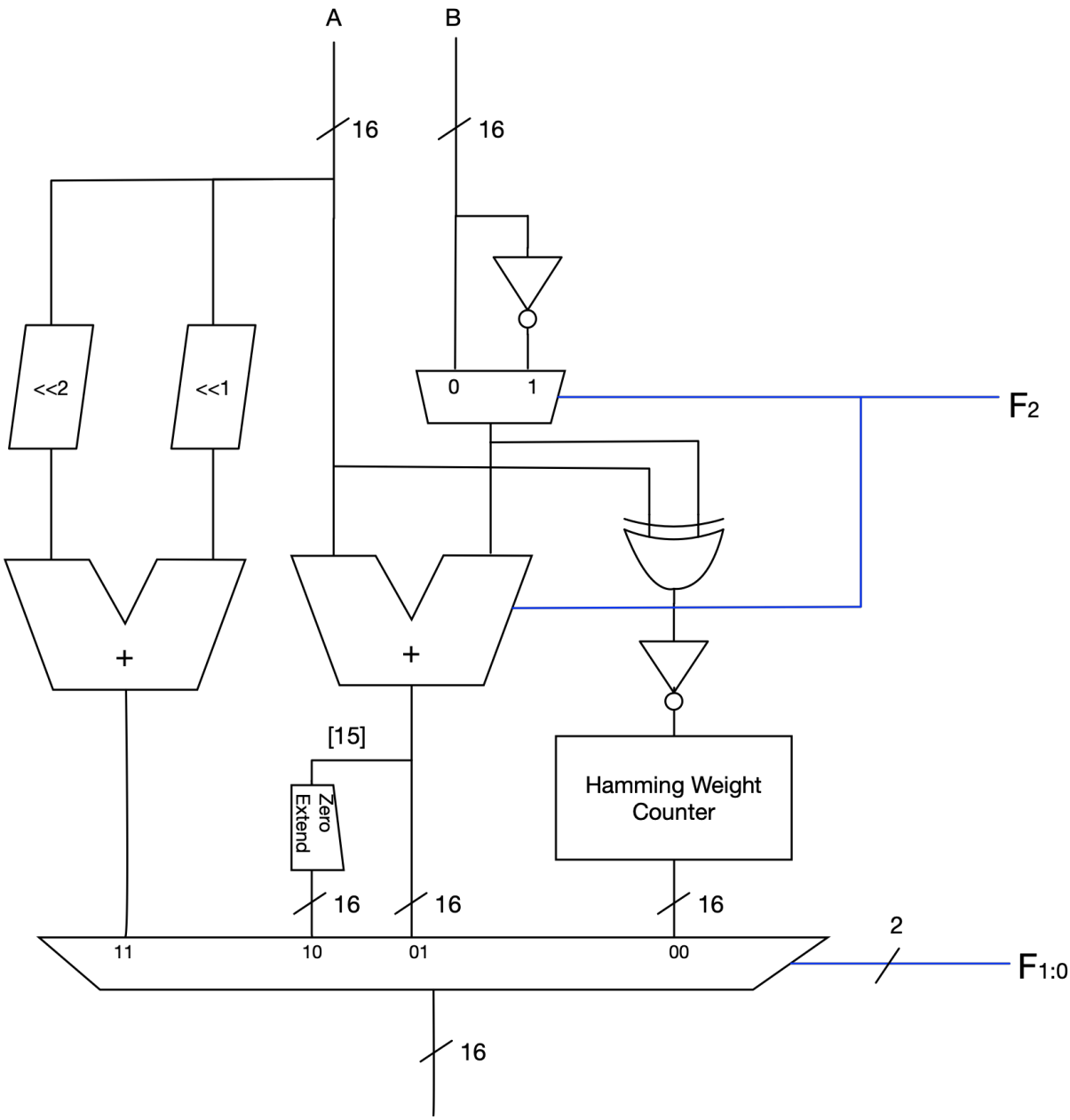
```
301         # read MachineCode
302         if program == 1 or program == 2:
303             Instruction = []
304             with open("p3_group_12_p"+str(program)+"_imem.txt", "r") as f:
305                 for line in f:
306                     if line == "\n" or line[0] == '#' or ':' in line:  # empty lines,comments
307                         continue
308                     mc = line.split('#')[0]
309                     inst = line.split('#')[1]
310                     machineInstruction.append((mc.strip()))
311                     Instruction.append((inst.strip()))
312         elif program == 3:
313             MC1 = []
314             MC2 = []
315             instr1 = []
316             instr2 = []
317             with open("p3_group_12_p1_imem.txt", "r") as f:
318                 for line in f:
319                     if line == "\n" or line[0] == '#' or ':' in line:  # empty lines,comments
320                         continue
321                     mc = line.split('#')[0]
322                     inst = line.split('#')[1]
323                     MC1.append((mc.strip()))
324                     instr1.append((inst.strip()))
325             with open("p3_group_12_p2_imem.txt", "r") as f:
326                 for line in f:
327                     if line == "\n" or line[0] == '#' or ':' in line:  # empty lines,comments
328                         continue
329                     mc = line.split('#')[0]
330                     inst = line.split('#')[1]
331                     MC2.append((mc.strip()))
332                     instr2.append((inst.strip()))
```

```
    # read Data Memory
    print("Data memory pattern we have:")
    print("1] patternA.txt")
    print("2] patternB.txt")
    print("3] patternC.txt")
    print("4] patternD.txt")
    pattern = int(input("Please select pattern:"))
    if pattern < 1 or pattern > 4:
        print("wrong data pattern selection")
        exit()
    pattern_pool = ['A', 'B', 'C', 'D']
    pattern = pattern_pool[pattern-1]
    with open('pattern'+pattern+'.txt', 'r') as data_file:
        for line in data_file:
            if line == "\n" or line[0] == '#':              # empty lines,comments ignored
                continue
            line = line.strip()
            Memory.append(int(line, 2))


    print("********* Simulator **********")
    print("Simulator has 2 modes: ")
    print(" 1] Normal execution")
    print(" 2] Debug mode")
    simMode = int(input("Please select simulator's mode: "))
    if simMode == 1:
        debug_mode = False
    elif simMode == 2:
        debug_mode = True
        Nsteps = int(input("Debug Mode selected. Please enter # of debugging steps: "))
    else:
        print("Error, unrecognized input. Exiting")
```

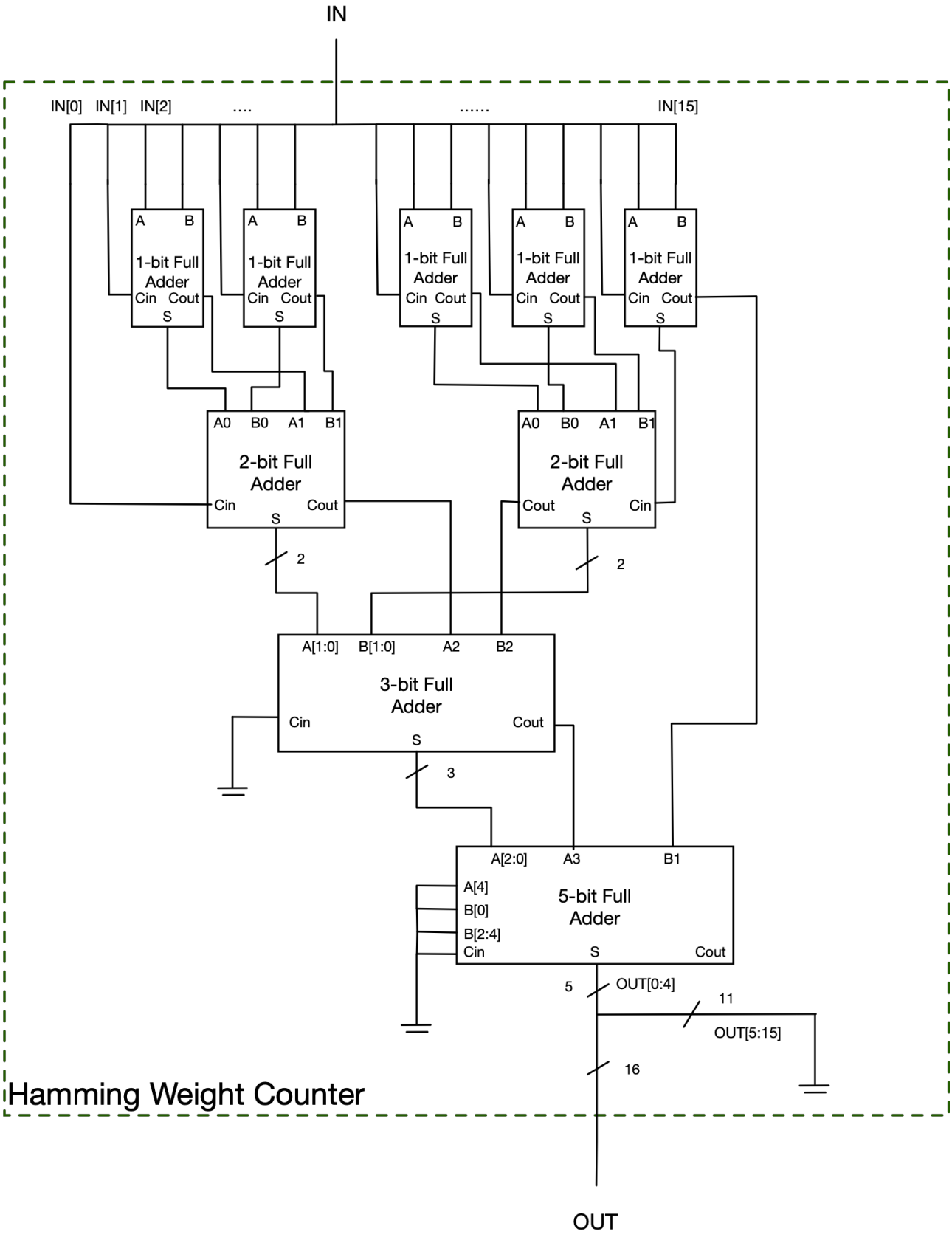
```
        exit()

    if program == 1 or program == 2:
        simulate(machineInstruction, Instruction, Nsteps, debug_mode, Memory)

    elif program == 3:
        simulate(MC1, instr1, Nsteps, debug_mode, Memory)
        simulate(MC2, instr2, Nsteps, debug_mode, Memory)

    # store the memory back
    data = open("p3_group_12_dmem_"+pattern+'.txt', "w")
    for i in range(len(Memory)):
        data.write(format(Memory[i], "016b"))
        data.write("\n")
    data.close()


if __name__ == "__main__":
    main()
```

4. Hardware Schematics

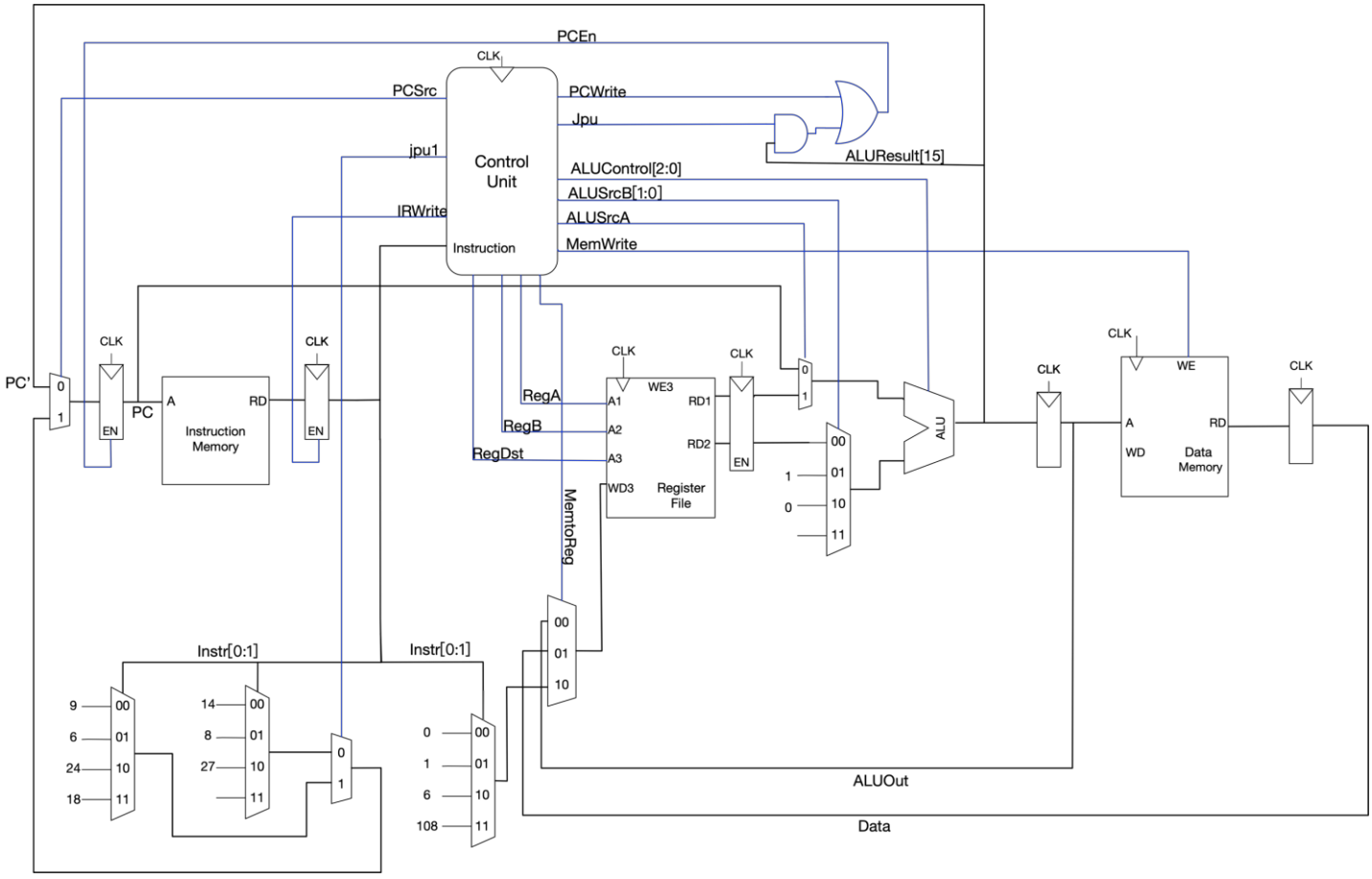
4.a. ALU Schematic



Hamming Weight Counter



4.b. CPU Datapath



4.c. FSM for multi-cycle implementation