Group 15 ISA Project 2

1. ISA Intro

1. “Group 15 ISA” was designed with simplicity in mind. Due to the number of registers, instructions and immediates used, we could not create a more formatedISA that determines instructions by the number of N bits etc. Instead, we created our ISA to hold a decimal value for each different instruction in both our program one and two(e.g. add $8, $18,$0 is set to the equal 0 = 00000000). This is an extremely simple/specific ISA which was best for our program. An explanation as to each different binary value is not needed as our ISA is too specific provided the comment code or instructions itself is understandable.

2. Below is a list of different opcode instructions and their binary values. Not all are included but enough to explain our ISA. A total of 14 opcodes(instructions), 8 registers are utilized.

|  |  |  |  |
| --- | --- | --- | --- |
| Decmal Value | Binary/Machine code | Instruction | Description |
| 0 | 0000000 | add $8, $18,$0 |  |
| 1 | 0000001 | add $10, $0, $8 | # To calculate the modulus 17 we make succesive substractions |
| 2 | 0000010 | add $10, $0, $14 | # If not: then current score will be saved as best score |
| 3 | 0000011 | add $10, $18, $0 |  |
| 4 | 0000100 | add $13, $13, $14 | # We add the result of the hamming count. If it's zero, then nothing happens |
| 5 | 0000101 | add $13, $0, $0 | # We will be storing the hamming distance in $13, therefore, we load a 0 |
| 6 | 0000110 | add $13, $10, $0 |  |
| 7 | 0000111 | add $14, $8, $0 |  |
| 8 | 0001000 | add $14, $10, $0 | # the base is elevated to the power of two |
| 9 | 0001001 | add $18, $18, $14 | # We add the value $14 to the register $18 |
| 10 | 0001010 | addi $8, $0, 1 |  |
| 11 | 0001011 | addi $10, $0, 6 | # constant 6 is loaded into register 10 |
| 12 | 0001100 | addi $11, $0, 20 | # We store the number of words in the array, usfeul for oops |
| 13 | 0001101 | addi $10, $10, -17 |  |
| 14 | 0001110 | addi $11, $11, -1 |  |
| 15 | 0001111 | addi $12, $0, 0x2000 | # Load the data address in $12 useful for loading operands |
| 16 | 0010000 | addi $12, $0, 17 | # We load a 17 for the modulus |
| 17 | 0010001 | addi $13, $13, -1 |  |
| 18 | 0010010 | addi $14, $0, 32 |  |
| 19 | 0010011 | addi $15, $0, 0x2000 | # start address of the data is stored in $15 |
| 20 | 0010100 | addi $15, $0, 0x2004 | # We store the address to which the result shall be saved in the register $15 |
| 21 | 0010101 | addi $15, $0, 20 | # numbers of words in the array |
| 22 | 0010110 | addi $15, $15, -1 | # We must do this for all words in the array |
| 23 | 0010111 | addi $15, $0, 0 | # Counter of scores |
| 24 | 0011000 | addi $15, $15, 1 | # If they are equal, the score is incremented |
| 25 | 0011001 | addi $17, $17, 4 | # advance to next array word |
| 26 | 0011010 | addi $17, $15, 0xC | # address of the array is stored in $17 |
| 27 | 0011011 | addi $17, $17, 4 | # Move to the next position in the array |
| 28 | 0011100 | addi $17, $0, 0x2004 |  |
| 29 | 0011101 | addi $17, $0, 0x205C | # Here we save start address of score array in $ |
| 30 | 0011110 | addi $17, $0, 0x2008 | # We recover the given address to store result |
| 31 | 0011111 | addi $18, $0, 0 |  |
| 32 | 0100000 | andi $11, $15, 1 | # If not: it gets lowest bit value by andi operation between $15 and 1 |
| 33 | 0100001 | andi $14, $16, 1 | # With the handy andi, we get the most significant bit |
| 34 | 0100010 | beq $11, $0, Continue |  |
| 35 | 0100011 | beq $11, $0, EndOfProgram |  |
| 36 | 0100100 | beq $13, $0, ReturnToPrevious | # If one Operand is zero we return, since the result is zero |
| 37 | 0100101 | beq $14, $0, ReturnToPrevious | # If one Operand is zero we return, since the result is zero |
| 38 | 0100110 | beq $15, $0, ExponentZero | # If Exponent is Zero we go to ExponentZero |
| 39 | 0100111 | beq $16, $0, endHamming |  |
| 40 | 0101000 | bne $11, $0, ReturnToPrevious |  |
| 41 | 0101001 | bne $13, $0, SkipSaving |  |
| 42 | 0101010 | bne $15, $0, LoopWords |  |
| 43 | 0101011 | bne $10, $16, NextStep | # If it is lower, then nothing will happen |
| 44 | 0101100 | j MainLoop | # Jump to Mainloop |
| 45 | 0101101 | j SecondaryLoop | # We keep goint until $13 is zero |
| 46 | 0101110 | j EXIT |  |
| 47 | 0101111 | j Modulus | # Keep calculating modulus |
| 48 | 0110000 | j Hamming | # Keep looping |
| 49 | 0110001 | j LoopScores |  |
| 50 | 0110010 | jal Multiplication |  |
| 51 | 0110011 | jal Modulus |  |
| 52 | 0110100 | jal IncrementCount |  |
| 53 | 0110101 | jr $31 |  |
| 54 | 0110110 | lw $10, 4($15) | # The current best matching score is loading into $10 from memory |
| 55 | 0110111 | lw $11, 0($15) | # pattern is loaded into $11 |
| 56 | 0111000 | lw $15, 0($12) | # the exponent is loaded into 15 |
| 57 | 0111001 | lw $16, 0($17) | # We load a score from array |
| 58 | 0111010 | slt $11, $10, $12 | # If we found out that the number is already lower thant seventeen, then we're done |
| 59 | 0111011 | slt $13, $14, $10 | # If the score is lower than the best, we will not save it |
| 60 | 0111100 | srl $15, $15, 1 | # it advances to the next bit |
| 61 | 0111101 | srl $16, $16, 1 | # We shift to compare with the next bit |
| 62 | 0111110 | sub $14, $14, $13 | # Calculation of score |
| 63 | 0111111 | sw $8, 4($12) | # it saves 1 in the result because if the exponent is 0 |
| 64 | 1000000 | sw $10, 0($17) | # We save the best score in the given address |
| 65 | 1000001 | sw $10, 0($15) | # We save the result |
| 66 | 1000010 | sw $14, 80($17) | # Calculate direction is score array |
| 67 | 1000011 | sw $15, 0($17) | # the result is stored back into data |
| 68 | 1000100 | xor $16, $16, $11 | # The xor will give us the information needed to count for the hamming distance |

3. A total of 8 registers are supported from our first 2 programs. This is due to lack of efficiency in our first project.

4. We support BEQ, BNE, J , JAL, JLR and JR. Some of these are almost equivalent to each other but with different wording. For example, we could get rid of BNE and use BEQ instead but its logic was implemented at the time of creating project 1. OUR ISA will outline all the branching and to where it sends pc address to.

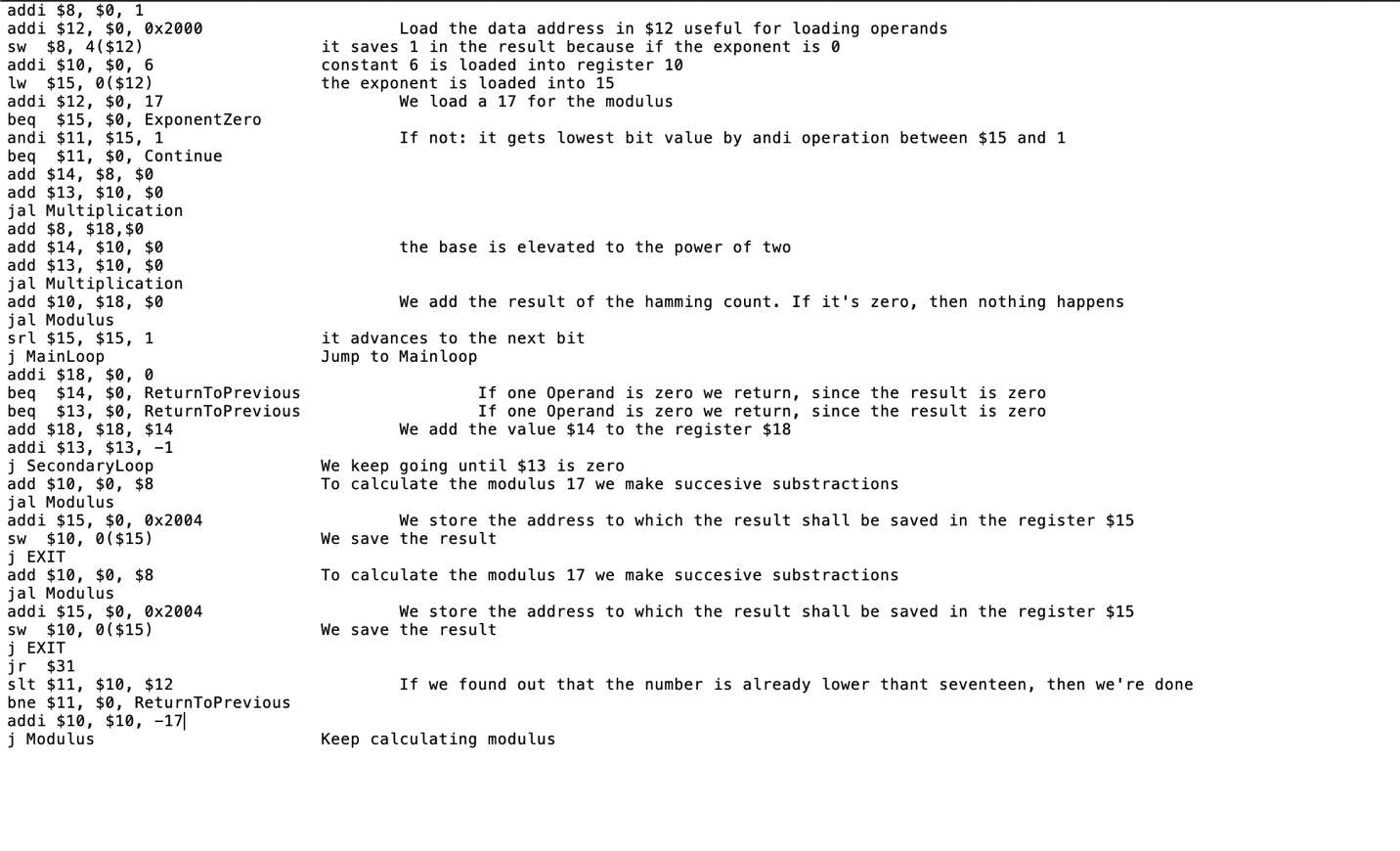
5. Data memory and addressing modes are the same to what we have been doing in class. PC will increment by the decimal value 1 to the next instruction.

1. Answers to questions
2. Compared to the Straightforward ISA, ours excels in simplicity and understanding. There is no need to decipher our machine code. The only thing required to disassemble our machine code is to convert the machine code into decimal. From there, we can look at out ISA table and determine the specific instruction. Although it might lack complexity, and hinder efficiency, we believe it is the easiest ISA to understand.
3. Depending on the Hardware architecture, this could lead to a low cost design as it primarily involves the use of the control unit(Decoder). All the control unit has to do is determine its decimal value and send it to the proper destination. No binary segment of the machine code needs to be evaluated, instead the whole machine code is deciphered. Although this may lead to more work on the control unit, it also means the control unit does not wait on any input from data memory or registry but send information to memory and registry instead which creates a linear workflow. This lack of input time leads also leads to faster processing time as fewer clock cycles are needed.
4. Based on our project 1, it was thought we needed 1 extra bit based on conventional ISA method which was to set N bits for opcode, N bits for registers, etc. A lot of registers and opcode were used which is what created the problem to begin with. Then we counted the number of different instructions within both our programs and determined it to be 69. The number of different instructions we could have is 2^7 which gives us 128 possibilities. Our 69 instructions would falls well within our boundaries so a decimal value for each instructions would be the simplest ISA implementation we could create. Therefore the 7 bits were enough for our ISA. Based off this methodology, we would only need to rewrite about 5 instructions to utilized 64 options which is only 6 bits. As previously stated, this could be accomplished by rewording either our BNE, BEQ and even some arithmetic opcodes.
5. Our team got together once a week to discuss progress and outline the next steps to take. Most of the work was done individually since parts of this project were assigned to each individual. When we got together, we meshed our parts together to create the cohesiveness of the ISA, Python and machine code.
6. The biggest change we would implement would be the assembly code for project 1. It was decided to use this specific code because it met level 2 standards for both parts of project 1. This would come at a price as the code had a large number of different opcodes and utilized, maybe an unnecessary amount of, many registers. As to our ISA, it would depend on the number of different instructions and how we implement memery, etc.
7. Software Package
8. There is no algorithm for our ISA Since it is based onout the decimal value system. But our ISA does achieve any functionality that our project 1 code achieves.
9. Below is the machine code for each program.

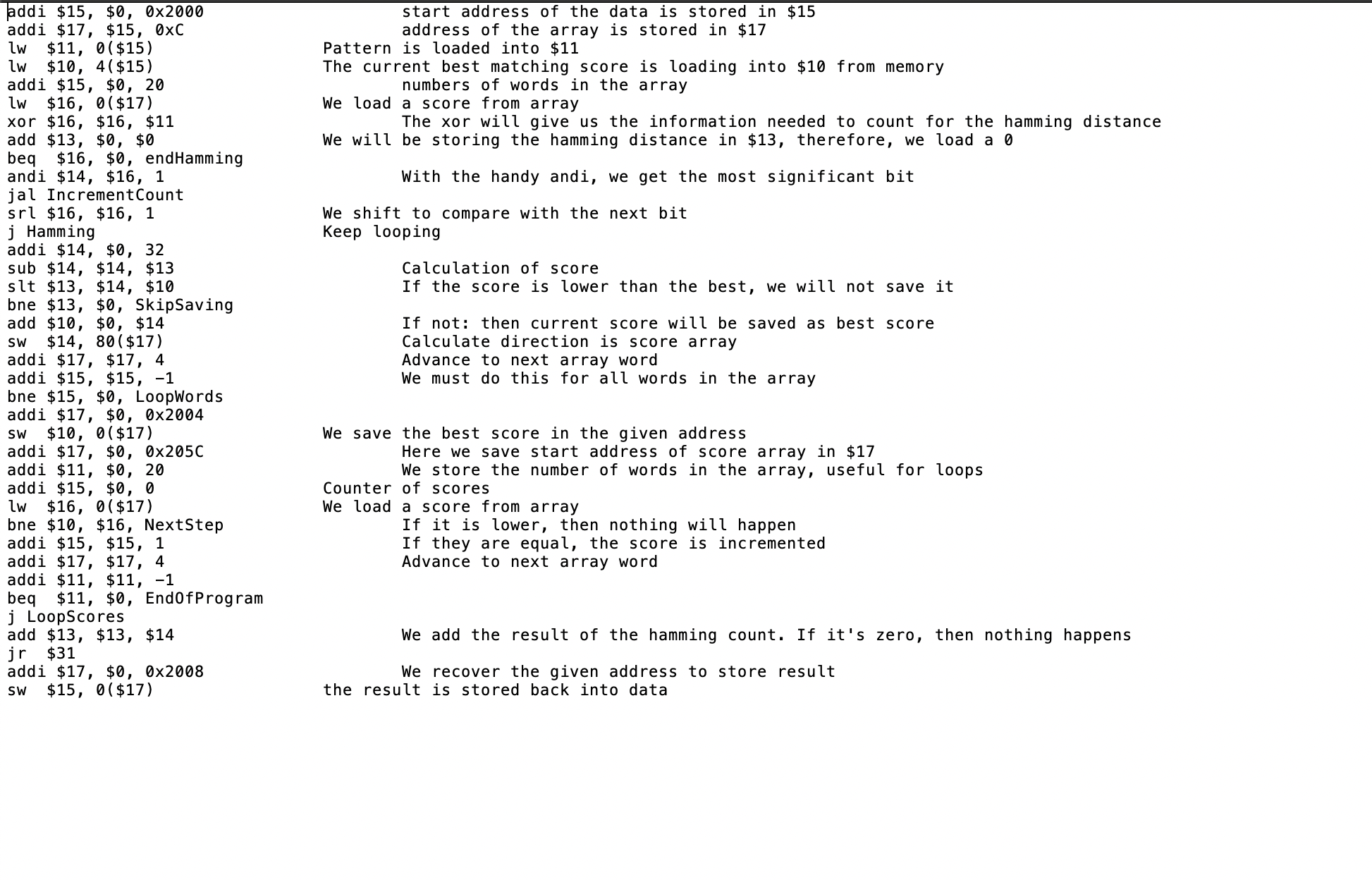
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Machine code for program 1 | |  | Machine code for program 2 | |
|  | 0001010 |  |  | 0010011 |
|  | 0001111 |  |  | 0011010 |
|  | 0111111 |  |  | 0110111 |
|  | 0001011 |  |  | 0110110 |
|  | 0111000 |  |  | 0010101 |
|  | 0010000 |  |  | 0111001 |
|  | 0100110 |  |  | 1000100 |
|  | 0100000 |  |  | 0000101 |
|  | 0100010 |  |  | 0100111 |
|  | 0000111 |  |  | 0100001 |
|  | 0000110 |  |  | 0110100 |
|  | 0110010 |  |  | 0111101 |
|  | 0000000 |  |  | 0110000 |
|  | 0001000 |  |  | 0010010 |
|  | 0000110 |  |  | 0111110 |
|  | 0110010 |  |  | 0111011 |
|  | 0000011 |  |  | 0101001 |
|  | 0110011 |  |  | 0000010 |
|  | 0111100 |  |  | 1000010 |
|  | 0101100 |  |  | 0011001 |
|  | 0011111 |  |  | 0010110 |
|  | 0100101 |  |  | 0101010 |
|  | 0100100 |  |  | 0011100 |
|  | 0001001 |  |  | 1000000 |
|  | 0010001 |  |  | 0011101 |
|  | 0101101 |  |  | 0001100 |
|  | 0000001 |  |  | 0010111 |
|  | 0110011 |  |  | 0111001 |
|  | 0010100 |  |  | 0101011 |
|  | 1000001 |  |  | 0011000 |
|  | 0101110 |  |  | 0011001 |
|  | 0000001 |  |  | 0001110 |
|  | 0110011 |  |  | 0100011 |
|  | 0010100 |  |  | 0110001 |
|  | 1000001 |  |  | 0000100 |
|  | 0101110 |  |  | 0110101 |
|  | 0110101 |  |  | 0011110 |
|  | 0111010 |  |  | 1000011 |
|  | 0101000 |  |  |  |
|  | 0001101 |  |  |  |
|  | 0101111 |  |  |  |

1. Below is our output for python disassembler

Program 1



Program 2



1. Python code for disassembler

Disassembler for project 2

#This code works for both program 1 and 2

input\_file = open("MachineCodeProgram2.txt", "r")

output\_file = open("project2\_group\_15\_p2\_asm.txt", "w")

for line in input\_file:

if(line[0:7] == "0000000"):

output\_file.write("add $8, $18,$0\n")

elif(line[0:7] == "0000001"):

output\_file.write("add $10, $0, $8\t\t\tTo calculate the modulus 17 we make succesive substractions\n")

elif(line[0:7] == "0000010"):

output\_file.write("add $10, $0, $14\t\t\tIf not: then current score will be saved as best score\n")

elif(line[0:7] == "0000011"):

output\_file.write("add $10, $18, $0\t\t\tWe add the result of the hamming count. If it's zero, then nothing happens\n")

elif(line[0:7] == "0000100"):

output\_file.write("add $13, $13, $14\t\t\tWe add the result of the hamming count. If it's zero, then nothing happens\n")

elif(line[0:7] == "0000101"):

output\_file.write("add $13, $0, $0\t\t\tWe will be storing the hamming distance in $13, therefore, we load a 0\n")

elif(line[0:7] == "0000110"):

output\_file.write("add $13, $10, $0\n")

elif(line[0:7] == "0000111"):

output\_file.write("add $14, $8, $0\n")

elif(line[0:7] == "0001000"):

output\_file.write("add $14, $10, $0\t\t\tthe base is elevated to the power of two\n")

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")

1. Hardware Implementation