Architecture name: Definitely not our ISA

**Overall Philosophy:**  
 Our group wanted to make a version of our ISA from project 2 that would utilize a set of instructions that were simpler to realize through hardware

**Specific Goals:**   
 Using the basic ISA from Project 2, our group stuck with the same commands and functions but switched around the format of the machine code to make it more normalized. Specifically, we aimed to reduce the size of the MUX leading into our instruction memory, as before it required a 4 bit MUX, but now only requires 3.

**Instruction list and details:**

|  |  |  |  |
| --- | --- | --- | --- |
| Instruction | OP code | Format | Example |
| lw | 000 01 x y | lw Rx, Ry | Rx = M[Ry]  Rx = $0, $1; Ry = $0, $2 |
| sw | 001  x iii | sw Rx, imm | M[imm] = Rx  Rx = $0, $1; imm = [0:7] |
| bgtR0 | 000 1  x ii | bgtR0 Rx, imm | Pc = pc + imm if Rx > $0  Rx = $1, $2; imm = [1:4] |
| bltR0 | 100 1 x  ii | bltR0 Rx, imm | Pc = pc + imm if Rx < $0  Rx = $1, $2; imm = [1:4] |
| shiftL | 010 10 xx | shiftL Rx | Rx << 1  Rx = [$0-$3] |
| incMem5 | 010 11 00 | IncMem6 | M[5] = M[5] + 1 |
| j | 11   iiiii | J Label | PC = imm  Imm = [0:31] |
| add | 100 0 xx y | add Rx, Ry | Rx = Rx + Ry  Rx = [$0-$3]; Ry = [$0&$2] |
| sub | 010 0 xx y | sub Rx, Ry | Rx = Rx - Ry  Rx = [$0-$3]; Ry = [$0&$2] |
| addi | 101 xx ii | addi Rx, imm | Rx = Rx + imm  Rx = [$0-$3]; imm = [-2:1] |
| init | 011  xx ii | init Rx, imm | Rx = imm  Rx = [$0-$3]; imm = [0:3] |

**Register design:**

Our ISA uses registers 0, 1, 2, and 3. None of these registers have special properties.

**Control flow and branch options:**

When branching, to keep the immediate as short as possible, it is only possible to branch forward, and in order to branch backwards a jump must be used instead. The target addresses for our branches is simple adding onto the PC, and for our jump it overwrites our PC value.   
 Branches can jump forward up to 4 commands, and jump can set PC to any value under 32.  
By limiting branches to jumping forward, they were able to skip 4 commands ahead using 1 less bit for the immediate value than we otherwise would have needed.

**Data memory addressing and storing:**

Our load and store words work very differently based on the different need of load ranges. Because our programs only need to store values into relatively low values in memory, our store function places either register 0 or register 1’s value into the memory address of an unsigned 3 bit immediate (0:7)

However, because loading is required on a much wider range of memory, our load function loads an address of memory defined by either register 0 or register 1 into either register 0 or register 2.

**Questions:**

1) The most significant advantages of our ISA is the versatility and range of commands. Our ISA can do almost all simple things and it starts to become limited by its shortage of registers. This shortage caused many problems when attempting to implement level 2 completion of program 2, but the wide spread of commands makes most programming under this ISA very possible / easy. A small limitation that could easily be changed if the requirements needed it to is the 3 bit range in storing memory. This could easily be switched to a register’s value in the event widespread storing was required. Rather than perfecting anything, our ISA is a good do-it-all ISA, with specifications for the needed tasks that make it more effective implemented where possible.

2) Not a lot of time had been spent on minimizing DIC, as the timeframe did not allow for it. However, in project 3 our ISA had some formatting changes made to make our hardware require shorter, or even less, MUX gates. If this project was restarted from the beginning a total overhaul on the ISA would be required. Ours is probably too generalized and can be totally reworked to form an ISA with more specific commands that would require a lower DIC, however our current ISA does not cause too much complication in our hardware, so any additional effort put forward would focus on DIC reduction rather than HW simplification, at least at first.

3)

A: My favorite part of this project was its reliance on previous projects, and being forced to deal with the limitations you set yourself earlier in the semester. This forces you to think outside the box and to do what you can with what you’ve already committed yourself to using. This directly leads into my least favorite part of the project, which is after realizing how many different ways you could approach your ISA design to make the code simpler, DIC lower, and hardware lighter, the time allotted for project 3 does not leave time for reworking a new, significantly better ISA.

B: If I was giving advice to somebody starting ECE 366 today, it would be to keep track of their algorithms from project 1, and to try to implement them with as little unnecessary capabilities as possible, because if you design yourself into a corner during the early stages of project 2, the end of project 2 and project 3 will be much harder than they need to be.

C: Most of the value of this project came from being forced into a situation where you need to make something very specific with a shortage of resources (bits, time) to work with. It forces consideration of what is absolutely necessary and what is not, and emphasizes the logical thinking skills required to do so. On top of this it also is in a team setting, where different members often had to make totally different parts to the same whole, and have them match up perfectly. All told, this project may not be a resume builder by itself, but the problem solving and teamwork skills it forced us to utilize are valuable to any workplace in any field.

**Pattern A:**

Program 2 Final DIC Count 1051, Program 1 DIC count 232

0000000000001001

0000000000010001

0000000000001011

0000000000000000

0000000000000000

0000000000000000

**Pattern B:**

Program 2 Final DIC Count 1051, Program 1 DIC count 7239

0000000100001011

0001000000000011

0000100101101111

0101010101010101

0000000000000000

0000000000000000

**Pattern C:**

Program 2 Final DIC Count 1051, Program 1 DIC count 116

0000000000000101

0000000000011101

0000000000000100

0000000000001111

0000000000010000

0000000000000100

**Pattern D:**

Program 2 Final DIC Count 1051, Program 1 DIC count 232

0000000000001001

0000000000100101

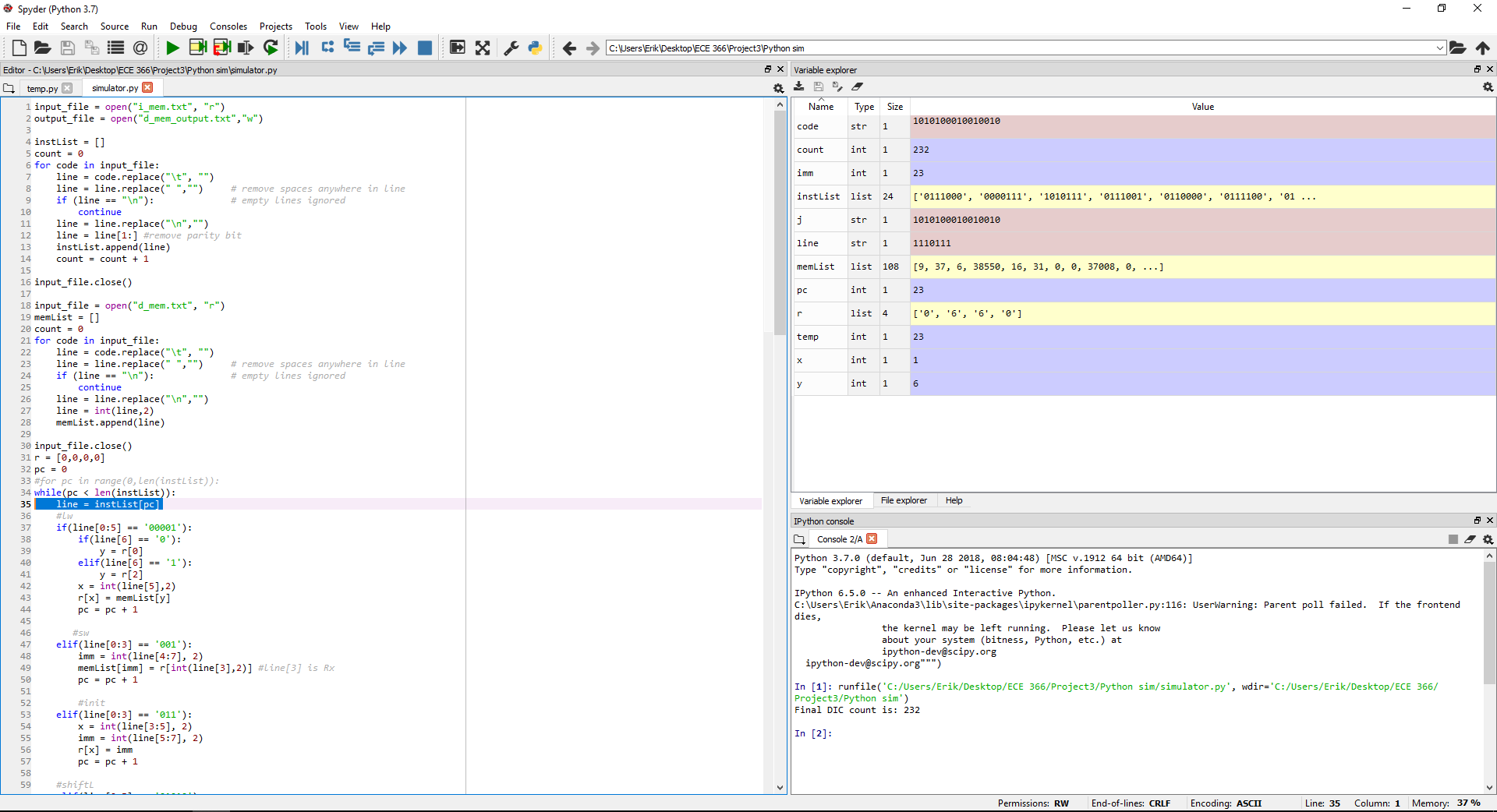
0000000000000110

1001011010010110

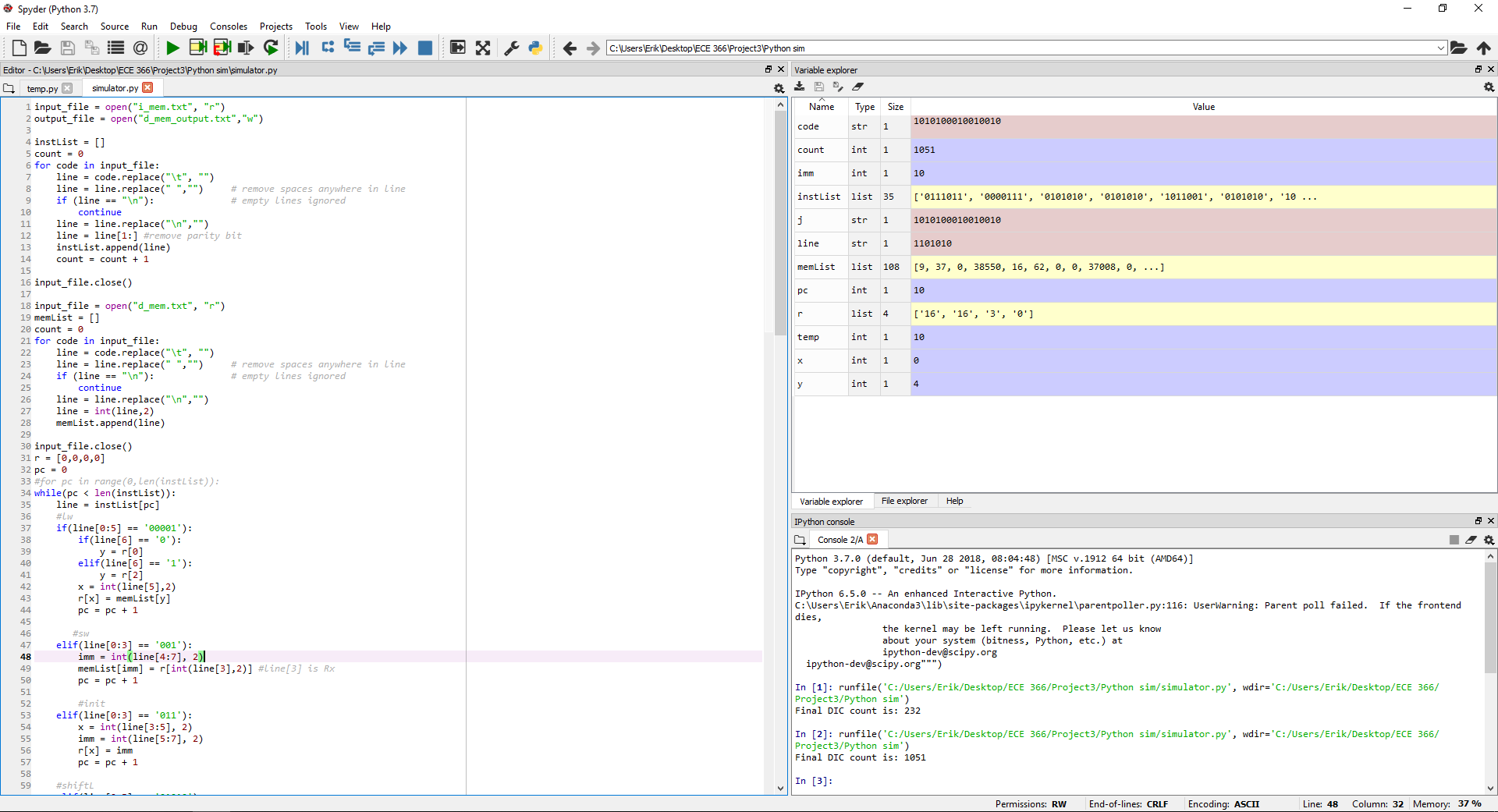
0000000000010000

0000000000011111

**Program 1 and 2 Python Screenshot:**



**Program 1 above**

**Program 2 below**

**Program 1 Instructions: Machine Code:**

init $2, 0 (1)011 1100

lw $1, $2 (1)000 0111

addi $1, -1 (1)1010111

init $2, 1 (0)011 1001

pwr: init $0, 0 (0)011 0000

init $3, 0 (0)011 1100

shiftL $2 (1)010 1010

add $0, $2 (0)100 0001

shiftL $0 (0)010 1000

add $2, $0 (0)100 0100

mod: init $0, 1 (1)011 0001

lw $0, $0 (1)000 0100

bltR0 $2, sMod (1)100 1110

sub $2, $0 (0)010 0100

j mod (0)110 1001

sMod: init $0, 1 (1)011 0001

sub $1, $0 (0)010 0010

init $0, 0 (0)011 0000

bltR0 $1, end (1)100 1001

j pwr (0)110 0011

end: init $1, 0 (1)011 0100

add $1, $2 (1)100 0011

sw $1, 2 (1)011 1010

done: j done (1)111 0110

**Program 2 Instructions Machine Code Comments**

init $2, 3 (1)0111011 $2 = 3  
 lw $1, $2 (1)0000111 $0 = M[$3] = M[3]  
 shiftL $2 (1)0101010 $2 = 6  
 shiftL $2 (1)0101010 $2 = 12  
 addi $2, 1 (0)1011001 $2 = 13  
 shiftL $2 (1)0101010 $2 = 26  
 addi $2, 1 (0)1011001 $2 = 27  
 shiftL $2 (1)0101010 $2 = 54  
 shiftL $2 (1)0101010 $2 = 108  
 j newWord (1)1101011 skips the placehold stall  
placeHold: j placeHold (0)1101010 stops PC  
newWord: addi $2, -1 (1)1011011 $2 = $2 – 1  
 lw $0, $2 (0)0000101 $1 = M[$2]  
 bltR0 $1, notEqual (1)1001010 Skips increment  
 bgtR0 $1, notEqual (0)0001001 Skips increment  
 incMem5 (1)0101100 M[5] = M[5] + 1  
notEqual: init $0, 2 (1)0110010 $0 = 2  
 shiftL $0 (0)0101000 $0 = 4  
 shiftL $0 (0)0101000 $0 = 8  
 addi $0, 1 (1)1010001 $0 = 9  
 bltR0 $2, noMoreWords(0)1001101 Checks if done  
 j newWord (1)1101011 Jumps to next work  
noMoreWords: init $2, 3 (1)0111011 $2 = 3  
 sub $0, $2 (0)0100001 $0 = 6  
 addi $0, -1 (0)1010011 $0 = 5  
 lw $1, $0 (0)0000110 $1 = M[5]  
 init $0, 0 (0)0110000 $0 = 0  
 bgtR0 $1, matchFound (0)0001001 If $1 > 0, exact found  
 j placeHold (0)1101010 Stops if exact not found   
 init $0, 2 (1)0110010 $0 = 2  
 shiftL $0 (0)0101000 $0 = 4  
 shiftL $0 (0)0101000 $0 = 8  
 shiftL $0 (0)0101000 $0 = 16  
 sw $0, 4 (0)0010100 M[4] = 16  
 j placeHold (0)1101010 jumps to stall

**Program 1 description:**

Program one utilizes a special algorithm that can exponentiate and modulate a certain number. The algorithm is R = R \* 6 % Q where R is initially set equal to 1. This pattern loops P number of times (exponent). The assembly code in the beginning sets the values of the registers to their respective values. Afterwards, the multiplication section of the algorithm takes place under pwr:. Next comes the actual modulate which happens by subtracting the mod value until that value is less than the given modulo value. The remainder is the value of the modulo. sMod: is the section that counts how many more times to loop. The loop program will store the values back into memory when the loop counter is equal to -1, otherwise the program will jump back to pwr: and the process will repeat.

**Program 1 machine code:**

10111000

10000111

11010111

00111001

00110000

00111100

10101010

01000001

00101000

01000100

10110001

10000100

01001110

00100100

01101010

10110001

00100010

00110000

11001001

01100100

10110100

11000011

10011010

11110111

**Program 2 machine code:**

10111011  
10000111  
10101010  
10101010  
01011001  
10101010  
01011001  
10101010  
10101010  
11101011  
11101010  
11011011  
00000101  
11001010  
00001001  
10101100  
10110010  
00101000  
00101000  
11010001  
01001101  
11101011  
10111011  
00100001  
11010010  
00000110  
00110000  
00001001  
01101010  
10110010  
00101000  
00101000  
00101000  
00010100  
01101010

**Data Memory Samples C & D**

****

****

**Hardware:**

