**ECE 366 Group 17 Project 3**

Part A) Our ISA is called "Ultimately Specific". It is designed based on the number of different instructions we have. We have a total of 43 different instructions represented by their decimal value from 0-42.The philosophy behind this idea is create the simplest and easiest ISA creatable to our knowledge. The only way to decipher the instruction is to find its decimal value and figure it out from the table. We have a total of 128 possible different instructions (2^7) so reusing instructions within each program is encouraged. This design is based off our previous project ISA that encouraged as little code lines as possible. This would increase efficiency within the code and decrease computation time. The cost of such design is the complexity of our hardware implementation. For now, we will not arrange instruction alphabetically to further increase the simplicity of the machine code and Python. In other words, as the PC instruction value increases, so do the Decimal/Binary values unless branching to a previous instruction. The positive side is that we use only 6 bits for both programs (limited with static Q = 17). ISA is without the parity bit but our machine code will include parity bit.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Dec. Val | | Binary/Machine code | | Instr. | | Description | |
| 0 | | 0000000 | | lw $7, P | | # $7 = P | |
| 1 | | 0000001 | | addi $9, $0, 1 | | # $9 = 1 | |
| 2 | | 0000010 | | add $10, $0, $0 | | # $10 = 0 = counter N = exponent | |
| 3 | | 0000011 | | andi $10, $7, 1 | | # $10 = $7 and 1 (0/1) | |
| 4 | | 0000100 | | srl $7, $7, 1 | | # $7 >> 1 | |
| 5 | | 0000101 | | beq $10, $0, Mult2 | | # if zero, go to Mult2 | |
| 6 | | 0000110 | | add $10, $9, $9 | | # temp $10 = $9 + $9 = $ 10 = 2 | |
| 7 | | 0000111 | | add $9, $9, $9 | | # $9 = $9 + $9 = $9 = 2 | |
| 8 | | 0001000 | | add $9, $10, $9 | | # $9 = $10 + $9 = $9 = 6 | |
| 9 | | 0001001 | | beq $10, $0, Mult4 | | # if zero, go to Mult2, check to multiply next bit by 4, else multiply by 2 | |
| 10 | | 0001010 | | beq $10, $0, Mult16 | | # if zero, go to Mult2, check to multiply next bit by 16, else Mod | |
| 11 | | 0001011 | | beq $10, $0, Mod | | # if zero, go to Mult2, check to multiply next bit by 4, else multiply by 2 | |
| 12 | | 0001100 | | slti $10, $9, 17 | | # if $9 < $8, $10 = 1, else $10 = 0, these next 3 lines will subtract by 17 until the is only the remainder | |
| 13 | | 0001101 | | bne $10, $0, End | | # if($10 =1) Save2, else cont | |
| 14 | | 0001110 | | subi $9, $9, 17 | | # $9 = $9 -17 = 36 -17 = 19 | |
| 15 | | 0001111 | | j Mod | | # Will loop back to Mod to get remainder | |
| 16 | | 0010000 | | sw $9, R | | #Stores the remainder value into R | |
| 17 | | 0010001 | | addi $5 $0, 0 | | # Counter i | |
| 18 | | 0010010 | | addi $7 $0, 0 | | # Counter J | |
| 19 | | 0010011 | | addi $6 $0, 400 | | # stop value of array | |
| 20 | | 0010100 | | beq $5, $6, Exit | | # if $5 = 400, Exit program | |
| 21 | | 0010101 | | lw $1, 0xC($0) | | # $1 = T = Value to compare to / first Anded value | |
| 22 | | 0010110 | | lw $2, 0x2020($5) | | # $2 = mem(0x2020 + $5) | |
| 23 | | 0010111 | | addi $5, $5, 4 | | # Increment $5 for next array index | |
| 24 | | 0011000 | | and $1, $1, $2 | | # $1 = $1 and $2 = T and Array(i) = Anded value/rid of T to save on registers | |
| 25 | | 0011001 | | addi $8 $0, 16 | | # stop value for bit # | |
| 26 | | 0011010 | | beq $7, $8, Save | | # Once $7 equals 16, go to next array index | |
| 27 | | 0011011 | | addi $7, $7, 1 | | # Increment $7 by 1 | |
| 28 | | 0011100 | | andi $2, $1, 1 | | # $2 = $1 and 1 = Anded value and 1, $2 is temp value | |
| 29 | | 0011101 | | add $3, $3, $2 | | # counts # of 1s in $1 (# of matching bits) | |
| 30 | | 0011110 | | srl $1, $1, 1 | | # $1 = $1 >> 1, reduces it by 1 bit till = 0 | |
| 31 | | 0011111 | | j Compare | | # now Compare with old values | |
| 32 | | 0100000 | | lw $1, 0x2010($0) | | # reuse $1 = S | |
| 33 | | 0100001 | | lw $2, 0x2014($0) | | # reuse $2 = C | |
| 34 | | 0100010 | | slt $6, $1, $3 | | # $6 = 1 if( $1(S) < $3), else $6 = 0 | |
| 35 | | 0100011 | | beq $6, $0, Check1 | | # if (s>= $3 then $6 = 0) => if ($6 = 0) go to check 1, else continue | |
| 36 | | 0100100 | | add $1, $0, $3 | | # $1 = $3, $1 will now take highest value | |
| 37 | | 0100101 | | addi $2, $0, 1 | | # $2(T) will be set to one since its first highest value | |
| 38 | | 0100110 | | sw $1, 0x2010($0) | | # mem(0x2010) = $1 = S | |
| 39 | | 0100111 | | sw $2, 0x2014($0) | | # mem(0x2014) = $2 = T | |
| 40 | | 0101000 | | j Start | | # start over/chech next array | |
| 41 | | 0101001 | | beq $1, $3, Check2 | | # if ($1 = $3) go to check 2, else continue | |
| 42 | | 0101010 | | addi $2, $2, 1 | | # Increment 2(T) by 1 since matching count was found | |

Part B) 1. The most significant advantage of our ISA is that it is extremely simplistic and easy to program new set of instructions the individual wants to add. The biggest limitation in our ISA is that a certain person cannot add more than 127 set of instructions as this is the limit. The main compromises we have done are reducing the amount of registers used to cut back on the number of instructions needed.

2. We have significantly shortened the amount of computation time need to run however we increased the complexity of the hardware as each instruction had its own machine code. A different approach we of taken is to simplify the hardware and assigning a set of instructions with set registers for each line of machine code. However I would assume that this would increase the computation time of the program.

3. A) I learned a whole new way of creating an extremely simple ISA that is efficient to its certain limitations yet very useful. The best part was being able to come up with less and less instructions as we continued to simplify the instructions needed to run program one and two. The worst part was trying to figure out how to create the simulation in python as we needed to conduct some trial and error to receive the correct results.

B) Some advice for a student taking this class in a future semester is to continuously work on the project day to day and set goals for oneself as to not neglect the deadline day and keeping up with the work of the project.

C) The value of this project is immense it allows us to communicate and work with individuals from different character and knowledge backgrounds, each bringing their own idea to the table. This allows for one to learn from each other as well as showing what it takes to successfully work as a team for a multilayered project.

Part C)

Part D)

1. Algorithms

Assembly for Program 1

.data

P: .word 15

Q: .word 17

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

.text

lw $7, P

lw $8, Q

addi $9, $0, 1 # $9 = 1

add $10, $0, $0 # $10 = 0 = counter N = exponent

Base6: andi $10, $7, 1 # $10 = $7 and 1 (0/1)

srl $7, $7, 1 # $7 >> 1

beq $10, $0, Mult2 # if zero, go to Mult2

add $10, $9, $9 # temp $10 = $9 + $9 = $ 10 = 2

add $9, $9, $9 # $9 = $9 + $9 = $9 = 2

add $9, $9, $9 # $9 = $9 + $9 = $9 = 4

add $9, $10, $9 # $9 = $10 + $9 = $9 = 6

Mult2: andi $10, $7, 1 # $10 = $7 and 1 (0/1)

srl $7, $7, 1 # $7 >> 1

beq $10, $0, Mult4 # if zero, go to Mult4

add $9, $9, $9 # $9 = $9 + $9 = 2x$9 10

Mult4: andi $10, $7, 1 # $10 = $7 and 1 (0/1)

srl $7, $7, 1 # $7 >> 1

beq $10, $0, Mult16 # if zero, go to Mult16

add $9, $9, $9 # $9 = $9 + $9 = 2x$9

add $9, $9, $9 # $9 = $9 + $9 = 4x$9

Mult16: andi $10, $7, 1 # $10 = $7 and 1 (0/1)

srl $7, $7, 1 # $7 >> 1

beq $10, $0, Mod # if zero, go Mod

add $9, $9, $9 # $9 = $9 + $9

add $9, $9, $9 # $9 = $9 + $9

add $9, $9, $9 # $9 = $9 + $9

add $9, $9, $9 # $9 = $9 + $9

Mod: slt $10, $9, $8 # if $9 < $8, $10 = 1, else $10 = 0

bne $10, $0, End # if($10 =1) Save2, else cont

sub $9, $9, $8 # $16 = $16 -17 = 36 -17 = 19

j Mod

End: sw $9, R

Assembly for Program 2

addi $5 $0, 0 # Counter for Array

addi $7 $0, 0 # Counter for bit #

Start: addi $6 $0, 400 # stop value of array

beq $5, $6, Exit # if $5 = 400, Exit program

lw $1, 0xC($0) # $1 = T = Value to compare to / first Anded value

lw $2, 0x2020($5) # $2 = mem(0x2020 + $5)

addi $5, $5, 4 # Increment $5 for next array index

and $1, $1, $2 # $1 = $1 and $2 = T and Array(i) = Anded value , gets rid

of T during this loop to save on registers

Compare: addi $8 $0, 16 # stop value for bit #

beq $7, $8, Save # Once $7 equals 16, go to next array index

addi $7, $7, 1 # Increment $7 by 1

andi $2, $1, 1 # $2 = $1 and 1 = Anded value and 1, $2 is temp value

add $3, $3, $2 # counts # of 1s in $1

srl $1, $1, 1

j Compare

Save: lw $1, 0x2010($0)

lw $2, 0x2014($0)

slt $6, $1, $3

beq $6, $0, Check1

add $1, $0, $3

addi $2, $0, 1

sw $1, 0x2010($0)

sw $2, 0x2014($0)

j Start

Check1: beq $1, $3, Check2

j Start

Check2: addi $2, $2, 1

sw $2, 0x2014

j Start

Exit:

2) Machine Code for Program 1

00000000

00000011

00000101

00000110

00001001

00001010

00001100

00001111

00010001

00010010

00010100

00010111

00011000

00011011

00011101

00011110

00100001

Machine Code for Program 2

00100010

00100100

00100111

00101000

00101011

00101101

00101110

00110000

00110011

00110101

00110110

00111001

00111010

00111100

00111111

01000001

01000010

01000100

01000111

01001000

01001011

01001101

01001110

01010000

01010011

01010101

3)

4) Python Code

|  |
| --- |
|  |
| def disassember(M, Nlines): |
|  | print("ECE 366 Group 17 Disassembler") |
|  | print("----------------") |
|  | #write to output file |
|  | for i in range(Nlines): |
|  | if(line[0:7] == "0000000"): |
|  | output\_file.write("lw $7, P #$7 = P \n") |
|  | elif(line[0:7] == "0000001"): |
|  | output\_file.write("add $9, $0, 1 #$9 = 1\n") |
|  | elif(line[0:7] == "0000010"): |
|  | output\_file.write("addi $10, $0, $0 # $10 =0 = counter N =exponent\n") |
|  | elif(line[0:7] == "0000011"): |
|  | output\_file.write("andi $10, $7, 1 # $10-$7 and 1(0/1) \n") |
|  | elif(line[0:7] == "0000100"): |
|  | output\_file.write("srl $7, $7, 1 # $7 >>1 \n") |
|  | elif(line[0:7] == "0000101"): |
|  | output\_file.write("beq $10, $0, Mult2 # if zero go to Mult2") |
|  | elif(line[0:7] == "0000110"): |
|  | output\_file.write("add $10, $9, $9 # temp $10 = $9 + $9 + $10 =2\n") |
|  | elif(line[0:7] == "0000111"): |
|  | output\_file.write("add $9, $9, $9 # $9 = $9 + $9 =$9 =2 $0\n") |
|  | elif(line[0:7] == "0001000"): |
|  | output\_file.write("add $9, $10, $9 # $9 = $10 +$9 = $9 = 6\n") # last one edited all after are from old code |
|  | elif(line[0:7] == "0001001"): |
|  | output\_file.write("add $18, $18, $14\t\t\tWe add the value $14 to the register $18\n") |
|  | elif(line[0:7] == "0001010"): |
|  | output\_file.write("addi $8, $0, 1\n") |
|  | elif(line[0:7] == "0001011"): |
|  | output\_file.write("addi $10, $0, 6\t\t\tconstant 6 is loaded into register 10\n") |
|  | elif(line[0:7] == "0001100"): |
|  | output\_file.write("addi $11, $0, 20\t\t\tWe store the number of words in the array, useful for loops\n") |
|  | elif(line[0:7] == "0001101"): |
|  | output\_file.write("addi $10, $10, -17\n") |
|  | elif(line[0:7] == "0001110"): |
|  | output\_file.write("addi $11, $11, -1\n") |
|  | elif(line[0:7] == "0001111"): |
|  | output\_file.write("addi $12, $0, 0x2000\t\t\tLoad the data address in $12 useful for loading operands\n") |
|  | elif(line[0:7] == "0010000"): |
|  | output\_file.write("addi $12, $0, 17\t\t\tWe load a 17 for the modulus\n") |
|  | elif(line[0:7] == "0010001"): |
|  | output\_file.write("addi $13, $13, -1\n") |
|  | elif(line[0:7] == "0010010"): |
|  | output\_file.write("addi $14, $0, 32\n") |
|  | elif(line[0:7] == "0010011"): |
|  | output\_file.write("addi $15, $0, 0x2000\t\t\tstart address of the data is stored in $15\n") |
|  | elif(line[0:7] == "0010100"): |
|  | output\_file.write("addi $15, $0, 0x2004\t\t\tWe store the address to which the result shall be saved in the register $15\n") |
|  | elif(line[0:7] == "0010101"): |
|  | output\_file.write("addi $15, $0, 20\t\t\tnumbers of words in the array\n") |
|  | elif(line[0:7] == "0010110"): |
|  | output\_file.write("addi $15, $15, -1\t\t\tWe must do this for all words in the array\n") |
|  | elif(line[0:7] == "0010111"): |
|  | output\_file.write("addi $15, $0, 0\t\t\tCounter of scores\n") |
|  | elif(line[0:7] == "0011000"): |
|  | output\_file.write("addi $15, $15, 1\t\t\tIf they are equal, the score is incremented\n") |
|  | elif(line[0:7] == "0011001"): |
|  | output\_file.write("addi $17, $17, 4\t\t\tAdvance to next array word\n") |
|  | elif(line[0:7] == "0011010"): |
|  | output\_file.write("addi $17, $15, 0xC\t\t\taddress of the array is stored in $17\n") |
|  | elif(line[0:7] == "0011011"): |
|  | output\_file.write("addi $17, $17, 4\t\t\tMove to the next position in the array\n") |
|  | elif(line[0:7] == "0011100"): |
|  | output\_file.write("addi $17, $0, 0x2004\n") |
|  | elif(line[0:7] == "0011101"): |
|  | output\_file.write("addi $17, $0, 0x205C\t\t\tHere we save start address of score array in $17\n") |
|  | elif(line[0:7] == "0011110"): |
|  | output\_file.write("addi $17, $0, 0x2008\t\t\tWe recover the given address to store result\n") |
|  | elif(line[0:7] == "0011111"): |
|  | output\_file.write("addi $18, $0, 0\n") |
|  | elif(line[0:7] == "0100000"): |
|  | output\_file.write("andi $11, $15, 1\t\t\tIf not: it gets lowest bit value by andi operation between $15 and 1\n") |
|  | elif(line[0:7] == "0100001"): |
|  | output\_file.write("andi $14, $16, 1\t\t\tWith the handy andi, we get the most significant bit\n") |
|  | elif(line[0:7] == "0100010"): |
|  | output\_file.write("beq $11, $0, Continue\n") |
|  | elif(line[0:7] == "0100011"): |
|  | output\_file.write("beq $11, $0, EndOfProgram\n") |
|  | elif(line[0:7] == "0100100"): |
|  | output\_file.write("beq $13, $0, ReturnToPrevious\t\t\tIf one Operand is zero we return, since the result is zero\n") |
|  | elif(line[0:7] == "0100101"): |
|  | output\_file.write("beq $14, $0, ReturnToPrevious\t\t\tIf one Operand is zero we return, since the result is zero\n") |
|  | elif(line[0:7] == "0100110"): |
|  | output\_file.write("beq $15, $0, ExponentZero\n") |
|  | elif(line[0:7] == "0100111"): |
|  | output\_file.write("beq $16, $0, endHamming\n") |
|  | elif(line[0:7] == "0101000"): |
|  | output\_file.write("bne $11, $0, ReturnToPrevious\n") |
|  | elif(line[0:7] == "0101001"): |
|  | output\_file.write("bne $13, $0, SkipSaving\n") |
|  | elif(line[0:7] == "0101010"): |
|  | output\_file.write("bne $15, $0, LoopWords\n") |
|  | elif(line[0:7] == "0101011"): |
|  | output\_file.write("bne $10, $16, NextStep\t\t\tIf it is lower, then nothing will happen\n") |
|  | elif(line[0:7] == "0101100"): |
|  | output\_file.write("j MainLoop\t\t\tJump to Mainloop\n") |
|  | elif(line[0:7] == "0101101"): |
|  | output\_file.write("j SecondaryLoop\t\t\tWe keep going until $13 is zero\n") |
|  | elif(line[0:7] == "0101110"): |
|  | output\_file.write("j EXIT\n") |
|  | elif(line[0:7] == "0101111"): |
|  | output\_file.write("j Modulus\t\t\tKeep calculating modulus\n") |
|  | elif(line[0:7] == "0110000"): |
|  | output\_file.write("j Hamming\t\t\tKeep looping\n") |
|  | elif(line[0:7] == "0110001"): |
|  | output\_file.write("j LoopScores\n") |
|  | elif(line[0:7] == "0110010"): |
|  | output\_file.write("jal Multiplication\n") |
|  | elif(line[0:7] == "0110011"): |
|  | output\_file.write("jal Modulus\n") |
|  | elif(line[0:7]== "0110100"): |
|  | output\_file.write("jal IncrementCount\n") |
|  | elif(line[0:7]== "0110101"): |
|  | output\_file.write("jr $31\n") |
|  | elif(line[0:7]== "0110110"): |
|  | output\_file.write("lw $10, 4($15)\t\t\tThe current best matching score is loading into $10 from memory\n") |
|  | elif(line[0:7]== "0110111"): |
|  | output\_file.write("lw $11, 0($15)\t\t\tPattern is loaded into $11\n") |
|  | elif(line[0:7]== "0111000"): |
|  | output\_file.write("lw $15, 0($12)\t\t\tthe exponent is loaded into 15\n") |
|  | elif(line[0:7]== "0111001"): |
|  | output\_file.write("lw $16, 0($17)\t\t\tWe load a score from array\n") |
|  | elif(line[0:7]== "0111010"): |
|  | output\_file.write("slt $11, $10, $12\t\t\tIf we found out that the number is already lower thant seventeen, then we're done\n") |
|  | elif(line[0:7]== "0111011"): |
|  | output\_file.write("slt $13, $14, $10\t\t\tIf the score is lower than the best, we will not save it\n") |
|  | elif(line[0:7]== "0111100"): |
|  | output\_file.write("srl $15, $15, 1\t\t\tit advances to the next bit\n") |
|  | elif(line[0:7]=="0111101"): |
|  | output\_file.write("srl $16, $16, 1\t\t\tWe shift to compare with the next bit\n") |
|  | elif(line[0:7]=="0111110"): |
|  | output\_file.write("sub $14, $14, $13\t\t\tCalculation of score\n") |
|  | elif(line[0:7]=="0111111"): |
|  | output\_file.write("sw $8, 4($12)\t\t\tit saves 1 in the result because if the exponent is 0\n") |
|  | elif(line[0:7]=="1000000"): |
|  | output\_file.write("sw $10, 0($17)\t\t\tWe save the best score in the given address\n") |
|  | elif(line[0:7]=="1000001"): |
|  | output\_file.write("sw $10, 0($15)\t\t\tWe save the result\n") |
|  | elif(line[0:7]== "1000010"): |
|  | output\_file.write("sw $14, 80($17)\t\t\tCalculate direction is score array\n") |
|  | elif(line[0:7]== "1000011"): |
|  | output\_file.write("sw $15, 0($17)\t\t\tthe result is stored back into data\n") |
|  | elif(line[0:7]== "1000100"): |
|  | output\_file.write("xor $16, $16, $11\t\t\tThe xor will give us the information needed to count for the hamming distance\n") |
|  |  |
|  | def simulate(I,Nsteps): |
|  | print("Project 3 Group 17 Ulitmately Simple: Simulator") |
|  | PC = 0 #prgram counter |
|  | DIC= 0 |
|  | Reg = [0,0,0,0] |
|  | # Memory = [0 for i in range(10)] # data memory, |
|  | print("\*\*\*\*\*\*\*\*\*\* Simluation starts \*\*\*\*\*\*\*\*\*\*") |
|  | finished = False |
|  | while( not(finished)): |
|  | fetch = Instructions[PC] |
|  | DIC += 1 |
|  | print(fetch) |
|  |  |
|  |  |
|  | def main(): |
|  | //instr\_file = open("P1\_Instruction.txt","r") |
|  | data\_file = open("project3\_group\_17\_p1\_bin.txt" ,"r") |
|  | data\_file2 = open("project3\_group\_17\_p2\_bin.txt", "r") |
|  | #we need a file for the data set |
|  | #Nsteps = 3 #How many cycles to run before output |
|  | Nlines = 0 #How may instrs total in input.txt |
|  | Instructions = [] #all instructions will be stored here |
|  | Memory = [] |
|  | print( " ECE 366 Group 8") |
|  | #print( " 1 = simulator") |
|  | print( " 2 = disassembler") |
|  | #print( " 3 = assembler") |
|  |  |
|  | mode= int(input( "Please enter the mode of Program: ")) |
|  | print( "Mode selected: ",end=" ") |
|  | modedis= int(input( "Please enter the which program 1 or 2: ")) |
|  | print( "Mode selected: ", end=" ") |
|  |  |
|  |  |
|  | if(mode == 1): #Check whether to use disassembler of assembler or simulator |
|  | #simulator(Instructions,Nsteps,debug\_mode,Memory) |
|  | print("assembler") |
|  | elif(mode== 2): |
|  | if (modedis == 1): |
|  | for line in data\_file: # Read in data P1\_Machine.txt |
|  | if(line== "\n" or line[0] =='#'): |
|  | continue |
|  | Memory.append(line) |
|  | Nlines+=1 |
|  | elif(modedis == 2): |
|  | for line in data\_file2: # Read in data P1\_Machine.txt |
|  | if(line== "\n" or line[0] =='#'): |
|  | continue |
|  | Memory.append(line) |
|  | Nlines+=1 |
|  | else: |
|  | print("That is not one of the options") |
|  |  |
|  | disassembler(Memory,Nlines) |
|  | print("disassembler is being done") |
|  | elif(mode== 3): |
|  | #assembler(Instructions,Nlines) |
|  | print("assembler is being done") |
|  | else: |
|  | print("Error. Unrecognized mode. Exiting") |
|  | exit() |
|  |  |
|  | #instr\_file.close() |
|  | data\_file2.close() |
|  | data\_file.close() |
|  |  |
|  | if \_\_name\_\_ == "\_\_main\_\_": |
|  | main() |
|  |  |

5) 

