ECE 366 Project 3, Group 4

(F2) Simulator

ISA

Part A. ISA Introduction

F2 (stands for Fast Assembly Super Turbo 2) ISA

Philosophy is to minimize use of loops except for looping through entries and have some instructions carry implicit details. A key component we focused on what capturing the counting of bits operation for program 2 in a single instruction that does not utilize loops. We also wanted to support more than 8 instructions by using unique bit encoding that would be particular to what the programs needed (non-generality).

1. Instruction list

Instruction	PC	Coding	Functionality	Example	
init Rx, imm	PC++	000 x iii	Rx = imm Rx ∈ {R0, R1} imm: [0,7]	init R0,4	000 0 100
ld Rx, Ry	PC++	001 xx yy	Rx = Mem[Ry]	ld R0, R1	001 00 01
str Rx, Ry	PC++	010 xx yy	Mem[Ry] = Rx Rx ∈ {R0, R1, R2, R3} Rx ∈ {R0, R1, R2, R3}	st R0, R1	010 00 01
addR Rx	PC++	01100 xx	R2 = Rx + Rx Rx ∈ {R0, R1, R2, R3}	addR R0	01100 00
addR2 Rx	PC++	01110 xx	R2 = R2 + Rx Rx ∈ {R0, R1, R2, R3}	addR2 R1	01110 01
addR3 Rx	PC++	01111 xx	R3 = Rx + Rx Rx ∈ {R0, R1, R2, R3}	addR3 R2	01111 10
subR3 Rx	PC++	01101 xx	R3 = R3 - Rx Rx ∈ {R0, R1, R2, R3}	subR3 R0	01101 00
addi Rx imm	PC++	100 xx ii	Rx = Rx + imm Rx ∈ {R0, R1, R2, R3} imm: [0,3]	addi R0, 2	100 00 01
sltR0 Rx,Ry	PC++	101 xx yy	R0 =1 if Rx < Ry Rx ∈ {R0, R1, R2, R3} Rx ∈ {R0, R1, R2, R3}	sltR0 R0,R1	101 00 11

beqR0 Rx imm	if Rx==RO:
	PC == MUX(imm)
	else:
	PC++
		11 xx iii	Rx ∈ {R0, R1, R3}
			Imm number used to select jumps in a mux
				beqR0 R0, brk	11 00 101

scrR3R2	PC++	1110 111	R3 = the match score of R3 and R2. This function is done using logic circuit.	scrR3R2	1110 111
haltPC	PC stable	1110 000	PC will not change	haltPC	1110 000

2. Register Design

Register Name	Number
R0	00
R1	01
R2	10
R3	11

3. Control Flow

Branch if equal uses a 3 bit immediate number to select any of the eight possible jump distances. Some of these had to be recalculated to make sure that the jumps were the predetermined values, otherwise the simulator would not step through its execution of the program. Since our instruction memory for each program is static, we were able to use branching with only 3 registers {R0, R1, R3}. Veering away from this would prove difficult and unnecessary since the programs already adjusted to this limited ISA. Comparing directly with R0, allows for pseudo-jump functionality in which the branch statement is always true therefore jumping the specified amount. We were able to jump as far as -22 and 21 and as short as 3.

Example:

Instruction:
beqR0, R3, equal # Label is +11 bytes from this instruction
All of the jumps are connected to a MUX
In this case we chose 101 as the select for 11
Machine Code:
(0)1111101 # First 4 bits are for beqR0 and register R3 (11)
Last 3 bits (101) are selecting 11 in the MUX

Instruction:
beqR0, R0, loop # Label is -12 bytes from this instruction
We chose 001 as the select for -12 in MUX design
Machine Code:
(1)1100001

4. Memory Model

4.1 Data Memory

- 16-bit double-byte addressable
- 128 memory 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]

Example:

Instruction:
ld r1, (r0) # r0 = 6 so r1 = mem[6]

Machine Code:
(0)0010100 # 001 is for ld instruction
01 is for r1(destination) and 00 for r0(address)

Instruction:
str r1, (r0) # r0 = 6 so mem[6] = contents of r1

Machine Code:
(0)0100100 # 010 is for str instruction
01 for r1 and 00 for r0(address)

Part B. Answers to Questions

1. What are the most significant advantages of your ISA (with regard to the two programs, hardware implementation, ease of programming, etc)? What are the main limitations? What are the main compromises that you have done to make things work, rather than perfecting everything?

Our advantage is that we keep everything relatively simple for hardware, we did not have to unnecessarily complicate the architecture of the ISA since we know the instructions are only for two programs. The versatility provided by the instructions is specific to the goals of the programs, e.g. score instruction, branch instruction, add instruction. The program could possibly handle other tasks as well because it is robust enough to load from any reasonable memory location, move PC far distances, use subtraction, addition, and initialization of registers these instructions form a subset of the fundamental operations useful in assembly programming. One of the compromises we made was in regards to multiplication, considering the tradeoff between using many add instructions or vastly complicating the hardware with a shift and subtraction operation for multiple by 6. We opted with many add instructions because it provided better versatility for both program 1 and program 2.

2. What have you done towards the goals of low DIC and HW simplification? What could have been done differently to better optimize for each of the two goals, if to start over?

We lowered DIC by using a score instruction to count the number of ones instead of using a nested loop to XOR and then to count the 1's by using a mask or shiting. Hardware was kept simple by avoiding the addition of many control signals to the control unit. If we were to start over we would build our next design based on the current one but look for places where we could add specialized instructions even though this would increase our hardware complexity it could have a significant impact on lowering the DIC making the trade off worth while.

3. Reflect on this project(1-3) experience:
a. What did you learn from this project? What was the best / worst thing about it?
b. What advice would you give to someone taking this project in a future semester?
c. How would you describe the value of this project experience in a job interview?

a. Having the experience of crafting an instruction set taught us how hardware and software must consider each other constantly during the design process. Analyzing the datapath unit helped solidify my understand of MIPS. The best thing about it was being able to work in a team to find solutions to tricky problems. A negative thing about it might be that doing three projects on seemingly similar subjects can be lack luster for some people.

b. I would advise someone undertaking this project to layout ideas for the ISA and spend a good deal of time thinking through the pros/cons. What I did was write down the instructions that were absolutely necessary once I coded those into my ISA I was able to add specialized instructions that were needed in order to complete the programs. Keeping in constant communication with your team because will ensure the project is moving in the right direction at all times.

c. This project provides a valuable example of our detailed understanding of hardware design such as the use of assembly language or the concepts of register transfer level and flow control. We also used Python to create a test fixture for our design which is an important part of the design process. Most importantly we learned how to work better as team which is a transferable skill for any job, position or career.

Part C. Simulation Results

Note: the complete resulting data memory files are included in the submission for verification.

1. Pattern A results

Python output:

```
[9, 17, 0, 0, 0, 0, 0, 0, -3967, -3966]
Registers init. to 0 r0=0 r1=0 r2=0 r3=0
Running Program 1...Program 1 finished!
DIC: 316
Registers r0=9 r1=9 r2=17 r3=11
mem [ 0 ]= 9
mem [ 1 ]= 17
mem [ 2 ]= 11 Result
mem [ 3 ]= 0
mem [ 4 ]= 0
mem [ 5 ]= 0
Running Program 2...Program 2 finished!
DIC: 1839
Registers r0=108 r1=108 r2=0 r3=8
mem [ 0 ]= 9
mem [ 1 ]= 17
mem [ 2 ]= 11 Result
mem [ 3 ]= 0
mem [ 4 ]= 10 Score
mem [ 5 ]= 7 Count
end
```

2. Pattern B results

Python output:

```
[267, 4099, 0, 21845, 0, 0, 0, 0, 2, 3]
Registers init. to 0 r0=0 r1=0 r2=0 r3=0
Running Program 1...Program 1 finished!
DIC: 9808
Registers r0=267 r1=267 r2=4099 r3=2415
mem [ 0 ]= 267
mem [ 1 ]= 4099
mem [ 2 ]= 2415 Result
mem [ 3 ]= 21845
mem [ 4 ]= 0
mem [ 5 ]= 0
Running Program 2...Program 2 finished!
DIC: 1888
Registers r0=108 r1=108 r2=21845 r3=9
mem [ 0 ]= 267
mem [ 1 ]= 4099
mem [ 2 ]= 2415 Result
mem [ 3 ]= 21845
mem [ 4 ]= 12 Score
mem [ 5 ]= 4 Count
end
```

3. Pattern C results
Python output:

```
[10, 20, 0, 1, 0, 0, 0, 0, -256, 21845]
Registers init. to 0 r0=0 r1=0 r2=0 r3=0
Running Program 1...Program 1 finished!
DIC: 415
Registers r0=10 r1=10 r2=20 r3=16
mem [ 0 ]= 10
mem [ 1 ]= 20
mem [ 2 ]= 16 Result
mem [ 3 ]= 1
mem [ 4 ]= 0
mem [ 5 ]= 0
Running Program 2...Program 2 finished!
DIC: 1934
Registers r0=108 r1=108 r2=1 r3=9
mem [ 0 ]= 10
mem [ 1 ]= 20
mem [ 2 ]= 16 Result
mem [ 3 ]= 1
mem [ 4 ]= 9 Score
mem [ 5 ]= 50 Count
end
```

4. Pattern D results
Python output:

```
[1007, 64, 0, -3856, 0, 0, 0, 0, 26214, -13108]
Registers init. to 0 r0=0 r1=0 r2=0 r3=0
Running Program 1...Program 1 finished!
DIC: 21208
Registers r0=1007 r1=1007 r2=64 r3=0
mem [ 0 ]= 1007
mem [ 1 ]= 64
mem [ 2 ]= 0 Result
mem [ 3 ]= -3856
mem [ 4 ]= 0
mem [ 5 ]= 0
Running Program 2...Program 2 finished!
DIC: 2025
Registers r0=108 r1=108 r2=-3856 r3=8
mem [ 0 ]= 1007
mem [ 1 ]= 64
mem [ 2 ]= 0 Result
mem [ 3 ]= -3856
mem [ 4 ]= 8 Score
mem [ 5 ]= 100 Count
end
```

Execution Process

Loading a program(I-Mem, D-Mem) from our text file

```
def load_program(cpu, instr_file_name, memory_file_name):
    instr_file = open(instr_file_name, "r")
    memory_file = open(memory_file_name, "r")

    for line in instr_file:
        line = line.strip()
        if len(line) < 1 or line.startswith("#") or line.startswith("U"):
            continue
        line = line.split(" ")
        cpu.instructions.append(line[0])

    for line in memory_file:
        line = line.strip()
        if len(line) < 1 or line.startswith("#"):
            continue
        number = int(line,2)
        if line.startswith("1"):
            number = (0xFFFF - int(line,2) + 1) * -1
        cpu.memory.append(number)

    for i in range(128-len(cpu.memory)):
        cpu.memory.append(0)

    instr_file.close()
    memory_file.close()

    return cpu
```

We use a cpu class to hold useful information and update the state of the running program:

```
class CPU:
    PC = 0 # Program Counter
    DIC = 0 # Insturction Counter
    R = [0] * 4 # Register Values
    instructions = [] # instructions in array
    memory = [] # memory in array
```

Program will determine the correct instruction to perform, do some calculation, update the appropriate registers, data memory, and advance PC to the next or branch to an immediate number.

```
        elif instr[1:4] == "000":
            # Init instruction
            Rx = registers[instr[3:5]] # Bit 4 picks whether or not this should be
            cpu.R[Rx] = int(instr[5:8], 2) # Cast the imm value into base ten
            cpu.PC = cpu.PC + 1
            cpu.DIC = cpu.DIC + 1
        elif instr[1:3] == "11":
            # Branch Equal R0
            Rx = registers[instr[3:5]]
            imm = imm_mux[instr[5:8]]
            if cpu.R[Rx] == cpu.R[0]:
                cpu.PC = cpu.PC + imm
            else:
                cpu.PC = cpu.PC + 1
            cpu.DIC = cpu.DIC + 1
        elif instr[1:4] == "100":
            # Add immediate
            Rx = registers[instr[4:6]]
            imm = registers[instr[6:8]] # imm value is [0,3] in this case encoded t
            cpu.R[Rx] = cpu.R[Rx] + imm
            cpu.PC = cpu.PC + 1
            cpu.DIC = cpu.DIC + 1
        elif instr[1:4] == "101":
            # Set less than, if so R0 = 1
            Rx = registers[instr[4:6]]
            Ry = registers[instr[6:8]]
            if cpu.R[Rx] < cpu.R[Ry]:
                cpu.R[0] = 1
            else:
                cpu.R[0] = 0
            cpu.PC = cpu.PC + 1
            cpu.DIC = cpu.DIC + 1
```

Running program is called in main to execute program 1, we print the register values (initialized to 0) and initial data memory. After run_program returns, DIC is accessed from cpu and printed, along with updated register values. Resulting data memory [0:5] is printed in base 10 for readability. Complete resulting data memory is written into a file to include in our submission.

```
if __name__ == "__main__":
    cpu1 = CPU()
    # load program 1
    cpu1 = load_program(cpu1, "prog1_group_4_p1_bin.txt", "patternB")
    print(cpu1.memory[0:10])
    print("Registers init. to 0 r0=" + str(cpu1.R[0]) + " r1=" + str(cpu1.R[1]) + " r2=" + str(cpu1.R[2]) + " r3="

    print("Running Program 1...", end='')
    cpu1 = run_program(cpu1)
    print("Program 1 finished!")
    print("DIC: " + str(cpu1.DIC))
    print("Registers r0=" + str(cpu1.R[0]) + " r1=" + str(cpu1.R[1]) + " r2=" + str(cpu1.R[2]) + " r3=" + str(cpu1.
    for m in range(0,6):
        if m == 2:
            print("mem", "[", m, "]=", cpu1.memory[m], "Result")
        else:
            print("mem", "[", m, "]=", cpu1.memory[m])

    # send resulting data memory to file
    output_file = open("p3_group_4_dmem_B.txt", "w")
    for i in range(0,108):
        bin1 = format(cpu1.memory[i], "016b")
        output_file.write(bin1 + "\n")
```

Subsequently program 2 is loaded. Similar it uses the same data memory that resulted from program 1 and runs through its execution. Resulting memory is outputted along with labels for result, score, and count. Final data memory is written to the file for our submission.

```
# load program 2
cpu2 = load_program(cpu1, "prog1_group_4_p2_bin.txt", "p3_group_4_dmem_B.txt")
# reset DIC
cpu2.DIC = 0
print("Running Program 2...", end='')
cpu2 = run_program(cpu2)
print("Program 2 finished!")
print("DIC: " + str(cpu2.DIC))
print("Registers r0=" + str(cpu1.R[0]) + " r1=" + str(cpu1.R[1]) + " r2=" + str(cpu1.R[2]) + " r3=" + str
for n in range(0,6):
    if n == 2:
        print("mem", "[", n, "]=", cpu1.memory[n], "Result")
    elif n == 4:
        print("mem", "[", n, "]=", cpu1.memory[n], "Score")
    elif n == 5:
        print("mem", "[", n, "]=", cpu1.memory[n], "Count")
    else:
        print("mem", "[", n, "]=", cpu1.memory[n])

# send resulting data memory to file
f = open("p3_group_4_dmem_B.txt", 'w') # overwrite previous file
for i in range(0,108):
    bin2 = format(cpu1.memory[i], "016b")
    f.write(bin2 + "\n")

print("end")
```

ISA Package

Program 1:

Assembly: Machine Code:

```
addi r3, 1      (0)1001101          #  register  that  will  be
exponentiated i.e. 6^p
init r1, 6      (1)0001110      # keep an incrementer in memory
str r0, r1     (0)0100001       # mem[6] = 0 initially

loop:
init r0, 0     (0)0000000       # r0 = 0
ld r0, r0      (1)0010000       # r0 = mem[0] = P
beqR0 r1, finish(0)1101111       #  if  P  (r0)  ==  incrementer
(r1), finish
                      # else do exponentiation
addR  r3       (0)0110011       # r2 = r3 + r3 = 1 + 1
addR2 r3       (1)0111011       # r2 = r2 + r3 = 1 + 2 = 3
addR3 r2       (1)0111110       # r3 = r2 + r2 = 3 + 3 = 6
init r0, 7     (1)0000111       # r0 = 7
str r0, r3     (1)0101100       # mem[7] = r3

mod:
init r1, 1     (0)0001001       # r1 = 1
ld r2, r1      (1)0011001       # r2 = mem[1] = Q
sltR0 r3, r2   (1)1011110       # if r3 < r2 then r0 = 1
beqR0 r1, done(0)1101001       # if r3 < 0, branch out
subR3 r2       (0)0110110       # r3 = r3 - r2
beqR0 r0, mod  (1)1100010       # otherwise keep subtracting

done:
addi r1, 1     (1)1000101       # r1 = 1 + 1 = 2
str r3, r1     (0)0101101       # mem[2] <= r3
init r0, 6     (0)0000110       # r0 = 6
ld r1, r0      (1)0010000       # r1 <= mem[6] = incrementer
addi r1, 1     (1)1000101       # r1 ++ (incrementer++)
str r1, r0     (0)0100100           # mem[6] <= incrementer
init r0, 2     (0)1000001       # r0 = 2
ld r3, r0      (1)0011100       # r3 = mem[2]
beqR0 r0, loop    (0)1100011        # jump back to loop

finish:
haltPC         (1)11 10 000     # PC not changing
```

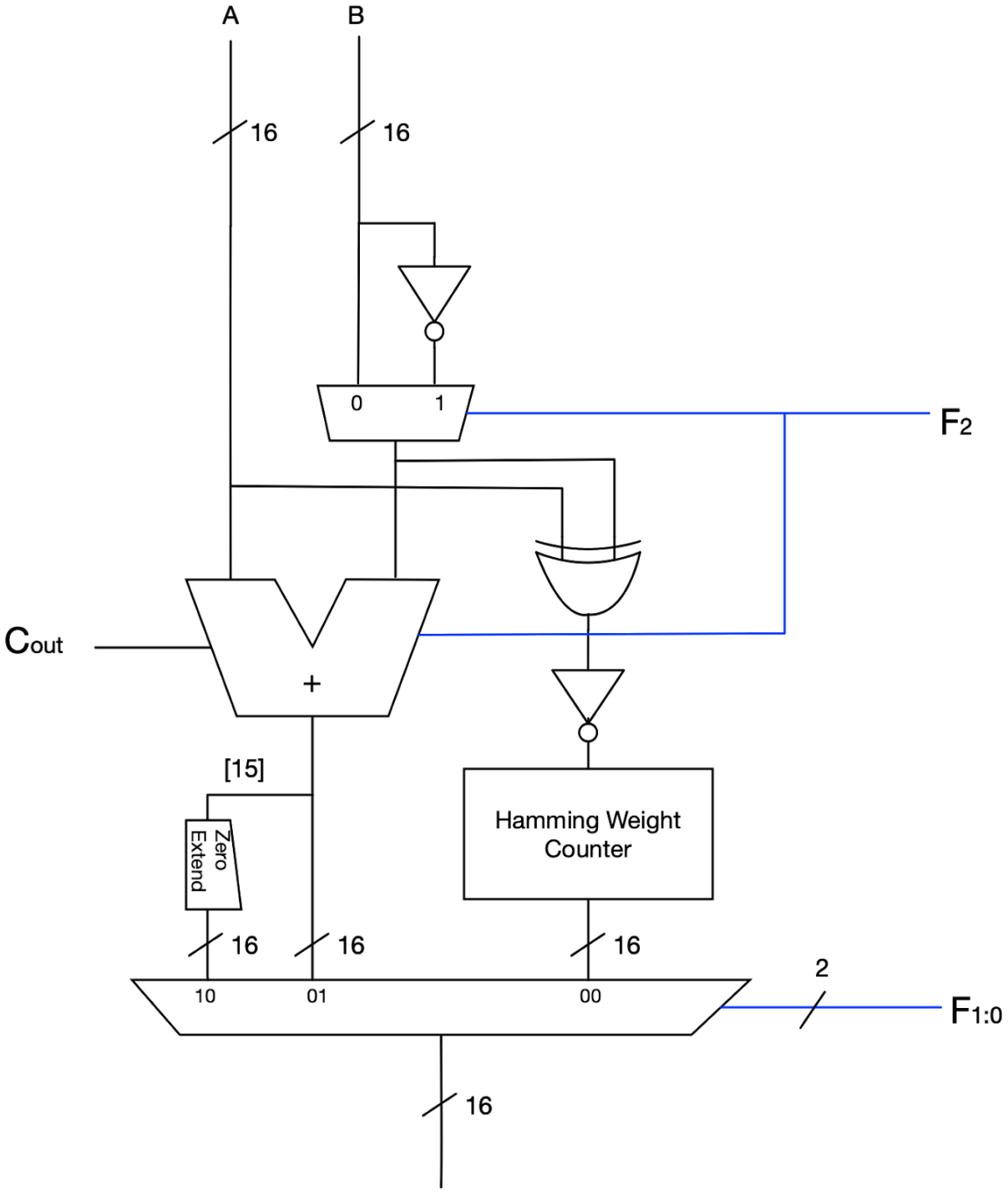
Program 2:

```
Assembly:              Machine Code:
init r1, 7             (0)0001111     # r1 = 7
addi r1, 3             (0)1000111     # r1 = 7 + 3 = 10
addR r1                (1)0110001     # r2 = 10 + 10
addR r2                (1)0110010     # r2 = r2 + r2 = 40
addR r2                (1)0110010     # r2 = r2 + r2 = 80
addR2 r1               (0)0111001     # r2 = r2 + 10 = 90
addR2 r1               (0)0111001     # r2 = r2 + 10 = 100
init r1, 7             (0)0001111     # r1 = 7
addi r1, 1             (1)1001001     # r1 = 8
addR2 r1               (0)0111001     # r2 = r2 + 8 = 108
init r1, 7             (0)0001111     # r1 = 7
str r2, r1             (1)0101001     # mem[r1] = mem[7] = 108

init r1, 3             (1)0001011     # r1 = 3
ld r2, (r1)            (1)0011001     # r2 = mem[3] = T
init r1, 7             (0)0001111     # r1 = 7
addi r1, 1             (0)1000101     # r1 = 7 + 1 = 8
init r0, 6             (0)0000110     # mem[6] will be our ptr
str r1, (r0)           (0)0100100     # mem[6] <=  8
loop:
ld r3, (r1)            (0)0011101           #  r3  =  mem[8]  =
Pattern_Arr
scr r3, r2             (0)1110111          # find score r3 and
str in r3
init r1, 4             (0)0001100     # r1 = 4
ld r0, (r1)            (0)0010001     # r0 = mem[4] = S (highest
score)
beqR0, r3, equal       (0)1111110          # if new scr == S, go
to equal
init r1, 1             (0)0001001     # redundant instr to allow
jump same imm
init r1, 1             (0)0001001     # redundant
sltR0 r0, r3           (0)1010011     # if new scr > S, r0 = 1
init r1, 1             (0)0001001     # r1 = 1
beqR0 r1 new           (1)1101100     # go to new if new scr > S
                                   # else, we go to next pattern
jump3:
init r0, 6             (0)0000110          # r0 = 6
ld r1, (r0)            (0)0010100     # r1 = mem[6] (array ptr)
addi r1, 1             (1)1000101     # array ptr++ / go to next
entry
str r1, (r0)           (0)0100100     # mem[6] <= array ptr
init r0, 7             (1)0000111
ld r0, r0              (1)0010000     # r0 = mem[7] = 108
beqR0 r1, done         (1)1101100     # if array ptr=108, done!
beqR0 r0, loop         (1)1100101     # else, go to loop
```

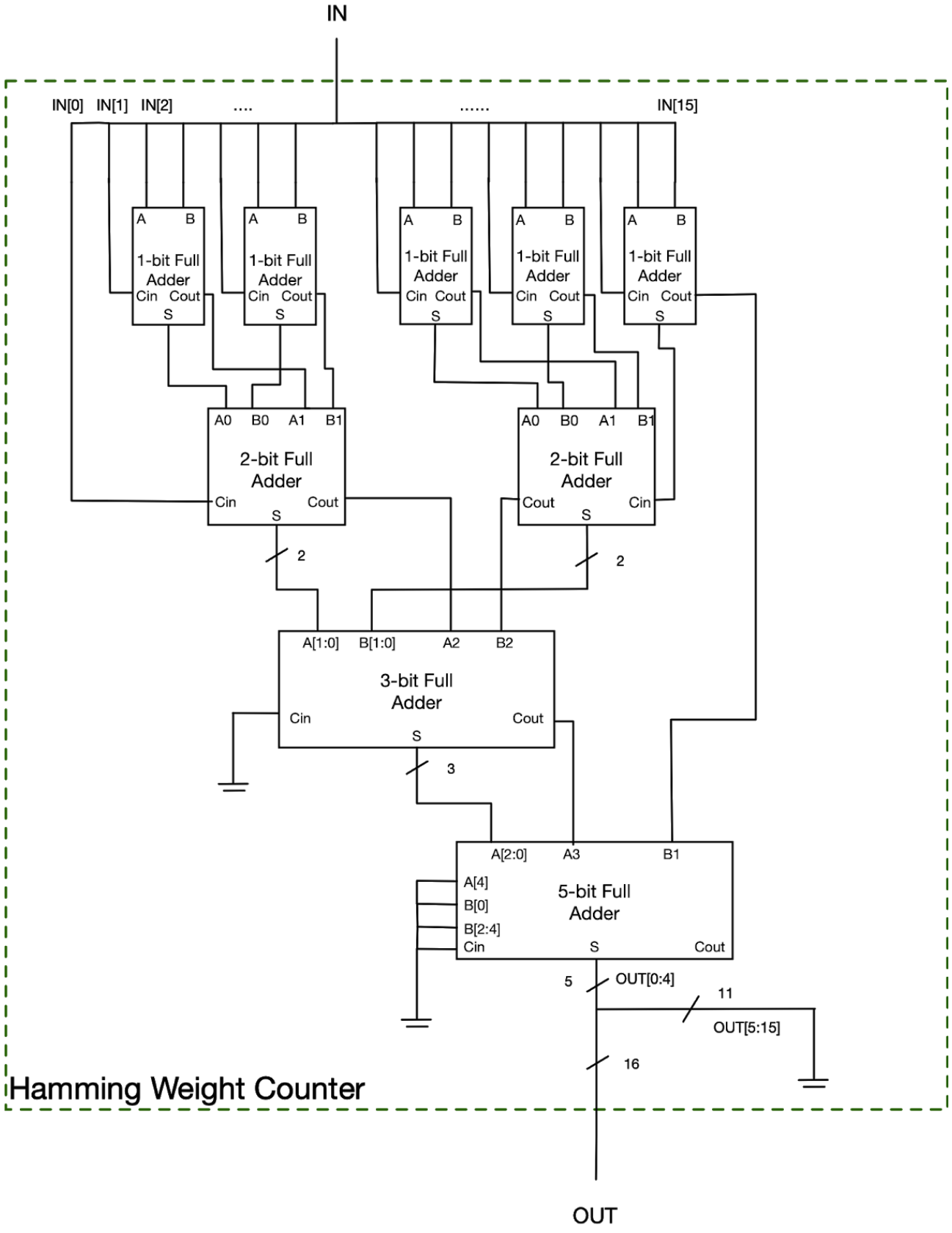
```
jump2:
init r1, 3           (1)0001011     # redundant instr to allow
make jump same imm
beqR0 r0, jump3      (1)1100111     # intermediate jump

equal:
init r0, 5           (0)0000101     # r0 = 5
ld r1, (r0)          (0)0010100     # r0 = mem[5] = C
addi r1, 1           (1)1000101     # r1++ (count++)
str r1, (r0)              (0)0100100     # mem[5] = r1
jump1:
beqR0, r0, jump2     (0)1100011     # intermediate jump

new:
init r1, 4           (0)0001100     # r1 = 4
str r3, (r1)              (0)0101101      # mem[4] <= r3 (new
score)
init r1, 5           (1)0001101     # r1 = 5
init r0, 1           (1)0000001     # r0 = 1
str r0, (r1)              (0)0100001      # mem[5] <= 1(reset
count)
beqR0, r0, jump1     (0)1100011     # intermediate jump

done:
init r0, 0           (0)0000000     # adjust jump distance
haltPC               (1)1110000     # pc not changing
```

Part D. Hardware Implementation

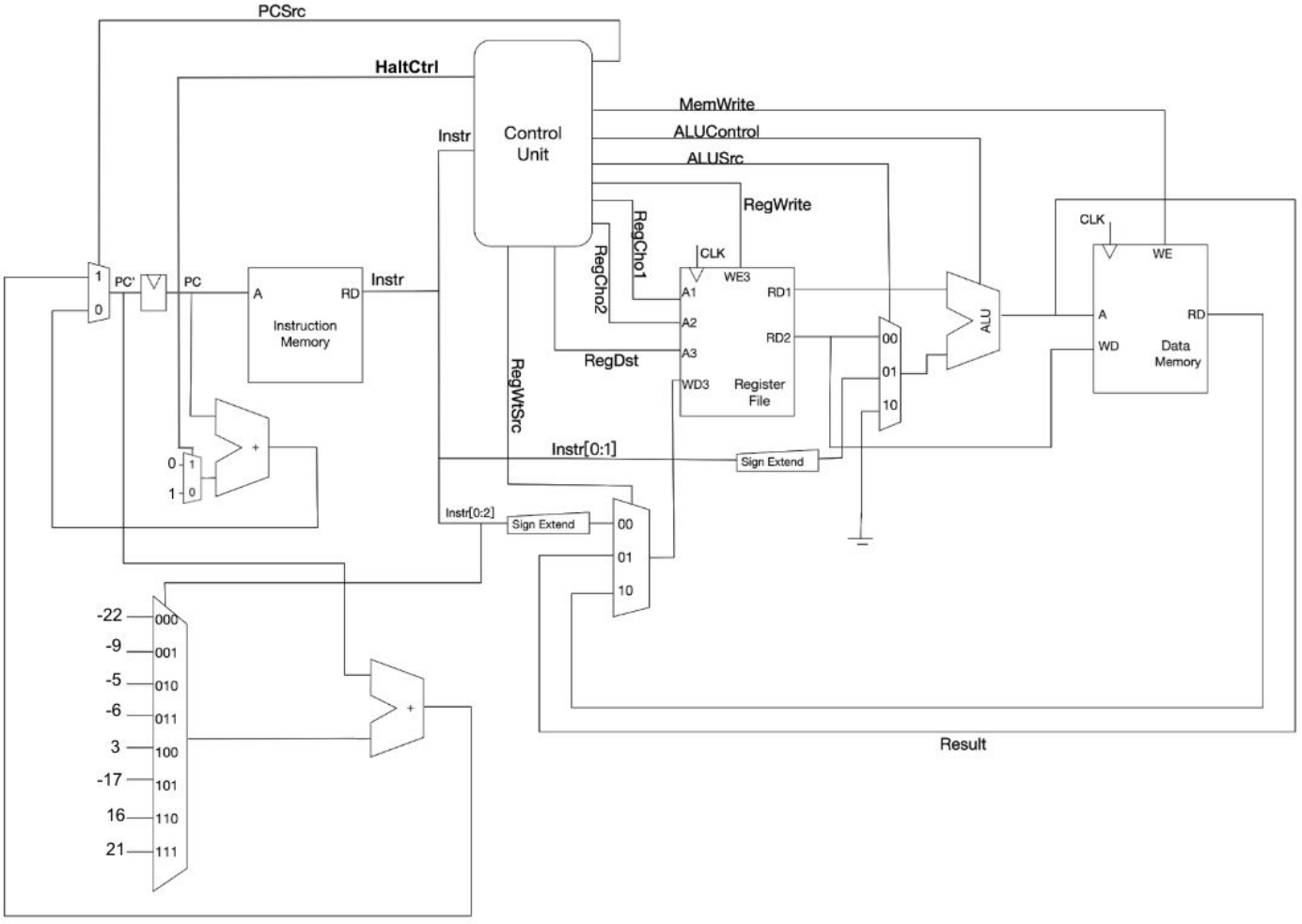
1. ALU schematic



Hamming Weight Counter



2. CPU Datapath



3. Control Logic

Truth Table for control unit:

Instr	Op	PCSrc	MemWrite	ALUControl	ALUSrc	RegWrite	RegCho1	RegCho2	RegDst	RegWtSrc	HaltCtrl
init	000 r iii	0	0	XXX	XX	1	XX	XX	0r	00	0
ld	001 rr ss	0	0	001	10	1	ss	XX	rr	10	0
str	010 rr ss	0	1	001	10	0	ss	rr	XX	XX	0
addR	01100 rr	0	0	001	00	1	rr	rr	10	01	0
addR2	01110 rr	0	0	001	00	1	10	rr	10	01	0
addR3	01111 rr	0	0	001	00	1	rr	rr	11	01	0
subR3	01101 rr	0	0	101	00	1	11	rr	11	01	0
addi	100 rr ii	0	0	001	01	1	rr	XX	rr	01	0
sltR0	101 rr ss	0	0	110	00	1	rr	ss	00	01	0
beqR0	11 rr iii	1	0	101	00	0	rr	00	XX	XX	0
scrR3R2	1110 111	0	0	000	00	1	10	11	11	01	0
haltPC	1110 000	0	0	XXX	XX	0	XX	XX	XX	XX	1