M.U.S.H Architecture Design Document

Group 4-F

Jim Gildersleeve,Guilherme Simas, Brian Suchy, and Songyu Wang

This document outlines the essential information to using the M.U.S.H. architecture. Here, you can find information on commands, registers, syntax and anything else required to use this architecture.

2

R-Type Instruction Layout:

4	4	4	4
OP	RS	RT	RD

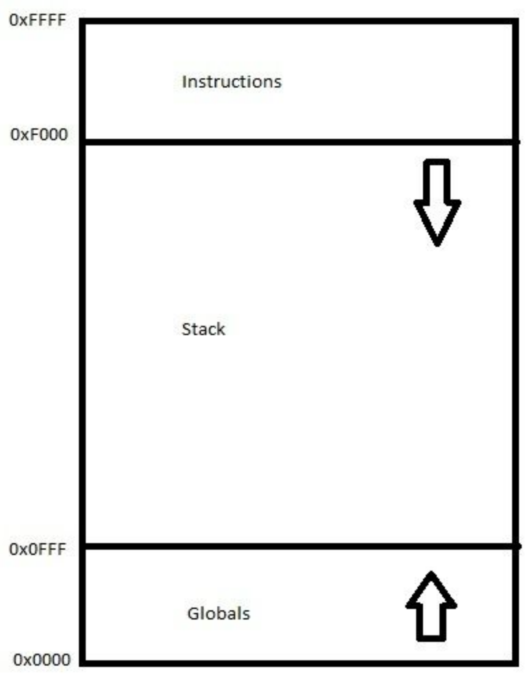
This instruction will be the bread and butter of our processor. All but one instruction will use solely
R-Type instructions. The R-Type instruction will have a 4-bit opcode and have 3 4-bit register addresses. The registers themselves will hold the jump locations and other necessary information as opposed to a J-Type or I-Type instructions which have immediates that have a limited range.

I-Type Instruction Layout:

4	12
OP	Immediate

This instruction will be used only for loading Immediates into a register. It will load the 12 bits into the least significant spots of the special $imm register.

Memory layout:



The following will be the registers available:

Id	Name	usage
0	$imm	implied register for the li command
1	$ret	used to store a return value from a function
2	$at	temporary value used only by assembler
3,4	$p0,$p1	two parameter registers
5-12	$t0-$t7	temporary values, not preserved across system calls
13	$ra	used by the ret and jal command to return after a function call
14,15	$s0,$s1	Saved temporaries, preserved across system calls
program counter	PC	unaddressable, used only to load instructions
stack pointer	SP	only used by push and pop to manipulate the stack
interrupt	int	special register used for interrupts such as input
compare value	cmp	implied register set by sub and ready by beq that holds a -1,0,1 to represent inequalities

I/O:
Input will be treated as an interrupt. The hardware will be responsible for detecting an input and allocating the data in a memory location reserved for that purpose only [The memory address will be decided later].
To output data the programmer must store the data in another memory location, also reserved for that purpose only. The hardware will be responsible for moving data from that memory location [to be decided] to the output port.

The following are syntax rules for the instructions:
// is a comment
Label use:

- use a single word with no spaces
- end label with a colon
- do not start label with a number

Assembly commands:

3opcode	assembly command	Instruction name	instruction type	Description
0x0	lw	load	R	RT = mem[RS]
0x1	sw	store	R	mem[RS] = RT
0x2	add	add	R	RD= RS + RT
0x3	j	jump	R	PC = {0xF,RS[11-0]}
0x4	beq	branch equal	R	if(RS==cmp) PC = {0xF,RT[11-0]}
0x5	sll	shift left logical	R	RD= RS << RT
0x6	srl	shift right logical	R	RD= RS >> RT
0x7	li	load immediate	I	$imm = { 4 { immediate [11] } , immediate }
0x8	and	and	R	RD= RS & RT
0x9	or	or	R	RD= RS | RT
~~0xA~~	~~ret~~	~~return~~	~~R~~	~~$PC = RA; RA = mem[$sp]; $sp = $sp + 2~~
0xB	sub	subtract	R	RD = RS - RT ; CMP={RS?RT->

				<=-1;==0;>=1}
0xC	jal	jump and link	R	RA = $PC + 2; $PC = RD
0xD	push	push	R	mem[$sp] = RT ; $sp = $sp - 2
0xE	pop	pop	R	$sp = $sp + 2; RD = mem[$sp]
0xF	mov	move	R	RD = RS

Example of Euclids algorithm with our architecture

address	assembly	machine code	C code
0xF000	li 0	7000	//t0=0
0xF002	mov $t0 $imm	F500	
0xF004	sub $t1 $t0 $p0	B653	//a==0?
0xF006	li 0x012	7012	//t2= address of if1
0xF008	mov $t2 $imm	F700	
0xF00A	beq $t0 $t2	4570	//if (a==0)True, go to if1
0xF00C	li 0x016	7016	//t1=address of while
0xF00E	mov $t2 $imm	F700	
0xF010	j $t2	3700	//if (a==0)False, go to while
	//return b		
	if1:		
0xF012	mov $ret $p1	F140	//prepare to return [return address(ret) = $p1(b)]
0xF014	j $ra	3D00	//return
	while:		//while
0xF016	sub $t1 $t0 $p1	B654	//b==0?

0xF018	li 0x040	7040	//t2 = address of returnA
0xF01A	mov $t2 $imm	F700	
0xF01C	beq $t0 $t2	4570	//if (b==0)True go to returnA
	//if (a>b)		
0xF01E	sub $t1 $p0 $p1	B634	//check a?b
0xF020	li 0x030	7030	//t2=address of if
0xF022	mov $t2 $imm	F700	
0xF024	li 1	7100	//t3=1
0xF026	mov $t3 $imm	F800	
0xF028	beq $t3 t2	4870	//if (a>b) go to if
0xF02A	li 0x038	7038	//t2=address of else
0xF02C	mov $t2 $imm	F700	
0xF02E	j $t2	3700	//jump to else
	//a=a-b		
	if:		
0xF030	sub $p0 $p0 $p1	B334	//a=a-b
0xF032	li 0x016	7016	//t2 =address of while
0xF034	mov $t2 $imm	F700	
0xF036	j $t2	3700	//goto while
	//b=b-a		
	else:		
0xF038	sub $p1 $p1 $p0	B443	//b=b-a
0xF03A	li 0x016	7016	//t2 =address of while
0xF03C	mov $t2 $imm	F700	

0xF03E	j $t2	3700	//goto while
	//return a		
	returnA:		
0xF040	j $ra	3D00	//return

Loading a 16-bit immediate (0x1234) into a register ($t0):

address	assembly	machine code	C code
0xF000	li 0x123	7123	Load upper 12bits
0xF002	mov $t0 $imm	F500	Move into register
0xF004	li 8	7800	load number of shifts
0xF006	sll $t0 $imm	5500	shift t0
0xF008	li 0x4	7400	load lower 4 bits
0xF00A	or $t0 $t0 $imm	9550	or them together

Simple “for” loop (for(i=0;i<[$t1];i++) [$t0]=[$t0]*2;)

0xF000	li 0	7000	
0xF002	mov $t2 $imm	f700	//i=0
	for:		
0xF004	sub $t5 $t2 $t1	ba76	
0xF006	li [block]	7012	
0xF008	mov $t4 $imm	f90q	
0xF00A	li -1	7fff	
0xF00C	beq $imm $t4	4090	
0xF00E	li [done]	701c	
0xF010	j $imm	3000	
	block:		

0xF012	li 1	7001	
0xF014	sll $t0 $t0 $imm	5550	
0xF016	add $t1 $t1 $imm	2660	
0xF018	li [for]	7004	
0xF01A	j $imm	3000	
	done:		
0xF01C	j $ra	3d00	

Accessing an indexed array element (A[4] when A is an array of 16 bit elements and the first element is located at 0x1234) :

address	assembly	machine code	C code
0xF000	/*load 0x1234 in $t0*/	See example above	See example above
0xF002	li 4	7004	load index
0xF004	mov $t1 $imm	F600	$t1 = index
0xF006	li 1	7001	prepare to shift
0xF008	sll $t1 $t1 $imm	5660	shift index once
0xF00A	add $t1 $t1 $t0	2665	add base address to index
0xF00C	lw $t1 $t1	0660	load the element

Machine Code:
7000F500B6537012F70045707016F7003700F1403D00B6547040F7004570B6347030F700700
1F80048707038F7003700B3347016F7003700B4437016F70037003D00 :)

RTL:

STEP	**R-Type (General)**	**SW**	**Beq**	**Load Immediate**	**Jump**	**Push**	**Pop**	**Jump and Link**	**LW**
Instruction Fetch	IR = Mem[PC] PC = PC + 2								
instruction Decode	A = Reg[IR[11-8]] B = Reg[IR[7-4]]								
ALU	ALUOut = A op B if(sub) CMP=-1/0/1	NOT USED	if(CMP==B) then PC={0xF,RS [11-0]};	ALUOut = 0	ALUOut = {0xF,A[11-0]}	NOT USED	NOT USED	NOT USED	NOT USED
Memory access	Reg[IR[3-0]] = ALUOUT	Mem[A] = B	NOT USED	Reg[ALUOut] = SE(Reg[11-0])	PC=ALUOut	Mem[SP] = Reg[IR[11-8]] SP=SP-2	Reg[IR[11-8]] = Mem[SP+2] SP=SP+2	Reg[D]= PC+2 PC={0xF,RS[11-0]}	Mem[DataOut] = Mem[A]
Register write	NOT USED	NOT USED	NOT USED	NOT USED	NOT USED	NOT USED	NOT USED	NOT USED	Reg[IR(11:8)]=Mem[DataOut]

List of components for RTL:

- ALU
- 16 addressable registers
- PC register
- A register
- B register
- AluOut register
- memory
- Instruction register
- Memory direct register
- sign extender
- a bunch of muxes
- a control unit to control those muxes
    - PCSRC
    - PCWrite
    - MAddr

    - MDin
    - MRead
    - MWrite
    - RFWA
    - RFWD
    - RFRead
    - RFWrite
    - SPWrite
    - AWrite
    - BWrite
    - ALUInA
    - ALUInB
    - ALUOP
    - ALUOutWrite
    - Branch
    - SPRel
    - PshPop

List of Input,Output, and Control Signals:

STEP	**INPUTS**	**OUTPUTS**	**CONTROL SIGNALS**
FETCH	MemData - What to load into IR (16) PC - PC that will be incremented by 2 (16)	Read1 - Future A (4) Read2 - Future B (4) Read3 - Future C (4)	IRWrite - Should IR write (1) PCSource - Pick next PC (2) PCWrite - Should PC write (1) IOrD - Should an Instruction be read or data (1) PCWrite - Should PC Write (1)
DECODE	Read1 - Future A (4) Read2 - Future B (4) Read3 - Future C (4) Write - Should it write(1) WriteData - Data to write (16) ALUOut - Future C's value (16)	ReadData1 - A (16) ReadData2 - B (16) ReadData3 - ALUOut (16)	RegWrite - Should Reg Write
ALU	A - A Input (16) B- B Input (16) OpCode - OpCode.... C - C Input (16)	Zero - Detects if no difference (1) Overflow - Detects a carryout (1) ALUOut - Result of ALU operation (16)	ALUSrcA - Choose A or PC (1) ALUSrcB - Choose B Value (2) OpCode - Sent from Control system to input (4) PCSource - Choose source for PC (2)
Memory Access	WriteData - Data going into mem/reg (16) Write - Where to write (4) Address - Address to write to (16)	MemData - Memory to IR & MDR (16) MDR - MemData (16)	MemWrite - Should it write data (1) IorD - Instruction or Data (1) MemRead - Should it read (1) MemToReg - Choose MDR vs ALUOut(1) RegWrite - Should data write (1) SPSource-Should SP update PushPop-Is it a pop or a push
Reg. Write	Write - Where to write (4) WriteData - Data going into reg (16)		RegWrite - Should Reg Write (1) MemToReg - Choose MDR vs ALUOut (1) RegDst - Choose write address (1)

Test:

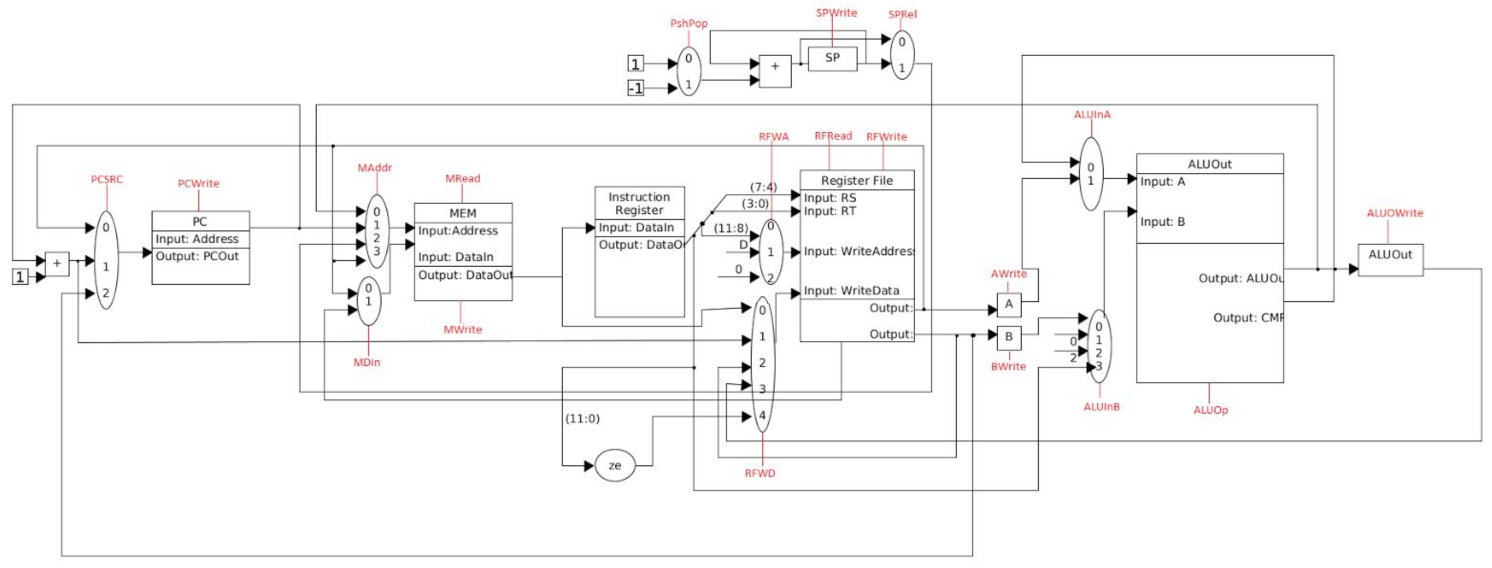
Test	**R**	**LW/SW**	**Beq**	**Load Immediate**	**Jump**	**Push**	**Pop**
Test Instruction Fetch	Input: PC=F000 Output: PC=F002,IR=data in Mem[F000]						
Instruction Decode	Input: IR=B334 Output: A=data in $p0, B=data in $p1, ALUOut=data in $p0						
ALU	Input: A=1 B=3 IR=2XXX Output: ALUOut=4	Input : A=5 Output: ALUOut=5	Input: A=2 B=2 IR=XXX0 Output: PC= data in $imm	Input: A=2 B=3 Output: ALUOut=0	Input: A=2 B=3 Output: ALUOut=0xF002	Assuming data in $sp=4 Output: ALUOut=2	Assuming data in $sp=4 Output: ALUOut= 6
Memory access	Input: ALUOut=3 IR=X2XX Output: data in $ret is 3	Input: B=2 ALUOut= F000 Output: MDR=whatever in Mem[F000] Mem[F000]=2	NOT USED	Input: IR=7222 Output: $imm=222	Input: ALUOut=F222 Output: PC=F222	Input: IR=X0XX $sp=0 Output: Mem[2]=the data in $imm	Input: $sp=2 IR=X0XX Output: $imm =whatever in Mem[0]
Register write	NOT USED	Input: IR=XX0X MDR=2 Output: $imm=2	NOT USED	NOT USED	NOT USED	NOT USED	NOT USED

The Datapath: https://drive.google.com/open?id=0B67zGSAD3YdwRE5OcW1VX1AzZW8

Our memory will be 65kb made the same as done in lab 7. Instruction, A, B, SP, CMP and aluout registers will all store 16-bit values. The datapath will all start at the PC register. Then, the address in the PC register goes to memory to pull the next instruction. After getting the instruction, the information hits the register file and fetches register values. The outputs of register file are stored in register A and B. A and B will both enter the ALU where it does some magic math and spits the answer to the ALUOut register. From there, information will go back to the register file to save. The push and pop instructions will use the SP register for arithmetic and memory writeback addresses.

Description of control signal:

PCsrc: Control if the PC should change
PCwrite:Control if the data is written
Maddr: Choosing the address in memory
Mdin: Choosing what is the data in
Mread: Control if the mem is able to be read
Mwrite: Control if the data should be written to memory
RFWD: Select what data to write in RF
RFWA: Select what address to write in RF
Awrite: Control if A should be written
Bwrite: Control if B should be written
ALUInA: Choosing what is input A for ALU
ALUinB: choosing what is input B for ALU
ALUOP: Control what is the operation for ALU
ALUWrite: Choosing if we write data into ALUOut
RFRead: Choosing if the Register File are able to read
RFWrite: Choosing if the Register File are able to write
SPWrite: Choosing if SP is able to write



How to Test:
Individual:(***Done***)

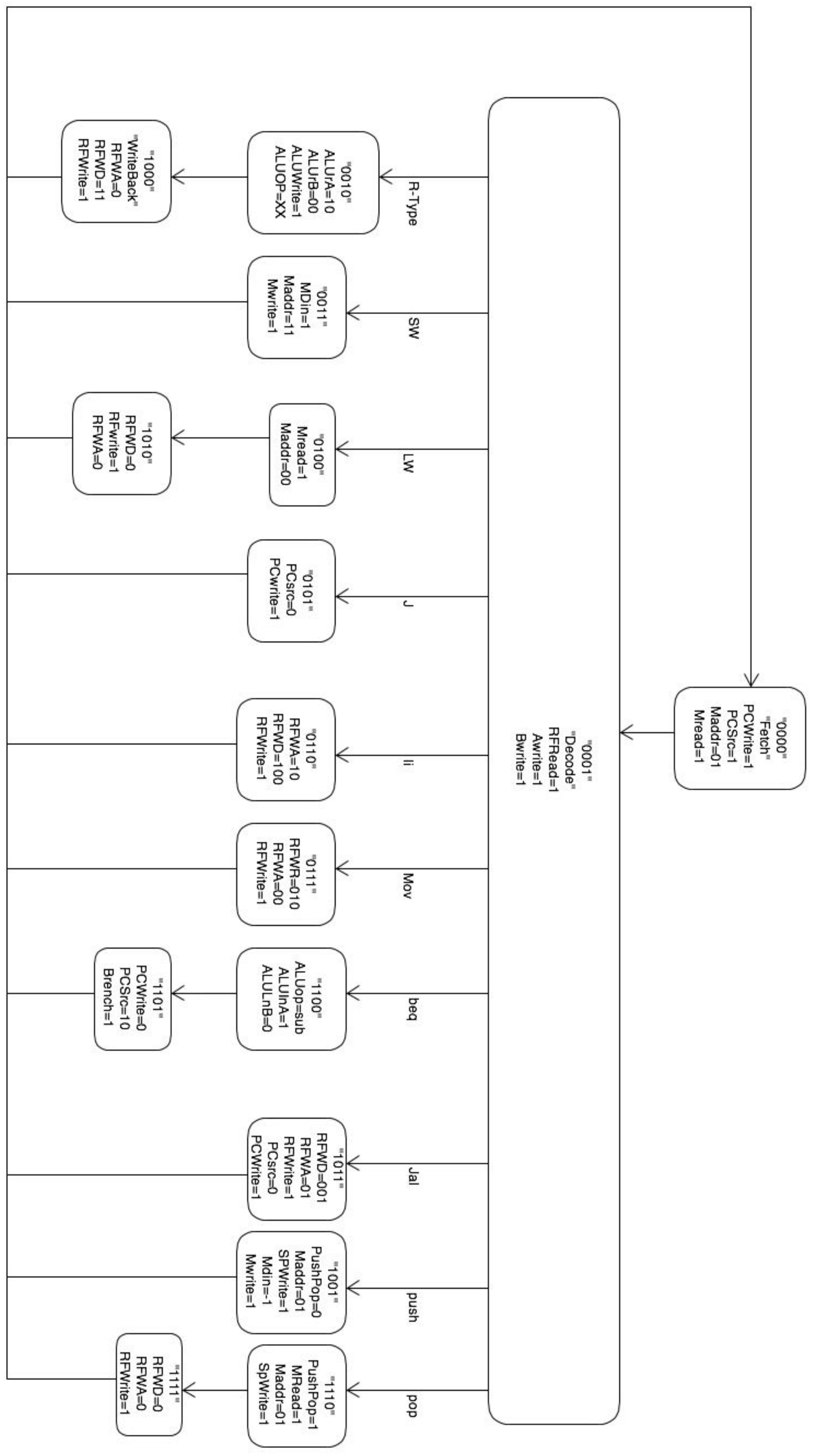
- We will test the memory by writing a bunch of stuff and trying to read it back
- We will test the IR by inputting an instruction and see if it gives right registers and opcode
- We will test RF by seeing if we are able to read from RF and write to RF
- We will test the ALU by inputting A, B and ALUop and see if it gives the correct result

Group of components:

- We can test the register file and memory together by using memory as an input for register addresses
- The ALU and register file can be tested together by adding parts of memory and storing them in the register file
- The PC and memory can be tested together by reading through a set of instructions

How to make the components:
PC register: copy-paste the register that is given by instructor
Mux: we can probs just tape a bunch of 1-bit muxes together
instruction register: copy-paste the register that was given to us
Register file: tape a bunch of given registers and select with a mux
SP register: take normal register and force 4 msb to be high all the time
ALU: Verilog OP
Aluout: another given register
Tests:
All the individual are done and they are in the XilinxProject files, open the project file to check it please.

FSM:


"0000" "Fetch" PCWrite=1 PCSrc=1 Maddr=01 Mread=1
"0001" "Decode" RFRead=1 Awrite=1 Bwrite=1
R-Type
"0010" ALUrA=10 ALUrB=00 ALUWrite=1 ALUOP=XX
"1000" "WriteBack" RFWA=0 RFWD=11 RFWrite=1
SW
"0011" MDin=1 Maddr=11 Mwrite=1
LW
"0100" Mread=1 Maddr=00
"1010" RFWD=0 RFwrite=1 RFWA=0
J
"0101" PCsrc=0 PCwrite=1
li
"0110" RFWA=10 RFWD=100 RFWrite=1
Mov
"0111" RFWR=010 RFWA=00 RFWrite=1
beq
"1100" ALUop=sub ALUInA=1 ALULnB=0
"1101" PCWrite=0 PCSrc=10 Brench=1
Jal
"1011" RFWD=001 RFWA=01 RFWrite=1 PCSrc=0 PCWrite=1
push
"1001" PushPop=0 Maddr=01 SPWrite=1 Mdin=-1 Mwrite=1
pop
"1110" PushPop=1 MRead=1 Maddr=01 SpWrite=1
"1111" RFWD=0 RFWA=0 RFWrite=1

”0000” is also the eset state

TRUTH TABLE FOR CONTROL SYSTEM:

Current State	**Opcode**	**Next State**	**NOTES**
0	X	1	**Initial State(Reset State)**
1	R-TYPE OPCODE	2	**Rtype**
2	X	8	
8	X	0	
1	X	3	**SW**
3	X	0	
1	0	4	**LW**
4	X	A	
A	X	0	
1	3	5	**j**
5	X	0	
1	7	6	**li**
6	X	0	
1	F	7	**mov**
7	X	0	
1	4	C	**beq**
C	X	D	
D	X	0	
1	C	B	**jal**
B	X	0	
1	D	9	**push**
9	X	0	
1	E	E	**pop**

E	X	F	
F	X	0	

TESTING THE CONTROL SYSTEM:

The way we will test the control system/state machine to verify that it works is by entering a current state into the test, along with an opcode and cmp values. After entering the information we will trigger a clock edge and check to see if the signals that need to be set are all correct, after which we will have a display that remarks “Pass” or “Fail”. We will make this test for all potential paths through the control system that we currently have.

Integration testing

1. PC, instruction memory, and control -jim
    a. fill instruction memory with some sweet instructions
    b. increment loop pc and read entire program
    c. check for proper control flags
2. ALU and Memory
    a. put a bunch of values into A and B
    b. use different opcodes to get some neeto data
    c. store in memory
    d. test if correct values
3. memory and register file
    a. put some dope values in the register file
    b. make some instructions in memory
    c. see what values come out of register file
4. ***Stack(Can not do it until the the whole datapath is made. So it is combined to Datapath test)***
    a. ***put a bunch of stuff on stack***
    b. ***pop it all off***
    c. ***???***
    d. ***profit***

System Test Plan

In order to test the entire datapath once all of the integration testing has been passed and validated, our group will test the entire datapath by attempting to run solve for a GCD using Euclid’s algorithm. The code that will be used to run this begins on page 4. If the greatest common denominator is found, then we will consider the datapath to work and is valid. From there we can begin to add extra features.