**Part A) ISA intro**

1. **Introduction: This should include the name of the architecture, overall philosophy, specific goals strived for and achieved.**

Name of architecture: ISA 2.0

This architecture is for an effective and practical ISA Design the goal is to achieve two programs; which were done in Project 1.

1. **Instruction list. Give all the instructions, their formats, opcodes, and an example.**

|  |  |  |  |
| --- | --- | --- | --- |
| Instruction | Format | Example | Opcode |
| init | init Rx, imm | Rx = imm | 1110 xx i |
| add | add Rx,Ry | Rx=Rx+Ry | 001 xx yy |
| load | load Rx, (Ry) | Rx=Mem[Ry] | 011 xx yy |
| store | store Rx, (Ry) | Mem[Ry]=Rx | 010 xx yy |
| jump | jump imm | imm [] | 110 iiii |
| slt | slt Rx, Ry | Rx=1 if Rx<Ry  Else Rx=0 | 101 xx yy |
| sub | sub Rx, Ry | Rx = Rx – Ry | 000 xx yy |
| bezR2 | bezR2 imm | If R2==0, PC+=imm else PC=PC+1 | 100 iiii |
| initR2 | initR2 | Initialize R2 to 5 | 1111000 |
| initR3 | initR3 | Initialize R3 to 5 | 1111001 |
| beqN | beqN | If R1=R0, branch to Next | 1111110 |
| beqInc | beqInc | If R0=R1, branch to Increment | 1111101 |
| stop | stop | Exit the program | 1111111 |

1. **Register design. How many registers are supported? Is there anything special about the registers?**

This architecture supports four registers (R0 to R3)

R2 is used for testing in bezR2 and it is initialized when initR2 is used; R3 is initialized when initR3 is used.

1. **Control flow (branches). What types of branches are supported? How are the target addresses calculated? What is the maximum branch distance supported? Give examples of your assembly branch instructions and their corresponding machine code.**

There are four branches that are supported; which are Jump, bezR2, beqN, beqInc. Jump has 4bits imm which makes it branch to a wide range. While beqN, beqInc are both fixed to branching into the same destination each time they are used. bezR2 branches to PC+imm iff R2=0; otherwise, it goes to PC+1. The maximum branch distance is supported with the Jump instruction which uses 4 imm so it can range from [1-23] in branching forward.

Examples:

|  |  |  |
| --- | --- | --- |
| Branch instruction | Example | Machine code |
| jump | PC = PC + 1 | 1100001 |
| bezR2 | If R2==0, PC+=0 else PC=PC+1 | 1000000 |
| beqN | If R1=R0, branch to Next | 1111110 |
| beqInc | If R0=R1, branch to Increment | 1111101 |

1. **Data memory addressing modes. What addressing modes are supported for data memory? How are the addresses calculated? Give examples of your assembly load / store instructions and their corresponding machine code.**

Addressing Modes supported are M(Ry) with Ry being a register; and the address is calculated such that M(0) is the address 1 of the data memory.

Examples:

|  |  |  |
| --- | --- | --- |
| Instruction | Example | Machine code |
| load | R0=Mem[R0] | 0110000 |
| store | Mem[R0]=R0 | 0100000 |

**Part B) Answers to questions**

1. **Comparing to the sample of “My\_straightforward\_ISA”, what are the unique features of your ISA? Explain why your ISA is better.**

In our ISA design, we are using 1-bit less so we have 7-bits and we have more instructions written within those 7-bits so this can help have less lines of coding.

1. **In what ways did you optimize for the two goals? If you optimized for anything additional, what and how?**

We have got high speed in SW by minimizing the instructions used in each program. And we have low cost HW by simplifying our HW design to make it simpler.

1. **What would you have done differently if you had 1 more bit for instructions? How about 1 fewer bit?**

With 1 more bit, we could have used more registers and therefore make coding part much easier for both programs; however, with 1 less bit, it would be much more difficult to have an efficient design.

1. **How did your team work together to accomplish this project? (Role of each team member, progress milestones, time spent individually and together?)**

Both of us worked together the same amount of time through every procedure.

Time spent is

1. **If you had a chance to restart this project afresh with 3 weeks’ time, how would your team have done differently?**

We would work on the hardware part to make the hardware more efficient.

**Part C) Software package**:

**1. Algorithms (in assembly code) of the two programs. Make sure your assembly format is either obvious or well described, and that the code is well commented. Extra credits: provide a convincing estimation:**

**i. on the dynamic instruction count for P1 (ME) with P = ~1000 and Q = ~500**

**ii. on the worst-case scenario of dynamic instruction count for P2(BMC).**

**Algorithm for prog 1:**

**.data**

**P: .word 11**

**R: .word -1 #R will be stored here**

**.text**

**addi $15,$15,0x2000**

**lw $8,0($15) #P <- M[0x2000]**

**addi $10,$0,0 #counter = 0**

**addi $12,$0,6 #b = 6**

**beq $8,$0,zero #if P = 0 branch**

**addi $10,$0,1 #counter = 1**

**beq $10,$8,one #if P = 1 branch**

**loop:**

**addi $11,$0,0 #a = 0**

**addu $11,$11,$12 #a = a+b**

**addu $11,$11,$12 #a = a+b**

**addu $11,$11,$12 #a = a+b**

**addu $11,$11,$12 #a = a+b**

**addu $11,$11,$12 #a = a+b**

**addu $11,$11,$12 #a = a+b**

**addi $10,$10,1 #counter++**

**addu $12,$11,$0 #b= a**

**beq $10,$8,next #when counter = p go to next**

**j loop**

**zero:**

**addi $11,$0,1 #a = 1**

**sw $11,4($15) #a -> M[0x2004]**

**j out**

**one:**

**addi $11,$0,6 #a = 6**

**sw $11,4($15) #a -> M[0x2004]**

**j out**

**next:**

**addi $13,$0,17 #c = 17**

**sub $11,$11,$13 #a = a-c**

**slt $14,$11,$13 #if a < c set a= 1 else a = 0**

**beq $14,$0,next**

**sw $11,4($15) #a -> M[0x2004]**

**j out**

**out:**

**j out**

**Algorithm for prog 2:**

**.data**

**T: .word 7**

**best\_matching\_score: .word -1 # best score = ? within [0,32]**

**best\_matching\_count: .word -1 # how many patterns achieve the best score?**

**Pattern\_Array: .word 0,1,2,3,4,5,6,7,8,9,10,11,13,14,15,16,17,18,19,20**

**.text**

**addi $8,$0,0x2000 #$8 = 0x2000 were T starts**

**addi $9,$0,0x200C #$9 = 0x200C were Pattern\_Array starts**

**addi $10,$0,0 #counter1 = 0 for best matching count**

**addi $11,$0,20 #counter2 = 20 to indicate**

**lw $12,0($8) #T <- M[0x2000]**

**loop:**

**lw $13,0($9) #Pattern\_Array <- M[0x200C]**

**beq $12,$13,increment #if T == Pattern\_Array[i] go to increment branch**

**addi $9,$9,4 #increments address by PC + 4 (2004,2008,200c,etc.)**

**subi $11,$11,1 #counter2--**

**beq $11,$0,result #if counter2 == 0 stores the best\_match\_count**

**j loop**

**increment:**

**addi $10,$10,1 #counter1++**

**addi $9,$9,4 #increments address by PC + 4 (2004,2008,200c,etc.)**

**subi $11,$11,1 #counter2--**

**j loop**

**result:**

**sw $10,8($8) #counter1 ->M[0x2008]**

**j exit**

**exit:**

**j exit**

**2. Machine code for each of the programs. We will not correct/grade the machine code. You will also not be able to verify whether your code works correctly or not in this project (without a simulator). Therefore, you have to rely on the help of the disassembler, strive to make your algorithm simple and easy to understand, as well as pursue the sw-hw “codesign” – this will avoid putting tremendous complexity at either the software or the hardware end.**

**Machine code for prog 1:**

**01110 00 1 #R0 = 1**

**1001 00 00 #R0 = 2**

**11110 01 1 # R1 = 1 (COUNTER)**

**01111000 # R2 = 5**

**1001 10 01 #R2 = 6**

**0010 10 00 #R2 ->Mem[2]**

**11110 11 0 #R3 = 0**

**0001 11 10 #R3 = R3 + R2**

**0001 11 10 #R3 = R3 + R2**

**0001 11 10 #R3 = R3 + R2**

**0001 11 10 #R3 = R3 + R2**

**0001 11 10 #R3 = R3 + R2**

**0001 11 10 #R3 = R3 + R2**

**01110 00 1 #R0 = 1**

**0001 01 00 #counter++ (R1 = R1 + R0)**

**1010 11 00 #R3 -> Mem[1]**

**1011 10 00 #R2 <- Mem[1] R2 = R3**

**11110 00 0 #R0 = 0**

**0011 00 00 # P = Mem[0] (R0 = Mem[0])**

**0100 0010 #if counter == p (r1 == r0) branch to next**

**1110 0010 #j loop**

**01110 11 1 #R3 = 1**

**01110 00 1 #R0 = 1**

**1001 00 00 #R0 = 2**

**1001 00 00 #R0 = 4**

**1001 00 00 #R0 = 8**

**1001 00 00 #R0 = 16**

**1001 11 00 #R3 = 17**

**1000 10 11 #R2 = R2 - R3**

**1101 10 11 # if R2 < R3 R2 = 1 else R2 = 0**

**0100 0111 #if R2 = 0 branch to next else PC = PC + 1**

**0010 10 00 #R2 ->Mem[2]**

**11111111 #STOP**

**Machine code for prog 2:**

**01110 00 1 #R0 = 1**

**11110 01 1 #R1 = 1**

**0001 00 01 #R0 = 2**

**0001 00 01 #R0 = 3**

**0011 00 00 #R0 <- Mem[3] (T)**

**01111000 #R2 = 5**

**1001 10 10 #R2 = 10**

**1001 10 10 #R2 = 20 counter**

**1001 11 11 #R1 = 2**

**1001 11 11 #R1 = 4**

**1001 11 11 #R1 = 8**

**0011 11 11 #R1 <- Mem[8] (PA)**

**01111101 #If R0 == R1 (T == PA) branch to increment**

**01110 11 1 #R3 = 1**

**0001 01 11 #R1 = 9**

**1000 10 11 #R2 = R2 - R3 (counter--)**

**0100 0111 #if R2 == 0 branch to result else PC = PC + 1**

**0110 1010 #j loop**

**11110 00 0 #R0 = 0**

**1001 00 11 #R0 = 1**

**1001 01 11 #R1 = 9**

**1000 10 11 #R2 = R2 - R3 (counter--)**

**1110 1011 #j loop**

**11111001 #R3 = 5**

**1010 00 11 #R0 -> Mem[5]**

**11111111 #STOP**

**3. Output of your Python disassembler for each program. This should be a line-by-line explanation of the machine code, what is done by each line of code.**

**Output of prog 1:**

**Disassembling Program 1...**

**init R0, 0 // R0 = 0**

**add R0, R0 // R0 = R0 + R0**

**init R0, 1 // R0 = 1**

**initR2 //initialize R2 to 5**

**add R1, R0 // R1 = R1 + R0**

**store R1, (R0) // M[R0] <- R1**

**init R1, 1 // R1 = 1**

**add R1, R1 // R1 = R1 + R1**

**add R1, R1 // R1 = R1 + R1**

**add R1, R1 // R1 = R1 + R1**

**add R1, R1 // R1 = R1 + R1**

**add R1, R1 // R1 = R1 + R1**

**add R1, R1 // R1 = R1 + R1**

**init R0, 0 // R0 = 0**

**add R0, R1 // R0 = R0 + R1**

**store R1, (R1) // M[R1] <- R1**

**load R1, (R0) // R1 <- M[R0]**

**init R0, 0 // R0 = 0**

**load R0, (R0) // R0 <- M[R0]**

**bezR2, 1 // if R2 ==0: PC = PC + imm , else: PC++**

**jump 1 // PC = PC + imm**

**init R1, 1 // R1 = 1**

**init R0, 0 // R0 = 0**

**add R0, R0 // R0 = R0 + R0**

**add R0, R0 // R0 = R0 + R0**

**add R0, R0 // R0 = R0 + R0**

**add R0, R0 // R0 = R0 + R0**

**add R1, R1 // R1 = R1 + R1**

**sub R1, R0 // R1 = R1 - R0**

**slt R1, R0 // if R1 < R0, R1 = 1**

**bezR2, 3 // if R2 ==0: PC = PC + imm , else: PC++**

**store R1, (R0) // M[R0] <- R1**

**STOP**

**Output of prog 2:**

**Disassembling Program 2...**

**init R0, 0 // R0 = 0**

**init R0, 1 // R0 = 1**

**add R0, R0 // R0 = R0 + R0**

**add R0, R0 // R0 = R0 + R0**

**load R0, (R0) // R0 <- M[R0]**

**initR2 //initialize R2 to 5**

**add R1, R0 // R1 = R1 + R0**

**add R1, R0 // R1 = R1 + R0**

**add R1, R1 // R1 = R1 + R1**

**add R1, R1 // R1 = R1 + R1**

**add R1, R1 // R1 = R1 + R1**

**load R1, (R1) // R1 <- M[R1]**

**beqInc // if R0 == R1 branch to Increment**

**init R1, 1 // R1 = 1**

**add R0, R1 // R0 = R0 + R1**

**sub R1, R0 // R1 = R1 - R0**

**bezR2, 3 // if R2 ==0: PC = PC + imm , else: PC++**

**jump 5 // PC = PC + imm**

**init R0, 0 // R0 = 0**

**add R0, R0 // R0 = R0 + R0**

**add R0, R1 // R0 = R0 + R1**

**sub R1, R0 // R1 = R1 - R0**

**jump 5 // PC = PC + imm**

**initR3 //initialize R3 to 5**

**store R0, (R0) // M[R0] <- R0**

**STOP**

**4. Python code for your ISA’s disassembler.**

input\_file = open("machine\_code\_prog1.txt","r")

output\_file = open("prog1\_asm.txt","w")

output\_file.write("Disassembling from Program 1...\n \n")

for line in input\_file:

if (line[1:8] == "1111000"):

output\_file.write("initR2 //initialize R2 to 5\n")

elif (line[1:8] == "1111001"):

output\_file.write("initR3 //initialize R3 to 5\n")

elif (line[1:8] == "1111101"):

output\_file.write("beqInc // if R0 == R1 branch to Increment\n")

elif (line[1:8] == "1111110"):

output\_file.write("beqN //if R1 == R0 branch to Next\n")

elif (line[1:8] == "1111111"):

output\_file.write("STOP\n")

elif (line[1:5] == "1110"):

register = str(int(line[5:7],2))

immediate = str(int(line[7],2))

output\_file.write("init R" + register + ", " + immediate)

output\_file.write(" // R" + register + " = " + immediate + "\n")

elif (line[1:4] == "101"):

register1 = str(int(line[4:6],2))

register2 = str(int(line[6:8],2))

output\_file.write("slt R" + register1 + ", R" + register2)

output\_file.write(" // if R" + register1 + " < R" + register2 + ", R1 = 1 \n")

elif (line[1:4] == "010"):

register1 = str(int(line[4:6],2))

register2 = str(int(line[6:8],2))

output\_file.write("store R" + register1 + ", (R" + register2 + ")")

output\_file.write(" // M[R" + register2 + "] <- R" + register1 + "\n")

elif (line[1:4] == "011"):

register1 = str(int(line[4:6],2))

register2 = str(int(line[6:8],2))

output\_file.write("load R" + register1 + ", (R" + register2 + ")")

output\_file.write(" // R" + register1 + " <- M[R" + register2 + "] \n")

elif (line[1:4] == "000"):

register1 = str(int(line[4:6],2))

register2 = str(int(line[6:8],2))

output\_file.write("sub R" + register1 + ", R" + register2)

output\_file.write(" // R" + register1 + " = R" + register1 + " - R" + register2 + "\n")

elif (line[1:4] == "001"):

register1 = str(int(line[4:6], 2))

register2 = str(int(line[6:8], 2))

output\_file.write("add R" + register1 + ", R" + register2)

output\_file.write(" // R" + register1 + " = R" + register1 + " + R" + register2 + "\n")

elif (line[1:4] == "110"):

imm = str(int(line[4:8], 2))

output\_file.write("jump " + imm)

output\_file.write(" // PC = PC + imm \n")

elif (line[1:4] == "100"):

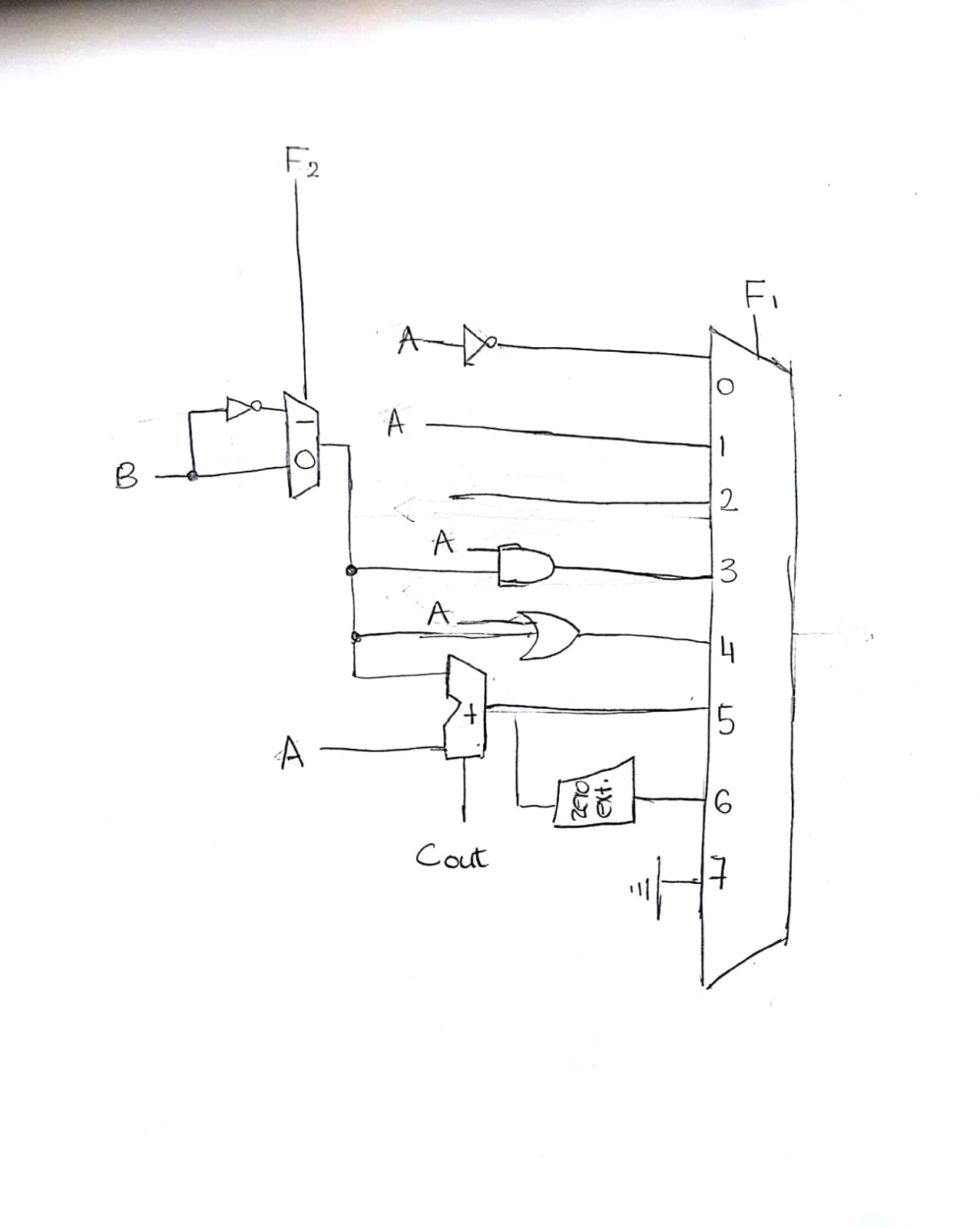
imm = str(int(line[4:8], 2))

output\_file.write("bezR2, " + imm)

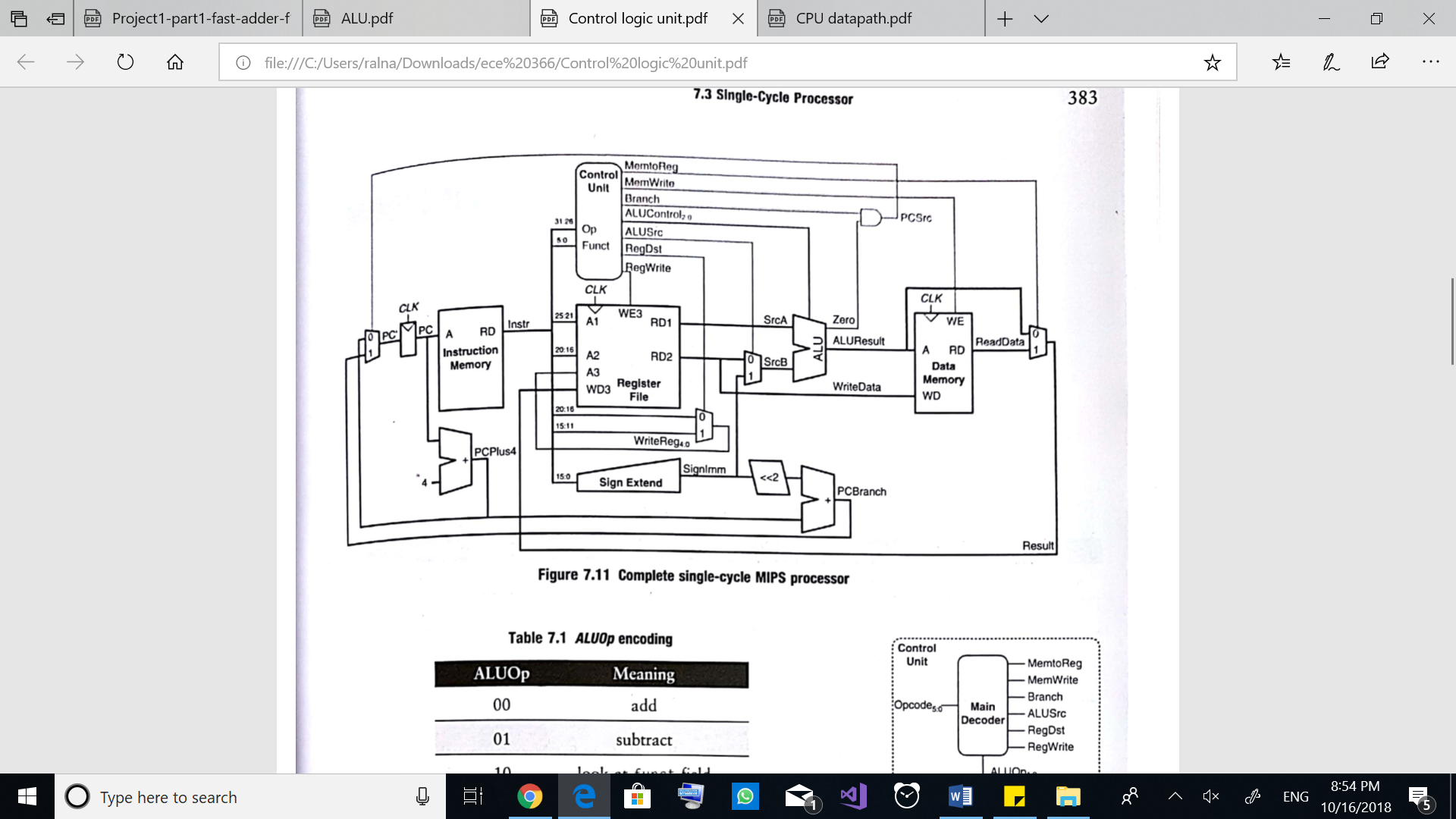
output\_file.write(" // if R2 ==0: PC = PC + imm , else: PC++ \n")

**Part D) Hardware implementation:**

**1. ALU schematic. A hierarchical sketch of your Arithmetic Logic Unit which implements whatever computation that your ISA instructions use (See textbook ch 5.2.4).**

****

**2. CPU Datapath design. A schematic including your register file, ALU, PC logic, and memory components (see textbook ch 7.3.1).**



1

1

**3. Control logic design. Decoder truth-table indicating how each control signal is generated from an instruction (see textbook ch 7.3.2).**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Opcode | Instruction | MemtoReg | MemWrite | Branch | ALUControl | ALUSrc | RegDst | RegWrite |
| 1110 | init | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| 001 | add | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 011 | load | 1 | 0 | 0 | 1 | 0 | 1 | 1 |
| 010 | store | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 110 | jump | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| 101 | slt | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 000 | sub | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 100 | bezR2 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| 1111000 | initR2 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| 1111001 | initR3 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| 1111110 | beqN | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| 1111101 | beqInc | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| 1111111 | stop | 0 | 0 | 0 | 0 | 0 | 1 | 0 |