Teja Thangella
Jacob Scott
11/6/18
ECE 366 Project #3 Report

Part A

1. The name of our ISA is called JT for its creators Jacob Scott and Teja Thangella. The goal of this ISA was to create the easiest possible way for the user to manipulate and write the program in as little code as possible. This is done by using labels and being flexible enough to support various commands and could theoretically be applied to other programs.
2.

OP	Format	Instruction	Purpose
0000	SAR (# of what register is going to become the active register)	SAR	Set which register is the active register.
0001	ADD (# of the supplementary register to be added to the active register)	ADD	To add the value of 2 registers. The value of supplementary register is added to the active register.
0010	MOV (# of the supplementary register to be to be have it's value moved to the active register)	MOV	Move values between registers. The value in the supplementary register is moved to the value of the active register.
0011	AND (# of the supplementary register to be to be have it's value anded to the value in the active register)	AND	And values between registers. The value in the supplementary register is anded with the value of the active register and saved in the active register.
0100	SLL (# of the supplementary register to shift the active registers value by)	SLL	Shift the value in active register by the value at the supplementary register.
0101	SRL (# of the supplementary register to shift the active registers value by)	SRL	Shift the values in the active register by the value at the supplementary register.

0110	LOAD (Supplementary register holding the memory location to be accessed)	LOAD	Load a value from memory at location dictated by the supplementary register and save it into the active register.
0111	STOR (Supplementary register holding the address of the memory where the value in the active register is to be stored)	STOR	Save the value from the active register into the memory location dictated by the supplementary register.
1000	SUB (Supplementary register to subtract from the value in the active register)	SUB	Subtract the value in the active register by the value in the supplementary register and save the value into the active register.
1001	MLT (Supplementary register to multiply by the value in active register)	MLT	Multiply the value in the active register by the value in the supplementary register and save the value into the active register.
1010	BSLT (Supplementary register to be compared)	BSLT	If the value at the active register is less than the value in the supplementary register then branch to the label indicated by register p.
1011	MOVI (A number ranging from [-4, 3])	MOVI	Put the value dictated by the argument [-4, 3] into the active register.
1100	ADDI (A number ranging from [-4, 3])	ADDI	Add the value dictated by the argument [-4, 3] to the value in the active register.
1101	SJL (Supplementary register whose value is going to be loaded into register p)	SJL	Set register p to equal to the value in the active register plus. the supplementary register
1110	B	B	Set PC to be equal to the label location where the label is dictated by the register p.
1111	STOP	STOP	Stop all execution.

3. There is one unique register that is called Rp this register is unique as it is only used in branching and can be set to label that will automatically jump to that label and execute that label. This allows for less code that would otherwise make the programmer write additional code to write the branch distance into a register. This is both tedious and increases the DIC. Thus, it is far more efficient to have a label that would do the calculation before hand.
   The other registers are general purpose register used to hold 16-bit values. There are 7 total general purpose register that can all become the active register via the instruction SAR (set active register). Setting a general purpose as active means that it becomes the target of many of the instructions in the ISA such as add, sub, and mov.
4. The types of branch control would the BSLT command this would be the used for comparing the active register and the supplementary register. If the active register is less than the supplementary register then the PC branches to desired label indicated by the current value of Rp. This type of branching is calculated by finding all of the labels inside of the code and assigning them a number starting at 0. Then all the line numbers where the branches were located are saved into a Jump File in order that they are found in the assembly code. By placing the desired branch value into the Rp register this would allow for the code to branch to the desired label. This allows for a theoretical unlimited branch number as you are only limited by number of labels in you program and the max value that will fit your 16-bit register.
   Assume R0 is active with a value of

Assembly Code	Machine Code	Comment
BSLT 2	1 1010 010	# If Ra < R2 then PC = Jump[Rp]

5. We have a load and store instruction to fetch and save values from and to data memory. They each work by passing in a supplementary register that holds the address to be accessed inside that data memory. In the instance of load the value at that address is put into the current active register and in the instance of store the value in the active registered is saved into the data memory at that address.

Assembly Code	Machine Code	Comment
LOAD 2	1 0110 010	# Ra = MEM[R2]
STOR 2	0 0111 010	# MEM[R2] = Ra

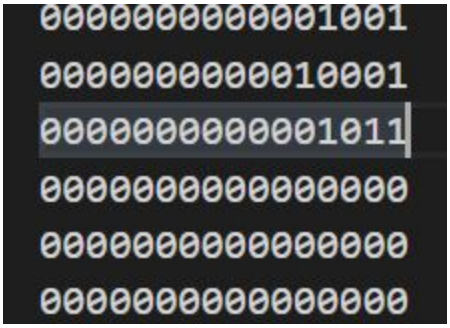
Part B

1. The biggest advantage out ISA has is that it has many different types of instructions. It has the best tools for writing whatever programs you need to, a lot of ISA designs we discussed usually had severe limitations in the amount of instructions supported, this is one of the reasons we chose this active register design principle. The main limitation is programmer headache, with so many instructions and such a small amount of bit space to run them on each instruction doesn't do much. This means that it takes more instructions to get anything done, for example adding values from 2 registers if they are already initialized takes 2 instructions and way more if one or both don't have the right values in them. We added a lot of hardware overhead to save on even more programmer headache, the original design of the ISA didn't have labels supported in it so branching in the programs was a huge pain. To remedy this we added the jump file which added more hardware but made the ISA usable because without labels programs were too hard to create.
2. To lower the DIC we took added jumping by labels this cut down on the sometimes 20 instructions that were needed to run to get the right jump location. This one changed easily saved hundreds of instructions in each program. However, this did make our hardware more difficult because we needed to add a jump file and update the data path a little bit. The hardware wasn't really simplified from version 1 to 2 but the advantage of labels from the programmer's perspective made it well worth the trouble getting the hardware to work. I would try and introduce a better way to jump to labels then are current method, also adding more conditional jumping statements and taking out instructions that aren't needed as much would be a huge benefit. This would require changing the control unit and the ALU so we didn't think it was worth the effort when looking at the ISA at the start of this project.
3. Reflection
    a. We learned that it is very hard to design an ISA that works for both the programmer and hardware designer. The cuts and additions that need to be made always make one person's job harder while taking the load off another. These last two projects have opened our eyes to what it must have been like working for IBM in the 80's.
    b. I would tell them to figure out a solid programmer friendly way to branch right away. That should be the first problem that they solve because both programs can be done with around five or six distinct instructions and maybe less if you are willing to use NAND more. However, the amount of time that will be saved cannot be accurately measured in hours when the programmer is able to write code that can predictably branch to labels.
    c. Working in groups and creating something new is always a difficult experience. Especially when some people are better at software and others love to design

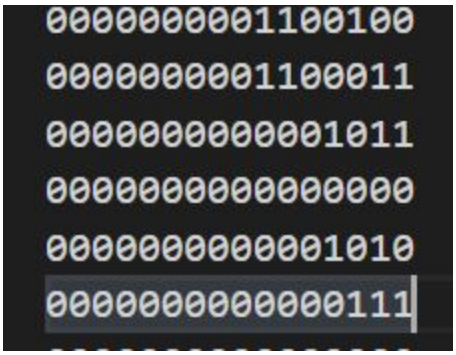
hardware. That alone would be a good topic to talk about in an interview but we built something very cool. This would be a project that I can proudly display and talk about every piece of with a potential employer and the value of that is limitless.

Part C:

1. Result for Pattern A



This is after Program 1 has run



This after both programs have run.

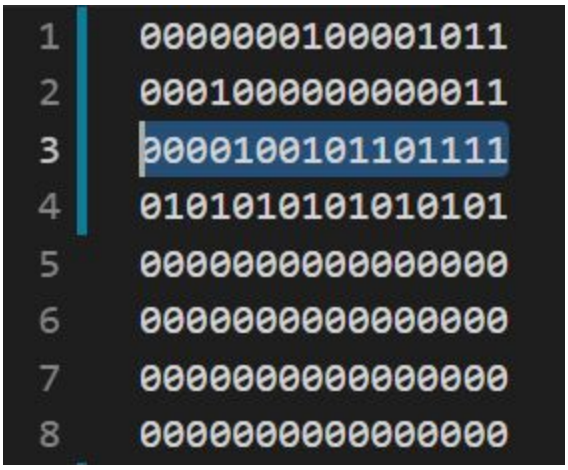
```
Registers: {'0': 2, '1': 9, '2': 11, '3': 9, '4': 17, '5': 6, '6': 4, '7': 8, 'p': 4}
Dynamic Instrcution Count: 592
```

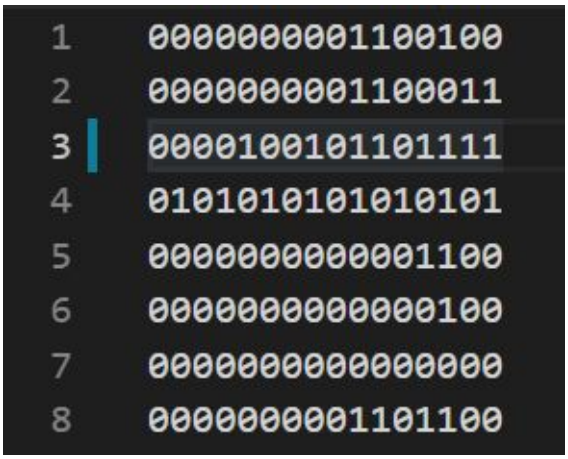
The what is occuring while program 1 is running. Also has the DIC

```
Registers: {'0': 100, '1': 99, '2': 0, '3': 0, '4': 0, '5': 0, '6': 0, '7': 0, 'p': 9}
Dynamic Instrcution Count: 109322
```

The what is occuring while program 2 is running. Also has the DIC

For Pattern B:

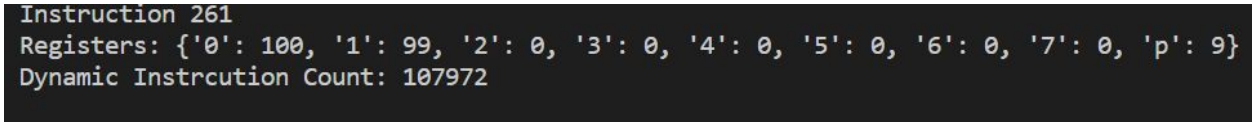




This is the output after Program 1 runs

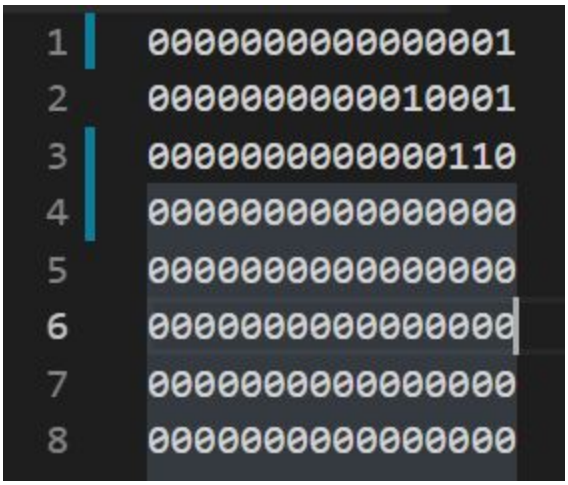
This is the output after Program 2 runs

This is the DIC for Program 1 and the result that is finally printed out and what is occurring during testing.

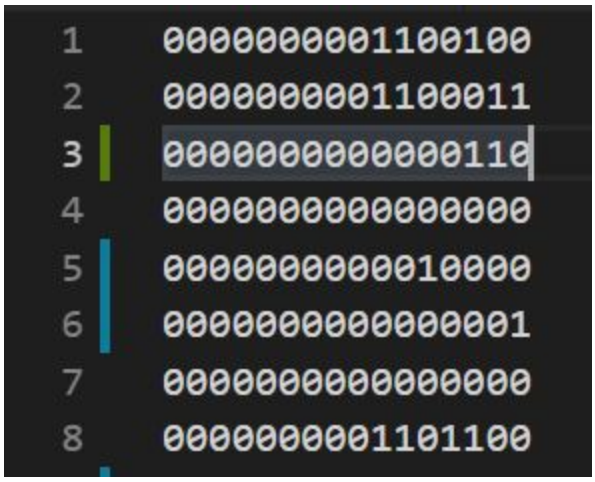


This is the DIC for Program 2 and the result that is finally printed out and what is occurring during testing.

For Pattern C:



This is the result after Program 1 runs



This is the output after Program 2 runs

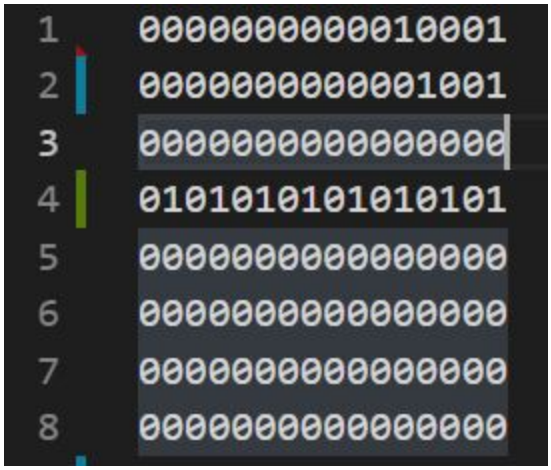
```
Instruction 63
Registers: {'0': 2, '1': 1, '2': 6, '3': 1, '4': 17, '5': 6, '6': 4, '7': 0, 'p': 4}
Dynamic Instrcution Count: 56
```

This is the DIC for Program 1 and the result that is finally printed out and what is occurring during testing.

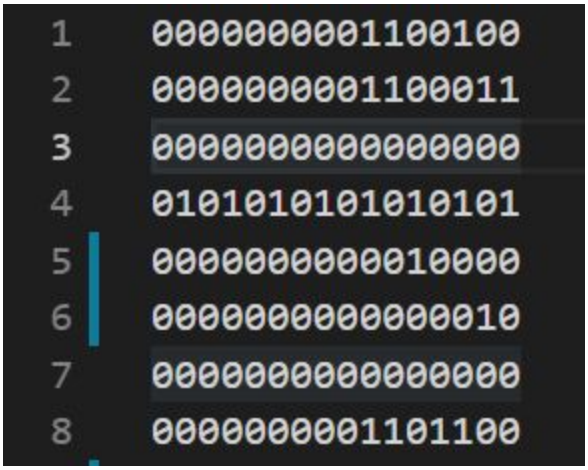
```
Instruction 261
Registers: {'0': 100, '1': 99, '2': 0, '3': 0, '4': 0, '5': 0, '6': 0, '7': 0, 'p': 9}
Dynamic Instrcution Count: 109346
```

This is the DIC for Program 2 and the result that is finally printed out and what is occurring during testing.

Pattern D:



This is the result after Program 1 runs



This is the output after Program 2 runs

```
Instruction 63
Registers: {'0': 2, '1': 17, '2': 0, '3': 17, '4': 9, '5': 6, '6': 4, '7': 16, 'p': 4}
Dynamic Instrcution Count: 624
```

This is the DIC for Program 1 and the result that is finally printed out and what is occurring during testing.

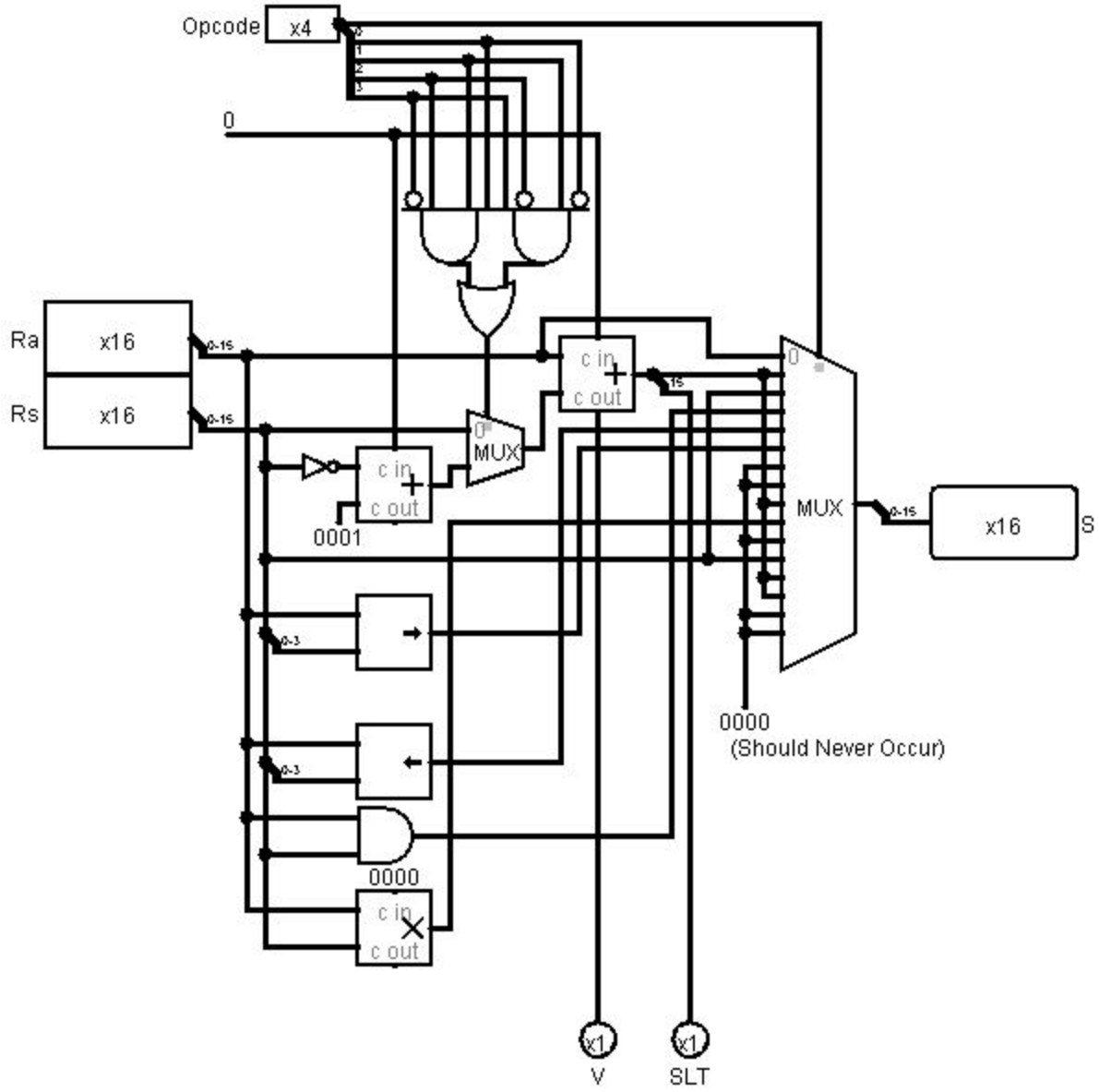
```
Instruction 70
i = 261
Instruction 261
Registers: {'0': 100, '1': 99, '2': 0, '3': 0, '4': 0, '5': 0, '6': 0, '7': 0, 'p': 9}
Dynamic Instrcution Count: 108080
```

This is the DIC for Program 2 and the result that is finally printed out and what is occurring during testing.

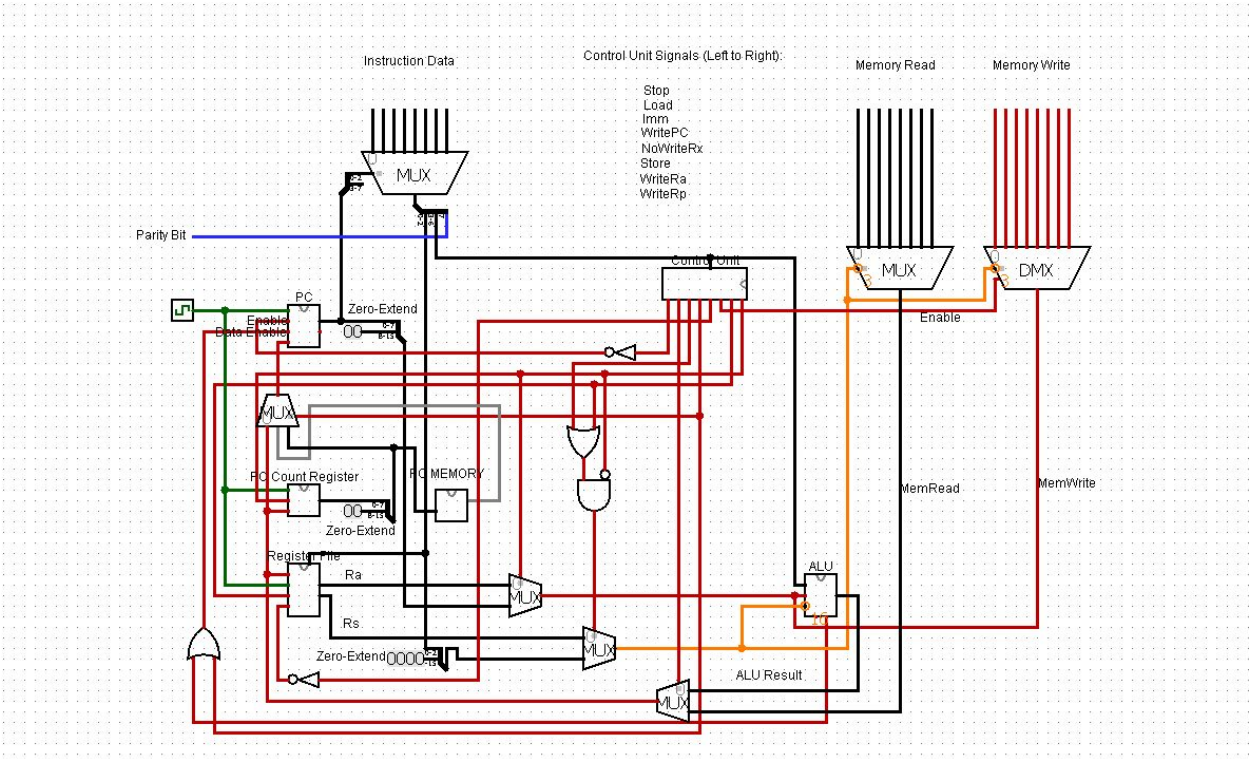
Part D:
Algorithms:
Check GitHub they are listed as Program1.asm and Program2.asm. They are commented as well.
Machine Code:
They are attached on Blackboard but can be seen on github listed as ProgramX_binary.txt
Data Mem:
They are on Github listed as patternC.txt and patternD.txt respectively.
Python Simulator:
Attached to blackboard submission but is not the entire simulator. To completely check simulator. Please go the the gitHub and see instructions on the README.md on how to the run the simulator properly.

HW:

1. ALU Schematic



2. CPU Datapath



3. Control Signal Truth Table

	Opcode	Imm	Load	NoSetRx	SAR	SetPC	Store	SetRp	Stop
SAR	0000	0	0	1	1	0	0	0	0
ADD	0001	0	0	0	0	0	0	0	0
MOV	0010	0	0	0	0	0	0	0	0
AND	0011	0	0	0	0	0	0	0	0
SLL	0100	0	0	0	0	0	0	0	0
SRL	0101	0	0	0	0	0	0	0	0
LOAD	0110	0	1	0	0	0	0	0	0
STOR	0111	0	0	1	0	0	1	0	0
SUB	1000	0	0	0	0	0	0	0	0
MLT	1001	0	0	0	0	0	0	0	0
BSLT	1010	0	0	1	0	1	0	0	0
MOVI	1011	1	0	0	0	0	0	0	0
ADDI	1100	1	0	0	0	0	0	0	0
SPC	1101	1	0	1	0	0	0	1	0
B	1110	0	0	1	0	1	0	0	0
STOP	1111	X	X	X	X	X	X	X	1