Project 2: ISA Design

ECE 366 - FA18
Professor Rao
October 17th, 2018

Singee Nguyen - snguye38
Jenshin Chen - chen172
Imran Babar - ibabar2

ISA Design:

- We're supporting 8 registers, R0-R6 and a flag register, R7
- R0 is read only, and always set to 0
- All other registers are both read and write
- R1-R4 are our general data registers
- R5 and R6 are our pointer registers, our instructions that access memory will specifically
- reference these two pointers to know which address to access
- We also have a flag register, R7, for our comparison instruction (SLT, we're just going to borrow from MIPS for this)
- All instructions besides B will simply make pc = pc + 1
- We also have room to sneak in an branch instruction that branches if R6 == 0 if we combine AND and XOR

"<u>Give us full points for our pointers</u>"

Instruction	**Details**
B #i	machine code: 000 iiii If R6==1, pc = pc + imm, supports imm range: [-8, 7] else pc = pc + 1 We're going to use imm == 0 as our halt instruction
ADD Rx, Ry	machine code: 001 xxyy Functionality: Rx = Rx + Ry Supports registers [1,4]
ADDI Rx, #i	machine code: 010 xxii Supports registers [1, 4] Supports imm range: [-2, 1], although excluding 0 if (#i == 0), then Rx = 0 Rx = Rx + #i
SUB Rx, Ry	machine code: 011 xxyy Supports registers [0, 3] Rx = Rx - Ry
SLT Rx, Ry	machine code: 100 xxyy Supports registers [0, 3] If Rx < Ry, then R6 = 1 else aka Rx >= Ry, then R6 = 0

	Note: This is signed comparsion so we can just follow the template provided in class
XOR Rx, Ry	machine code: 101 xxyy Supports registers [0,3] Does bit-wise XOR between Rx and Ry and stores the results in Rx Remember that R0 is read-only I think we can combine XOR and AND because we don't really need them to support more than two registers right?
AND Rx, Ry	machine code: 110 xxyy Does bit-wise AND between Rx and Ry and stores the results in Rx Supports registers [0,3]

STORE Rx, [Rp]	machine code: 111 10xp
	Supports R2, R3
	if Rx == R2, then x = 0
	else if Rx == R3, then x = 1
	if Rp == R5, then p = 0
	else if Rp = R6, then p = 1
	Rx = value at memory address of the value in Rp

LOAD Rx, [Rp]	machine code: 111 11xp
	Supports R2, R3
	if Rx == R2, then x = 0
	else if Rx == R3, then x = 1
	if Rp = R5, then p = 0
	else if Rp = R6, then p = 1
	Rx = memory_data[Rp]

AddPtr Rp, #n	machine code: 111 00pi
	Supports R5, R6
	if Rp == R5, then p = 0
	else if Rp == R6, then p = 1
	if n == -1, then i = 0
	else if n == 1, then i = 1
	Rp = Rp + n

ResetPtr Rp, #f	machine code: 111 01pf f will represent free variable Supports R5, R6 if Rp == R5, then p = 0 else if, Rp == R6, then p = 1

	Rp = 0 Doesn't really matter what the last bit is actually, we could sneak in another instruction here if we wanted to.

Part B:

1. Comparing to the sample of "My_straightforward_ISA", what are the unique features of your ISA? Explain why your ISA is better.

To begin with, Our ISA only supports 7 bits, whereas My_straightforward_ISA supports 8 bits. Our ISA is much more efficient because it takes less memory. Our ISA is much more restrictive, but is also flexible in its use. Specifically, we have two instruction that can manipulate 2 registers we can use as pointers to read from and write into memory. AddPtr is able to increment and decrement register R5 or R6, and although it is only able to move 1 at a time, we have the ability to have 2 addresses subsequently. ResetPtr is useful to return to address 0, where important values can be easily accessed. Load and Store use R5 and R6 specifically to perform their functions, making the instructions very powerful when used in conjunction with each other.

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

We didn't really try to minimize the dynamic instruction count in favor of easier coding and low cost in hardware. We kept our ISA relatively simple which limited how low we could keep our instruction count (a lot of linked branches), but this meant that our instructions wouldn't need complicated circuits to achieve their functions. We were able to essentially model our hardware off of proven hardware design, albeit with modifications that we had to make to adjust for our other constraints.

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

If we had one more bit for instructions, we probably would have had used more registers because that would have made the coding part a lot easier for us. We had very limited choices in our ISA and often resorted to using memory to consistently alter values as "makeshift registers".

If we had 1 fewer bit then it would require a lot more optimization because that would seriously hinder our instructions because we wouldn't be able to reduce our number of registers easily. Many instructions would have to be limited to accessing half as many registers as before with 1 fewer bit. (e.g. if our ADD instruction opcode got changed to 001xxy instead of 001xxyy, Rx would range from R1-R4, while Ry would be limited to R1-R2 or R3-R4)

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

We basically sat down together in the library and worked on ISA design and the programs together. We started with a base template for our ISA, and did rough drafts on our programs. This allowed us to see whether or not our program was reasonably achievable given the ISA, and we made some necessary adjustments to either make things easier or possible. This gave us a final draft of the ISA which allowed us to work on the hardware design. For the hardware design, we drew out the main components of the hardware design (register files, data memory, instruction memory and ALU) then added the necessary auxiliary components (logic gates, mux gates, adders, etc.) to make sure the correct information was transmitted in the circuit. We discussed what components were necessary, then split off and designed the finer details of each components. Everything was combined at the end, and then a truth table assignment was finalized. After this, we met up one last time to finish the report and finalize the code for the two programs. Most of the work was done together, although there was some side-work that we did individually.

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

Definitely start earlier, but in terms of design, we definitely would have supported the BNLT instruction that is suggested in the ISA design. Also, there would be some more optimization of the code, especially the modulus portion which we just brute forced.

Part C:
<u>Algorithms (in assembly code) of the two programs.</u>

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

The estimate for the DIC is 4153, obviously with some error.

1000 in binary is 11 1110 1000   This will be relevant in a bit

We start off with 17 lines to initialize all the values, then we started the code to find all the mod values that we needed to store into memory, which were a total of 16 since P is a 16 bit unsigned integer. The values were calculated by hand real briefly, and a pattern of repetition was recognized for every 4 binary digits. The mod values ended up being 6, 36, 46, and 16 repeatedly. Since our way of finding the mod value was just repeated addition, we calculated how many lines it'd take for each subtraction, and each mod run takes 6 instructions unless we exit, which takes 7. In addition, our multiplication is just repeated addition, so each iteration of the addition loop is 3 instructions. We needed to calculate these two sets of values four times.

The first set of values is the repeat multiplication set, and before each repeated addition loop, there’s 7 lines of set up work, so (4*7 + 3*6 + 3*36 + 3*46 + 3*16) = 340, which we multiply by 4 to get 1360 lines for calculating the values that we need to mod. To calculate the mod values that we needed, which were 6^2, 36^2, 46^2, and 16^2, it took 6, 150, 252, and 30 lines of code for each value respectively, which when multiplied by 4 yielded 1752. This portion of the code took a total of ~3129 dic, and we essentially begin with a blank slate (besides the values in memory).

Afterwards, it took 14 lines to reset all the registers to the correct values, and then we had to check which digits of P were a 1 to determine which values to pull from register. Each check took 12 lines, thus a total of 12 * 16 lines. When it came to the final set multiplication and mods, the first run took 10 lines, then run afterwards was 11 + multiplication + mod. The values we had to mod this time were 16*36, 26*46, 46*16, 36*6, and 16*36, and they took 11, 23, 14, 4, and 11 subtractions respectively. For the multiplication, our total multiplication was 16*36*46*16*6*36. Thus, for the total DIC of this portion, we got 14 + 12*16 for initializing and checking 0s or 1s, then 10 + 11*6 + 11*5 + 23*5 + 14*5 + 4*5 + 11*5 for the multiplication, then 36*3 + 46*3 + 16*3 + 6*3 + 36*3, and then +13 lines to finish everything. This sums to about 1017 DIC for this portion, and we summed this value with the earlier one to get ~4153 DIC.

Program 1 Asm:

```
; Program 1 is suppose to calculate
(6^P) % Q where P and Q are 16 bit
unsigned integers stored

                    ; at memory address 0 and 1

                    AddPtr R6, #1

                    AddPtr R6, #1                ;This makes R6 into 2
                    which will be where we store a counter value,

                                            ;although we will eventually
                    write the results at this memory address

                    Addi R3, #1
```

```
Add R3, R3          ; R3 == 2

Add R3, R3          ; R3 == 4

Add R3, R3          ; R3 == 8

Add R3, R3          ; R3 == 16

Addi R3, #-1        ;R3 == 15

Store R3, [R6]

AddPtr R6, #-1


Addi R2, #1

Addi R2, #1

Addi R2, #1         ; R2 == 3

Add R2, R2          ; R2 == 6


;Addi R1, 1

;Addi R1, 1

;Addi R1, 1

;Add R1, R1         ;R1 == 6


AddPtr R5, #1
AddPtr R5, #1

AddPtr R5, #1           ;R5 now points to the array
that will hold the mod values we need



; The next block of code will find R2 mod R3 and
store the value in R2, note that R3 == Q for this part
of the code

Load R3, [R6]           ;R6 == 1, R3 == Q

SLT R2, R3

B #4                    ;exits when R2 < Q

Sub R2, R3

SLT R0, R3              ;Essentially a force branch

B #-5                   ;Branch for the mod, and it's
being used as part of Linked branch, set 1

SLT R0, R0              ;resets flag register to 0



;Storing the mod value into memory in the order that
we want

Store R2, [R5]

AddPtr R5, #1           ;Moves down the array



;This next block calculates the next value that we
need to mod to get a "mod value" to store in the array
```

```
Addi R1, #0

Add R3, R2              ;R3 = R2, and R3 will be used
as a counter again, although this is a different value
and will not need to be stored in memory

B #-6                   ;Linked branch, set 1

Addi R3, #-2            ;We need to decrement it by 2
though due to 5 * 5 only having 4 plus signs and how
our SLT works

Add R1, R2              ;R1 is acting as an intermediate
here; it's the number that we're going to repeatedly
add to do our multiplication

Add R2, R1

Addi R3, #-1            ;Decrement counterB

SLT R0, R3              ;Branches if R3 >= 0

B #-3

AddPtr R6, #1           ;R6 == 2 since counter is at
Mem[2]
B #-8                   ;linked branch, set 1

Load R3, [R6]

Addi R3, #-1            ;decrement counterA

Store R3, [R6]

AddPtr R6, #-1                    ;R6 == 1

SLT R3, R0              ;Branches out if counterA < 0
```

```
B #3

SLT R0, R2              ;This happens when R3 < 0

B #-8                   ;Repeat the mod process;
linked branch, set 1


;Clean slate!!!

Addi R1, #0



Addi R2, #0

Addi R2, #1



Addi R4, #0    ;I should have used R4 earlier as my
counter

Addi R4, #1

Addi R4, R4    ;R4 == 2

Addi R4, R4    ;R4 == 4

Addi R4, R4    ;R4 == 8

Addi R4, R4    ;R4 == 16

Addi R4, #-1 ;R4 == 15



ResetPtr R5
ResetPtr R6

AddPtr R5, #1

AddPtr R5, #1 ;R5 == 3



;This block of code will process P to see which bits
of its binary representation is a 1, later blocks will
multiplies the values as necessary

Load R3, [R6] ;R3 == P

AddPtr R6, #1

AddPtr R6, #1 ;R6 == 2

Load R2, [R6]

B #-4            ;linked branch set 2

And R3, R2

Addi R3, #-1   ;This is needed to make this SLT logic
work out, otherwise, R2 always >= 0

Addi R4, #-1

AddPtr R5, #1

Add R2, R2     ;"left shift" R2

SLT R3, R0

B #-7   ;R3 == 0 before I did the subtraction, thus
there was a 0 at this digit; linked branch set 2
```

```
;Note the value in R1 should be the result of all the multiplication that we need to do

;Its value will need to be shifted back into one of the registers that have access to memory after everything is said and done

;This had to happen though in order to preserve some of the values that would be used.

;Stores the bitchecker value

Store R2, [R6]

ResetPtr R6

Addi R2, #0

B #-4           ;Linked branch set 3, although this is going to be linked to set 2 from this point above

Load R3, [R5]

SLT R0, R1      ;This will only evaluate to false during the first run that we found a 1

B #4

Add R1, R3      ;This is initializing R1 since it should be zero when this line executes

SLT R1, R0      ;force branch back to the up to start of checking digits

B #-6           ;Linked branch set 3
```

```
;Start of the repeated addition loop

Add R2, R1

Addi R3, #-2

B #-3               ;Linked branch set 4, although it will
be linked to branch set 3 from now on

Add R1, R2

Addi R3, #-1

SLT R0, R3

B #-3

SLT R4, R0                  ;branch out when R4 is
negative
B 3

SLT R0, R1

B #-8               ;Linked branch set 4



Addi R2, #0

Add R2, R1

Addi R1, #0

ResetPtr R5

AddPtr R5, #1
```

```
Load R3, [R5]

AddPtr R5, #1

SLT R2, R3

B #4                    ;exits when R2 < Q

Sub R2, R3

SLT R0, R3              ;Essentially a force branch

B #-4                   ;Branch for the mod, and it's
being used as part of Linked branch, set 1

Store R2, [R5]          ;So...R2 should be the value
that I'm looking for now...and R2 == 2

B #0                    ;stall branch
```

__Program 2 ASM:__

```
; T is at Mem[3]

; Array goes from Mem[8-107]

; Store best match score at Mem[4]

; Store count of best match score at Mem[5]

; Mem [0-2], Mem[6], Mem[7] are available to us for manipulation.



;****************************INITIALIZATIONS*********************************

Addi    R2, #1

Addi    R1, #1

Addi    R2, 1
```

```
Add     $2, $2          ;R2 = 2

Add     $2, $2

Add     $2, $2          ;R2 = 8

Add     $2, $2

Add     $3, $2          ;R3 = (R2 = 16)**

Add     $3, $3          ;R3 = 32

Add     $3, $2          ;R3 = 48

Addi    $3, 1

Addi    $3, 1           ;R3 = 50

Add     $3, $3          ;(R3 = 100)**


ResetPtr $5

Store   $2, [$5

]       ;MEM[0] = 16, used to calculate BMS and to reset loopCtr

AddPtr  $5, 1


Store   $3, [$5]        ;MEM[1] = 100 = arrayCtr
```

```
AddPtr	$6, 1

AddPtr	$6, 1

AddPtr		$6, 1

AddPtr		$6, 1

AddPtr		$6, 1

AddPtr		$6, 1

AddPtr		$6, 1

AddPtr		$6, 1		;R6 = 8, and MEM[8] = Pattern_Array[1] initialization

;*************************************************************************

	SLT	$0, $2		;skip the BMS update and BMS check function on first
iteration

	B	7

;---------------------updateBMS Function---------------------------

ResetPtr $5

AddPtr	$5, 1

AddPtr	$5, 1

AddPtr	$5, 1

AddPtr	$5, 1
```

```
        SLT     $0, $0  ;set up branch link

        B       7       ;link to nextBit

        B       -7      ;link to updateBMS

AddPtr  $5, 1   ;R5 = 5, MEM[5] = BMC

Addi    $2, 0   ;reset R2 to 0

Addi    $2, 1   ;R2 = 1

Store   $2, [$5];when BMS is updated, BMC must be reset to 1


;-----------------------------------------------------------------

        SLT     $0, $0  ;set up branch link

        B       6       ;skip to nextBit

        B       -7      ;link to updateBMS

;********************checkBMS
Function*********************************
ResetPtr $5

Load    $2, [$5]        ;R2 = 16

AddPtr  $5, 1


SLT     $0, $0  ;set up branch link

        B       7       ;skip to nextBit
        B       -6      ;link to updateBMS

        B       -6      ;link to checkBMS

AddPtr $5, 1            ;R5 = 2

Load    $3, [$5]        ;R3 = MEM[2] = mismatchCtr

Sub     $2, $3          ;R2 = 16 - mismatchCtr = Current Match Score, CMS

        SLT     $0, $0  ;set up branch link

        B       3       ;skip to nextBit

        B       -7      ;link to updateBMS

        B       -7      ;link to checkBMS

        B       7       ;skip to nextBit

AddPtr          $5, 1

AddPtr          $5, 1           ;R5 = 4

Load    $3, [$5]        ;R3 = MEM[4] = Best Matching Score, BMS

SLT     $3, $2          ;when BMS < CMS, R7 = 1

B       -7              ;link to update BMS

        B       -7      ;link to checkBMS

        B       5       ;initial skip to nextBit
```

```
;*****************************************************************
*

;---------------------BMC++ Function------------------------------

Sub     $2, $3          ;R2 = CMS - BMS

SLT     $2, $0          ;if result is negative, branch forward

B       7               ;if result is zero, increment BMC (R2 cannot be positive
because we established that BMS > CMS in the SLT above)

        B       -5              ;link to checkBMS

        B       7               ;skip to nextBit

AddPtr  $5, 1           ;R5 = 5

Load    $2, [$5]        ;R2 = Best Match Counter

Addi    $2, 1           ;BMC++

Store   $2, [$5]


;------------------------------------------------------------------


        SLT     $0, $0  ;set up branch link

        B       -7      ;link to checkBMS


;********************nextBit Function*******************************
```

```
ResetPtr $5             ;R5 = 0
Load    $3, [$5
]       ;R3 = 16
Add     $4, $3          ;R4 = 16 = loopCtr
AddPtr  $5, 1
        SLT     $0, $0  ;set up branch link
        B       -6      ;link to checkBMS
        B       -6      ;link to nextBit
Load    $3, [$5
]
Addi    $3, -1          ;MEM[1]-- = arrayCtr--
Store   $3, [$5
]
AddPtr  $5, 1
Addi    $2, 0           ;reset R2 to 0
        SLT     $0, $0  ;set up branch link
        B       -8      ;link to checkBMS
        B       -8      ;link to nextBit
```

```
Store   $2, [$5

]       ;reset mismatchCtr : MEM[2] = R2 = 0

Addi    $1, 1           ;(re)initialize the bitChecker


Addi    $4, -1          ;loopCtr--

SLT     $4, $0          ;set branch flag when all bits have been checked

B       -6              ;link to checkBMS

Load    $2, [$5

]       ;R2 = Target Pattern

        SLT     $0, $0  ;set up branch link

        B       -8      ;

Load    $3, [$6]        ;R3 = Pattern_Array[i], where i = ($6-8)

XOR     $3, $2          ;compare P_A with T

AND     $3, $1          ;AND with bitChecker yields result in R3

SLT     $0, $3          ;R7 = 1 if there is a mismatch

B       4

Addi    $1, $1          ;bitChecker LSL 1

SLT     $0, $2          ;force branch condition R7 = 1
```

```
B       -8

Add     $1, $1          ;bitChecker LSL 1

AddPtr  $5, -1

Load    $3, [$5]

Addi    $3, 1           ;MEM[2]++ (mismatch Counter)

Store   $3, [$5]

AddPtr  $5, 1

SLT     $0, $2          ;force branch condition

B       -8

;*****************************************************************
```

Program 1 Machine Code:

```
11100110
11100110
11100110
01011010
00111110
00111110
00111110
00111110
01011110
11110110
11100100
01001011
01010011
01010011
00110101
```

```
11100010
11100010
11100010
11111111
10010110
00001000
01110110
10000110
00001010
10000001
11110000
11100010
01001000
00111100
00010100
01011100
00101101
00110011
01011110
10001101
00011011
11100111
00010001
11111111
01011111
11110110
11100100
10011001
00000110
10000100
00010001
01001000
01010000
01010011
01000001
01000010
00100001
00100001
00100001
00100001
```

```
01000111
11101001
11101101
11100010
11100010
11111111
11100110
11100110
11111010
00011000
11011101
01011111
01000010
11100010
00110101
10011001
00010010
11110011
11101101
01010000
00011000
11111100
10000010
00001001
00101110
10001000
00010100
00110011
01011100
00011011
00101101
01011111
10000111
00011011
10011001
00000110
10000010
00010001
01010000
00110011
```

```
01001000
11101011
11100010
11111100
11100010
10010110
00001001
01110111
10000111
00011000
11110011
00000000
```

Program 2 Machine Code:

```
01010011

01001011

01010011

00110101

00110101

00110101

00110101

00111100

00111111

00111100

01011010

01011010

00111111

11101110

11110000
```

11100010

11110101

11100111

11100111

11100111

11100111

11100111

11100111

11100111

11100111

10000100

00001111

11101110

11100010

11100010

11100010

11100010

10000001

00001111

00010010

11100010

01010000

01010011

11110000

11101110

11111001

11100010

10000001

00001111

00010100

00010100

11100010

11111100

01110111

10000001

00000110

00010010

00010010

00001111

11100010

11100010

11111100

10011100

00010010

00010010

00001010

01110111

10010000

00001111

00010111

00001111

11100010

11111001

01010011

11110000

10000001

00010010

11101110

11111100

00100111

11100010

10000001

00010100

00010100

11111100

01011111

11110101

11100010

01010000

10000001

00010001

00010001

11110000

01001010

01000111

10011001

00010100

11111001

10000001

00010001

11111111

10111101

11011011

10000111

00001001

01001011

10000100

00010001

00101011

11100000

11111100

01011010

11110101

11100010

10000100

00010001

Output of Python disassembler for Program 1:

```
ADDPTR $6, #1
ADDPTR $6, #1
ADDPTR $6, #1
ADDI $3, 1
ADD $3, $3
ADD $3, $3
ADD $3, $3
ADD $3, $3
ADDI $3, -1
STORE $3, [$6]
ADDPTR $6, #-1
ADDI $1, 1
ADDI $2, 1
ADDI $2, 1
ADD $2, $2
ADDPTR $5, #1
ADDPTR $5, #1
ADDPTR $5, #1
LOAD $3, [$6]
SLT $2, $3
B #4
SUB $2, $3
SLT $0, $3
B #5
SLT $0, $0
STORE $2, [$5]
ADDPTR $5, #1
ADDI $1, 0
ADD $3, $2
B #-6
ADDI $3, -2
ADD $1, $2
ADD $2, $1
ADDI $3, -1
SLT $1, $2
B #-3
ADDPTR $6, #1
B #-8
LOAD $3, [$6]
ADDI $3, -1
STORE $3, [$6]
ADDPTR $6, #-1
```

```
SLT $3, $0
B #3
SLT $0, $2
B #-8
ADDI $1, 0
ADDI $2, 0
ADDI $2, 1
ADDI $4, 0
ADDI $4, 1
ADD $4, $4
ADD $4, $4
ADD $4, $4
ADD $4, $4
ADDI $4, -1
RESETPTR $5
RESETPTR $6
ADDPTR $5, #1
ADDPTR $5, #1
LOAD $3, [$6]
ADDPTR $6, #1
ADDPTR $6, #1
LOAD $2, [$6]
B #-4
AND $3, $2
ADDI $3, -1
ADDI $4, 1
ADDPTR $5, #1
ADD $2, $2
SLT $3, $0
B #-7
STORE $2, [$6]
RESETPTR $6
ADDI $2, 0
B #-4
LOAD $3, [$5]
```

```
SLT $0, $1
B #4
ADD $1, $3
SLT $1, $0
B #-6
ADD $2, $1
ADDI $3, -2
B #-3
ADD $1, $2
ADDI $3, -1
SLT $0, $3
B #-3
SLT $3, $0
B #3
SLT $0, $1
B #-8
ADDI $2, 0
ADD $2, $1
ADDI $1, 0
RESETPTR $5
ADDPTR $5, #1
LOAD $3, [$5]
ADDPTR $5, #1
SLT $2, $3
B #4
SUB $2, $3
SLT $0, $3
B #-4
STORE $2, [$6]
B #0
```

Output of Python disassembler for Program 2:

```
ADDI $2, 1
ADDI $1, 1
ADDI $2, 1
ADD $2, $2
ADD $2, $2
ADD $2, $2
ADD $2, $2
ADD $3, $2
ADD $3, $3
ADD $3, $2
ADDI $3, 1
ADDI $3, 1
ADD $3, $3
RESETPTR $5
STORE $2, [$5]
ADDPTR $5, #1
STORE $3, [$5]
ADDPTR $6, #1
ADDPTR $6, #1
ADDPTR $6, #1
ADDPTR $6, #1
ADDPTR $6, #1
ADDPTR $6, #1
ADDPTR $6, #1
ADDPTR $6, #1
SLT $0, $2
B #7
RESETPTR $5
ADDPTR $5, #1
ADDPTR $5, #1
ADDPTR $5, #1
ADDPTR $5, #1
SLT $0, $0
```

```
B #7
B #-7
ADDPTR $5, #1
ADDI $2, 0
ADDI $2, 1
STORE $2, [$5]
RESETPTR $5
LOAD $2, [$5]
ADDPTR $5, #1
SLT $0, $0
B #7
B #-6
B #-6
ADDPTR $5, #1
LOAD $3, [$5]
SUB $2, $3
SLT $0, $0
B #3
B #-7
B #-7
B #7
ADDPTR $5, #1
ADDPTR $5, #1
LOAD $3, [$5]
SLT $3, $2
B #-7
B #-7
B #5
SUB $2, $3
SLT $2, $0
B #7
B #-5
B #7
ADDPTR $5, #1
LOAD $2, [$5]
```

```
ADDI $2, 1
STORE $2, [$5]
SLT $0, $0
B #-7
RESETPTR $5
LOAD $3, [$5]
ADD $4, $3
ADDPTR $5, #1
SLT $0, $0
B #-6
B #-6
LOAD $3, [$5]
ADDI $3, -1
STORE $3, [$5]
ADDPTR $5, #1
ADDI $2, 0
SLT $0, $0
B #-8
B #-8
STORE $2, [$5]
ADDI $1, 1
ADDI $4, -1
SLT $3, $0
B #-6
LOAD $2, [$5]
SLT $0, $0
B #-8
LOAD $3, [$6]
XOR $3, $2
AND $3, $1
SLT $0, $3
B #4
ADDI $1, 1
SLT $0, $2
B #-8
```

```
ADD $1, $1
ADDPTR $5, #-1
LOAD $3, [$5]
ADDI $3, 1
STORE $3, [$5]
ADDPTR $5, #1
SLT $0, $2
B #-8
```

Python code for ISA’s disassembler:

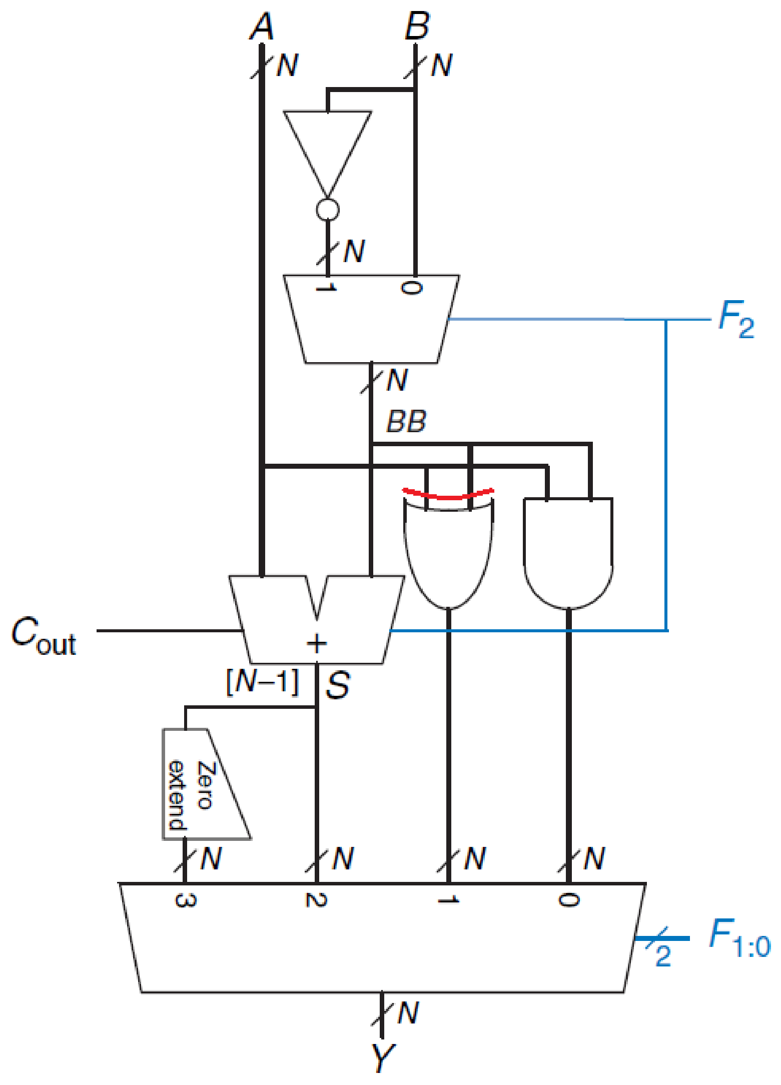
```
# ECE 366 Project 2 Disassembler

# This file disassembles two project files

p1_input_file = open("Program1_Bin.txt", "r")
```

```
p1_output_file = open("Program1_Asm.txt", "w")

p2_input_file = open("Program2_Bin.txt", "r")
p2_output_file = open("Program2_Asm.txt", "w")


def convert_bin_to_asm(input_file, output_file):
    for line in input_file:

        if line == "\n":  # empty lines ignored
            continue
        line = line.replace("\n", "")  # remove
'endline' character
        print("Machine Instr: ", line)  # show the asm
instruction to screen


        if line[0:3] == '001':  # add instruction
            op = line[0:3]
            rx = line[3:5]
            ry = line[5:7]
            parity = line[7:8]
            if line[3:5] == '01':
                c = "1"
            elif line[3:5] == '10':
                c = "2"
            elif line[3:5] == '11':
                c = "3"
            else:
                c = "4"
            if line[5:7] == '01':
                d = "1"
            elif line[5:7] == '10':
                d = "2"
            elif line[5:7] == '11':
                d = "3"
            else:
                d = "4"
            a= "ADD"
            output_file.write(str(a) + " " + '$' +
str(c) + "," + " " + '$' + str(d) + "\n")
        if line[0:3] == '011':  # sub instruction
            op = line[0:3]
            rx = line[3:5]
            ry = line[5:7]
            parity = line[7:8]

            c = int(rx, 2)
            d = int(ry, 2)
            a = "SUB"
            output_file.write(str(a) + " " + '$' +
str(c) + "," + " " + '$' + str(d) + "\n")
        if line[0:3] == '000':  # Branch instruction
           op = line[0:3]
           imm= line[3:7]
           parity = line[7:8]
           if line[3:7] == '0000':
               c = "0"
           elif line[3:7] == '0001':
               c = "1"
           elif line[3:7] == '0010':
               c = "2"
           elif line[3:7] == '0011':
               c = "3"
           elif line[3:7] == '0100':
               c = "4"
           elif line[3:7] == '0101':
               c = "5"
           elif line[3:7] == '0110':
               c = "6"
           elif line[3:7] == '0111':
               c = "7"
           elif line[3:7] == '1111':
               c = "-1"
           elif line[3:7] == '1110':
               c = "-2"
           elif line[3:7] == '1101':
               c = "-3"
           elif line[3:7] == '1100':
               c = "-4"
           elif line[3:7] == '1011':
               c = "-5"
           elif line[3:7] == '1010':
               c = "-6"
           elif line[3:7] == '1001':
               c = "-7"
           else:
               c = "-8"

           a = "B"
           output_file.write(str(a) + " " + "#"+ str(c)
+ "\n")

        if line[0:3] == '010':  # ADDI instruction
           op = line[0:3]
           rx = line[3:5]
           imm = line[5:7]
           parity = line[7:8]
           if line[3:5] == '01':
                c = "1"
           elif line[3:5] == '10':
                c = "2"
           elif line[3:5] == '11':
                c = "3"
           else:
                c = "4"
           if line[5:7] == '01':
                d = "1"
           elif line[5:7] == '10':
                d = "-2"
           elif line[5:7] == '11':
                d = "-1"
           else:
                d = "0"
           a = "ADDI"
           output_file.write(str(a) + " " + '$' + str(c)
+ "," + " " + str(d) + "\n")
        if line[0:3] == '100':  # SLT instruction
            op = line[0:3]
            rx = line[3:5]
            ry = line[5:7]
            parity = line[7:8]

            c = int(rx, 2)
            d = int(ry, 2)
            a = "SLT"
            output_file.write(str(a) + " " + '$' +
str(c) + "," + " " + '$' + str(d) + "\n")
        if line[0:3] == '101':  # XOR instruction
            op = line[0:3]
            rx = line[3:5]
            ry = line[5:7]
            parity = line[7:8]

            c = int(rx, 2)
            d = int(ry, 2)
            a = "XOR"
            output_file.write(str(a) + " " + '$' +
str(c) + "," + " " + '$' + str(d) + "\n")
        if line[0:3] == '110':  # AND instruction
```

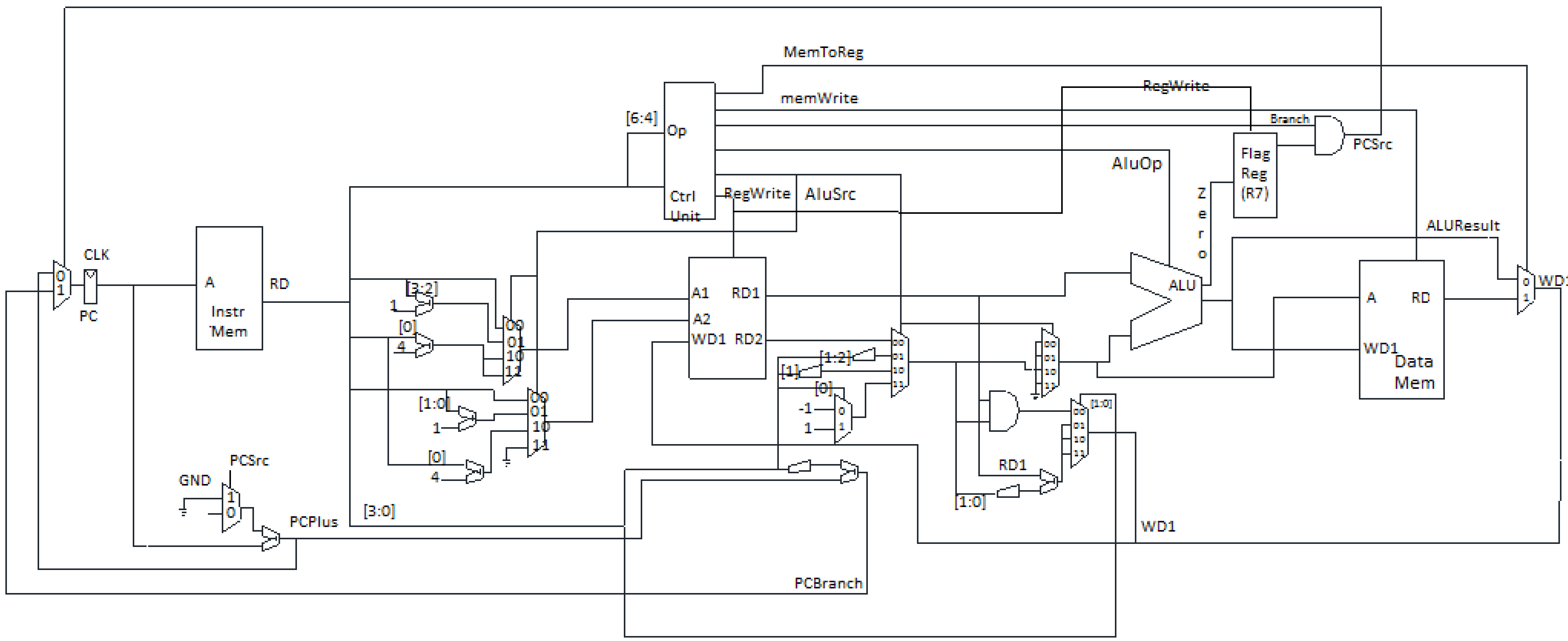
```
            op = line[0:3]
            rx = line[3:5]
            ry = line[5:7]
            parity = line[7:8]

            c = int(rx, 2)
            d = int(ry, 2)
            a = "AND"
            output_file.write(str(a) + " " + '$' +
str(c) + "," + " " + '$' + str(d) + "\n")
        if line[0:5] == '11110':  # Store instruction
            op = line[0:5]
            rx = line[5:6]
            imm = line[6:7]
            parity = line[7:8]
            if line[5:6] == '0':
                c = "2"
            else:
                c= "3"
            if line[6:7]== '0':
                p="$5"
            else:
                p="$6"



            a = "STORE"
            output_file.write(str(a) + " " + '$' +
str(c) + "," + " " + '['+ str(p)+ ']' + "\n")

        if line[0:5] == '11111':  # Load instruction
            op = line[0:5]
            rx = line[5:6]
            imm = line[6:7]
            parity = line[7:8]
```

```
           if line[5:6] == '0':
               c = "2"
           else:
               c= "3"
           if line[6:7]== '0':
               p="$5"
           else:
               p="$6"
           a = "LOAD"
           output_file.write(str(a) + " " + '$' +
str(c) + "," + " " + '[' + str(p) + ']' + "\n")

       if line[0:5] == '11100':  # ADDPTR instruction
           op = line[0:5]
           rp = line[5:6]
           p = line[6:7]
           parity = line[7:8]
           if line[5:6] == '0':
               c = "5"
           else:
               c= "6"
           if line[6:7]== '0':
               p="-1"
           else:
               p="1"
           a = "ADDPTR"
           output_file.write(str(a) + " " + '$' +
str(c) + "," + " " + '#' + str(p) +"\n")

       if line[0:5] == '11101':  # RESETPTR instruction
           op = line[0:5]
           rp = line[5:6]
           p = line[6:7]
           parity = line[7:8]
           if line[5:6] == '0':
               c = "5"
           else:
               c= "6"
           if line[6:7]== '0':
               p=""
           else:
               p=""
           a = "RESETPTR"
           output file.write(str(a) + " " + '$' +
str(c) + "" + " " + '' + str(p) +"\n")



convert bin to asm(p1 input file, p1 output file)
convert bin to asm(p2 input file, p2 output file)
```

ALU SCHEMATIC



CONTROL UNIT:


MemToReg
memWrite
RegWrite
Branch
PCSrc
[6:4]
Op
Ctrl Unit
RegWrite
AluSrc
AluOp
Flag Reg (R7)
Zero
ALU
ALUResult
WD1
CLK
PC
A
RD
Instr Mem
A1
RD1
A2
WD1
RD2
A
RD
WD1
Data Mem
[3:2]
1
[0]
4
00
01
10
11
[1:0]
1
[0]
4
[1:2]
[1]
[0]
-1
1
[1:0]
RD1
[1:0]
WD1
PCBranch
PCPlus
[3:0]
GND
PCSrc
0
1