M.U.S.H. Architecture

Group 4-F

This Group Best Group

Final Report

Jim Gildersleeve

Guilherme Simas

Brian Suchy

Songyu Wang

Contents

Executive Summary:

The purpose of this report is to describe the creation of the M.U.S.H processor. The design tasked our group with creating a processor which could at minimum fid the first relatively prime number to an entered number. The strategy our group used to solve this problem would be to begin with a load store processor and create usability additions to the processor which would incorporate the other design types. The process began by initially brainstorming a set of operations we would need to fully execute Euclid's algorithm . Once the instructions were listed, we began to design a processor that could accomplish these tasks, along with extra instructions for that gucci extra credit. Once the system was created we began our testing process. We first began by testing our individual components. These

tests passed without failure. We then proceeded to the next phase which was integration testing. Two of our three integration tests ran completely perfectly. The test that failed was an integration test between the memory and control system, which was later corrected. Once the integration testing was completed, the final test of running Euclid's alogirthm and RelPrime began. This phase was the most involved pase of the entire project. Debugging the assembly and datapath for this phase require many manhours (including a single all-nighter); however, once everything was debugged it was smooth sailing. Our team then collected performance data and concluded that our processor was quite efficient. In conclusion, our team sucessfully designed a processor that could compute the first relatively prime number to a given number. It also has extra functionality added to make it a more robust system. Some potential changes that our group has been considering is as follows: create a pipeline processor, if not pipeline then a single cycle, create a ternary brach statement, and finally create a single jump to a label command (as in not a pseudo-instruction). Our teams reccomendations are as follows: utilize a HDL (hardware description language) such as VHDL or Verilog, use minimal flip-flop encoding over one hot encoding for resource conservation, separate output and next state logic, and finally have fun with the project.

Introduction:

The M.U.S.H architecture we made is a general purpose 16-bits processor that can execute programs stored in an external memory. After all the team members' hard working, the processor is fully functional. It can do basic operation for general purpose. Also, it can give the results of relprime for any input number.

Instruction Set Design:

Our instruction set was designed around the idea that we wanted keep all operations within the registers as much as possible. This lead to a design built around a single immediate-type instruction, load immediate. Since this was our only I-type command, we had to build every other command to manipulate only registers. The commands go as follows:

opcode	assembly command	Instruction name
0x0	lw	load
0x1	sw	store
0x2	add	add
0x3	j	jump
0x4	beq	branch equal
0x5	sll	shift left logical
0x6	srl	shift right logical
0x7	li	load immediate
0x8	and	and
0x9	or	or
0xB	sub	subtract
0xC	jal	jump and link
0xD	push	push
0xE	pop	pop
0xF	mov	move

Implementation:

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.

Xilinx model:

The following image is the final resultant data path. The descriptions for each 'section' are as follows:

Push/Pop: This section controls writing to and reading the Stack Pointer for Push and Pop commands. It has an adder/subtractor and a register that holds the values.

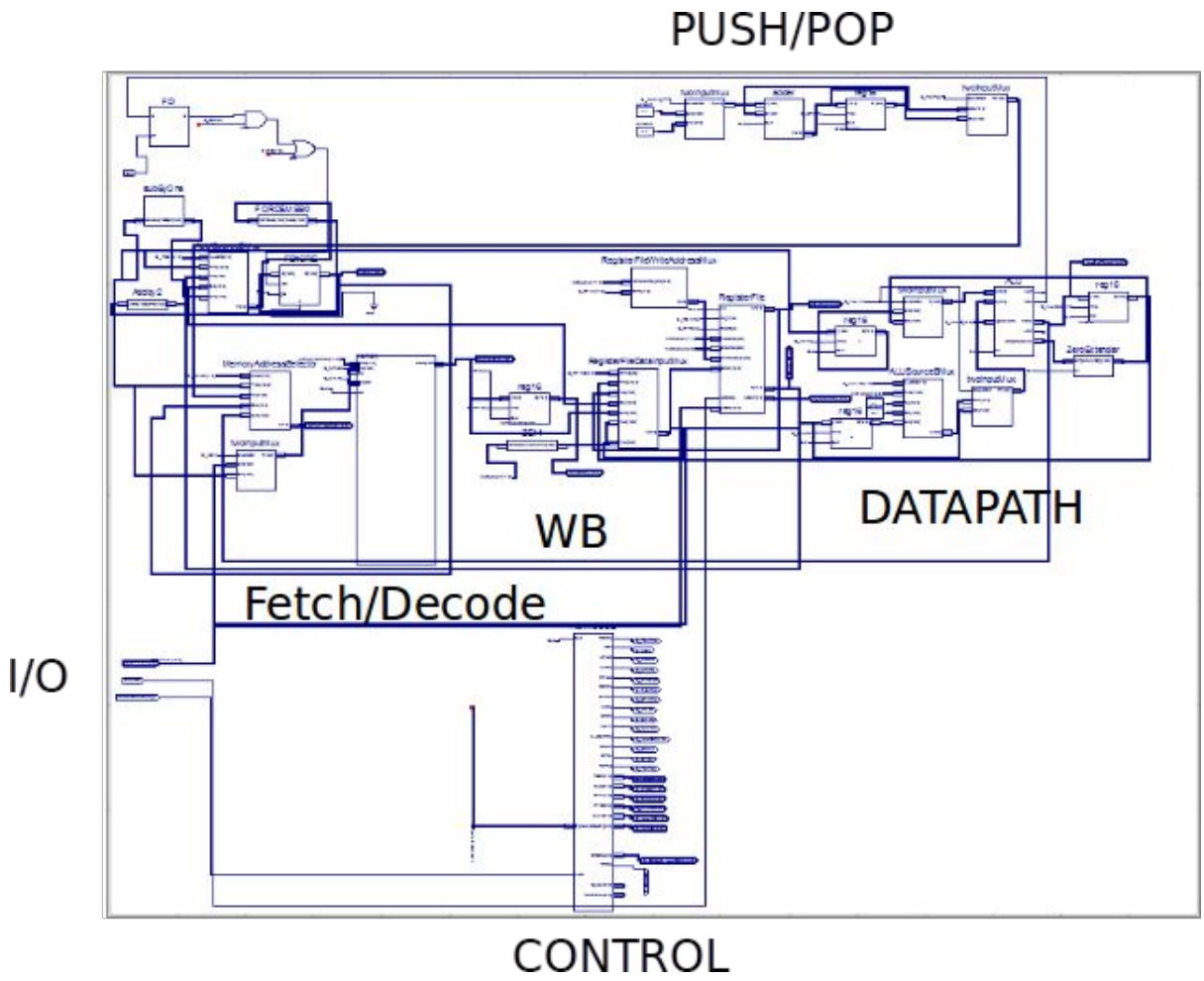
Datapath: This section contains the ALU and registers that are connected to it, it does a majority of our logic based operations; however, it is never used as a pass through for data.

Control: The brains of the operation. This controls all of the output signals for the system, with the exception of Zero, that controls what state we are in and what state we are going to.

I/O: For input and output we attached a data input and enable to write directly to a register in our reguster file. This bypasses the need for exception handling.

Fetch/Decode: This area contains the PC and memry for getting the instruction and storing it for use by the control system. Not much more….

WB: This is the bread and butter of our design. We are able to keep most of our cycles to 3 cycles because we bypass data transfer commands to go straight into the register which makes running euclids very easy due to the fact that the ALU is only needed for a lesser time than data transfer is needed.



Test and Performance Data:

1. The total number of bytes required to store both Euclid's algorithm and relPrime as well as any memory variables or constants.
   47 bytes(1byte= 16bits)
2. The total number instructions executed when relPrime is called with 0x13B0 (the result should be 0x000B using the algorithm specified in the project specifications).
   112372
3. The total number of cycles required to execute relPrime under the same conditions as Step 2.
   388157
4. The average cycles per instruction based on the data collected in Steps 2 and 3.
   3.4542 cycles per instruction
5. The cycle time for your design (from the Xilinx Synthesis report – look for the Timing summary).
   9.219ns
6. The total execution time for relPrime under the same conditions as Step 2.
   3578419ns
7. The gate count for your entire design (from the Xilinx Map report). This appears to have changed/is omitted in recent version. Extra credit for any group that finds a

reasonable way to estimate the equivalent gate count from the data in the Xilinx reports.

//

8. The device utilization summary (from the Xilinx Synthesis report).

Device Utilization Summary				[-]
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Total Number Slice Registers	405	9,312	4%	
Number used as Flip Flops	122			
Number used as Latches	283			
Number of 4 input LUTs	864	9,312	9%	
Number of occupied Slices	592	4,656	12%	
Number of Slices containing only related logic	592	592	100%	
Number of Slices containing unrelated logic	0	592	0%	
Total Number of 4 input LUTs	880	9,312	9%	
Number used as logic	864			
Number used as a route-thru	16			
Number of bonded IOBs	181	232	78%	
IOB Flip Flops	4			
IOB Latches	20			
Number of RAMB16s	20	20	100%	
Number of BUFGMUXs	2	24	8%	
Average Fanout of Non-Clock Nets	3.78			

Conclusion:

This Group Best Group learned a lot from this project. After a month of design and implimentation, we have all learned computer design principles, hardware debugging skills and how to work better as a team. We have all now ascended to a higher plane of omnicience with out newly aquired knowledge.

We now have a sweet processor that can do basic functions that is tested through running euclids algorithm to detect the first relativelty prime number to any input. Our group is content with the design that we have finished although there is always room for improvement.

Appendix A:

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.

R-Type Instruction Layout:

4	4	4	4
OP	RS	RT	RD

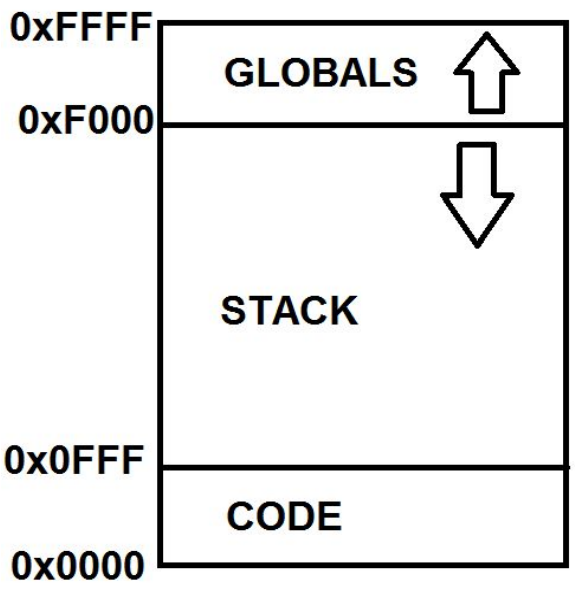
This instruction will be the bread and butter of our processor. All but one instruction will use solely
R-Type instructions. The R-Type instruction will have a 4-bit opcode and have 3 4-bit register addresses. The registers themselves will hold the jump locations and other necessary information as opposed 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	register used for input/output
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 points to first empty element on stack
compare value	cmp	implied register set by sub and ready by beq that holds a -1,0,1 to represent inequalities

I/O:
Input and Output will be dealed with using the I/O register. Inputs will be forwarded to the register, so reading an input is essentially reading the register. Outputs will be read from the I/O register as well, so outputting data would be storing that data in the register.

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:

opcode	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 = {0x0,RS[11-0]}
0x4	beq	branch equal	R	if(RD==cmp) PC = {0x0,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
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
0x0001	li 0	7000	//t0=0
0x0002	mov $t0 $imm	F500	
0x0003	sub $t1 $t0 $p0	B653	//a==0?
0x0004	li 0x012	7012	//t2= address of if1
0x0005	mov $t2 $imm	F700	
0x0006	beq $t0 $t2	4570	//if (a==0)True, go to if1
0x0007	li 0x016	7016	//t1=address of while
0x0008	mov $t2 $imm	F700	
0x0009	j $t2	3700	//if (a==0)False, go to while
	//return b		
	if1:		
0x000a	mov $ret $p1	F140	//prepare to return [return address(ret) = $p1(b)]
0x000b	j $ra	3D00	//return
	while:		//while
0x0016	sub $t1 $t0 $p1	B654	//b==0?
0x0018	li 0x040	7040	//t2 = address of returnA
0x001A	mov $t2 $imm	F700	
0x001C	beq $t0 $t2	4570	//if (b==0)True go to returnA
	//if (a>b)		
0x001E	sub $t1 $p0 $p1	B634	//check a?b
0x0020	li 0x030	7030	//t2=address of if
0x0022	mov $t2 $imm	F700	

0x0024	li 1	7100	//t3=1
0x0026	mov $t3 $imm	F800	
0x0028	beq $t3 t2	4870	//if (a>b) go to if
0x002A	li 0x038	7038	//t2=address of else
0x002C	mov $t2 $imm	F700	
0x002E	j $t2	3700	//jump to else
	//a=a-b		
	if:		
0x0030	sub $p0 $p0 $p1	B334	//a=a-b
0x0032	li 0x016	7016	//t2 =address of while
0x0034	mov $t2 $imm	F700	
0x0036	j $t2	3700	//goto while
	//b=b-a		
	else:		
0x0038	sub $p1 $p1 $p0	B443	//b=b-a
0x003A	li 0x016	7016	//t2 =address of while
0x003C	mov $t2 $imm	F700	
0x003E	j $t2	3700	//goto while
	//return a		
	returnA:		
0x0040	j $ra	3D00	//return

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

address	assembly	machine code	C code
0x0000	li 0x123	7123	Load upper 12bits
0x0002	mov $t0 $imm	F500	Move into register
0x0004	li 8	7800	load number of

			shifts
0x0006	sll $t0 $imm	5500	shift t0
0x0008	li 0x4	7400	load lower 4 bits
0x000A	or $t0 $t0 $imm	9550	or them together

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

0x0000	li 0	7000	
0x0002	mov $t2 $imm	f700	//i=0
	for:		
0x0004	sub $t5 $t2 $t1	ba76	
0x0006	li [block]	7012	
0x0008	mov $t4 $imm	f90q	
0x000A	li -1	7fff	
0x000C	beq $imm $t4	4090	
0x000E	li [done]	701c	
0x0010	j $imm	3000	
	block:		
0x0012	li 1	7001	
0x0014	sll $t0 $t0 $imm	5550	
0x0016	add $t1 $t1 $imm	2660	
0x0018	li [for]	7004	
0x001A	j $imm	3000	
	done:		
0x001C	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

0x0000	/*load 0x1234 in $t0*/	See example above	See example above
0x0002	li 4	7004	load index
0x0004	mov $t1 $imm	F600	$t1 = index
0x0006	li 1	7001	prepare to shift
0x0008	sll $t1 $t1 $imm	5660	shift index once
0x000A	add $t1 $t1 $t0	2665	add base address to index
0x000C	lw $t1 $t1	0660	load the element

Machine Code: 0xF000 7001 0xF001 F3000xF002 7002 0xF003 F400 0xF004 0440 0xF005 0330 0xF006 7000 0xF007 F500 0xF008 B653 0xF009 700f 0xF00A F700 0xF00B 4570 0xF00C 7011 0xF00D F700 0xF00E 3700 0xF00F F140 0xF010 3D00 0xF011 B654 0xF012 7025 0xF013 F700 0xF014 4570 0xF015 B634 0xF016 701e 0xF017 F700 0xF018 7001 0xF019 F800 0xF01A 4870 0xF01B 7022 0xF01C F700 0xF01D 3700 0xF01E B334 0xF01F 7011 0xF020 F700 0xF021 3700 0xF022 B443 0xF023 F700 0xF024 3700 0xF025 3D00
:)

RTL:

STEP	**R-Type (General)**	**SW**	**Beq**	**Load Immediate**	**Jump**	**Push**	**Pop**	**Jump and Link**	**LW**
Instruction Fetch	IR = Mem[PC]								
instruction Decode	A = Reg[IR[11-8]] B = Reg[IR[7-4]] PC = PC + 1								
ALU	ALUOut = A op B if(sub) CMP=-1/ 0/1	NOT USED	if(CMP== B) then PC={0x0, RS[11-0]} ;	ALUOut = 0	ALUOut = {0x0,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={0x0,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
- CrtState
- IRWrite

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) PCWrite - Should PC write (1) IOrD - Should an Instruction be read or data (1) MRead - Should it read (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)	AWrite - Should it write to A register (1) BWrite - Should it write to B register (1) PCWrite - Should PC write (1) PCSrc - Pick next PC (1) RFRead - Should the register file read (1)
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 CMP (2) ALUSrcB - Choose B Value (1) OpCode - Sent from Control system to input (4) ALUOutWrite - Should ALU output (1) Branch - is a branch happening (2) PCSrc - Pick next PC (1)
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 MDin - What data to load into memory (2) MAddr - What address goes into memory (4)
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=0x0002	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

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

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.

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

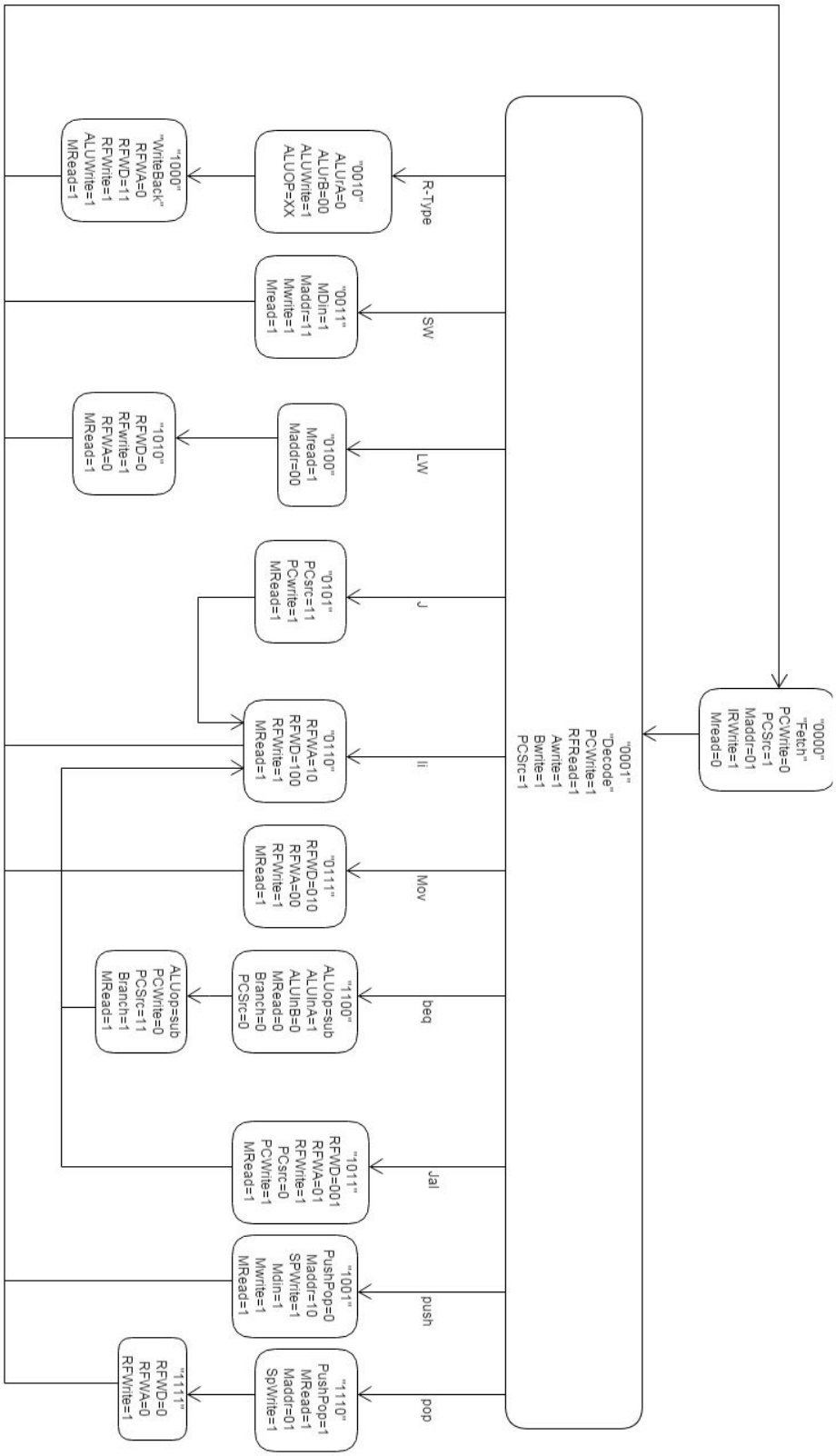
Group of components:

- 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” is also the reset 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.

Final boss assembly code:

```
li start
j $imm

retAddr:
li retAddr
j $imm

start:
li 8
mov $p0 $imm

li 6
mov $p1 $imm

euclid:
li 0
mov $t0 $imm

li done
mov $t4 $imm

mov $ret $p1 //ret = b
sub $imm $p0 $t0
beq $t4 $t0 //if a == 0 ret b

loop:
li 0
sub $imm $p1 $imm
beq $t4 $t0 //while b != 0

li 3 //-1
mov $t0 $imm

li else
mov $t5 $imm

sub $imm $p0 $p1
beq $t5 $t0 //if(a < b)
sub $p0 $p0 $p1

li endLoop
j $imm

else:
```

```
sub $p1 $p1 $p0

endLoop:
mov $ret $p0
li loop
j $imm

done:
mov $ret $p0
li retAddr
j $imm //return;
```

Appendix B:

Design Process Journal

The following is the rationale behind our choices for the processor:

October 6, 2015

1. 16 registers were selected because we do not wish to deal with a constrained amount of registers and we do not wish to be extremely cautious when using our registers. Also it reduces to 4 bits for addressing therefore only utilizing a fourth of the instruction per register. (Also, hex code is OP)
2. 5 unaddressable registers: PC, SP, ~~RA~~ (10/12), INT(interrupt) , and CV(compare value). these registers should not be touched except for the instructions explicitly used for this.
3. 16 instructions were selected to have an optimal amount of instructions while still only using a fourth of the instruction space. 8 instructions would be much too little while 32 was way too many types of instructions. (Also, hex code is OP)
4. Two different instruction types were selected for simplicity. All instructions use the R-type instruction except when we need to load an immediate value. By maximizing the use of R-type, we have a nice standard to implement in hardware.
5. We have decided to make a M.U.S.H. architecture. Our architecture utilizes aspects of Load-Store and stack architecture. This allows us to use registers for speed, but also store information easily on the stack.

October 9, 2015

6. Jim decided to start all of the instructions at the address 0xF000 so we can imply an F for any instruction address
7. Jim decided to put globals from 0x0000 to 0x0FFF so that whenever an address in memory would need to be fetched, that address could be expressed in 12 bits with implied leading zeros. This allows us to fetch without shifting and oring the last 4 bits of the address

October 12, 2015

8. We decided to sacrifice temporary at index 13 so that we could index $ra. When asked if he would be diggy down with this, Brian replied, “I would be diggy down”. This will save a couple instructions.
9. Because the change, jal and ret are both unneeded instructions. They will now be depreciated until possible replacement in the future (maybe replace with a load upper immediate, NSA WARNING MODE, or a find GCD)
10. We made an RTL that was based upon the multicycle RTL given out in class; however, it goes more into detail for specific instructions and shows how we will construct the datapath.
11. Due to the changes above, the Euclids algorithm is changed, too.

October 13,2015

1. Nah, we need jal. There's no way to access PC otherwise, and we need to get the value of PC into RA for calls.

October 14,2015

1. Changed jump instruction. Instead of getting the value from RD we now get it from RS, because then we don't need to add the functionality of getting a third value from the register file.
2. Group met at the classroom during class hour to work on Milestone 3. Started with DataPath.
3. We decided we're not pipelining it.
4. We changed Store Word to store the memory from RT to RS. Same thinking process of the jump instruction. Also we use the same wiring from Load Word.

October 20,2015

1. Continued working on the DataPath, on push and pop instructions
2. Decided that there should be wires out of both ALUOut and SP to wd because of the ordering of "Access Memory / Update SP" for push and pop. Other option would be to have only ALUOut, and in that case when we're pushing a word into the the stack ALUOut first needs to be SP+0 and then SP-2. We didn' t want to have to use the ALUOut 2 times so we went with the first option. Although now it looks single-cyclish. Should ask Sid. Will ask Sid.
3. Got rid of ALUOut = Reg[IR[3-0]] on the second step of the rtl. We couldn't figure out why it was there in the first place
4. Since everytime PC gets updated, the first byte is 0xF, the PC register will only update the 12 least significant bits, and 15-12 will always stay high. That way we don't need an extra component to overwrite these bits.
5. Updated RTL and instructions descriptions, such as sub (didn't include CMP mechanic).
6. We look through each component individually and figure out what are input and output. Then design how to test it.
7. We uses $SP in our architecture and it becomes an register in the datapath. It is a separate thing which is different than MIPS.

October 23,2015

1. After discussing it with Sid, decided that SP has a separate Adder (like PC). Updated the DataPath, Control System and RTL to match the changes. Need to update the Diagram that includes the DataPath and the Control System to include SP Source and PushPop
2. Brian made a majority of the hardware components in Xilinx utilizing Verilog, he has yet to implement the tests to verify the hardware works….. but progress is being made.
3. Tests are made.

October 25,2015

1. added a mux to select -2 and 2 to the stack pointer
2. The group met and created control system diagram (later to be coded in verilog by Brian).

October 25,2015

1. made a truth table for the state machine
2. updated list of control flags to match datapath and control design

3. decided that combinational logic on branch will be part of datapath-- not control system
4. changed lw and sw in RTL
5. Changed implementation of the stack pointer this can be seen in new datapath diagram\
6. Jim still needs to use mad mspaint skillz to hack the NSA (finish datapath image)
7. Changed the FSM diagram. Don't need CMP to go into the Control System, it can just go to a circuit which outputs to the "enable" of the PC Register. For that to happen we needed to add another control flag (branch). So the logic for the enable input of the register is EN=PCWrite + (CMPisZero.branch) //CMPisZero is a NOR of all CMP bits.

October 26,2015

1. Yay! jim used his mad 1337 mspaint h4x to put control signals back on that datapath
2. No combinational units means no truth table.#Rekt
3. We decided reset state is "0000"

October 30,2015

1. Most tests are done and most components are fixed
2. Register file and write decode is unfinished because Songyu do not fully understand what they are for. So someone else will fix them, I hope.
3. When Songyu tried to commit codes to svn, it said "no space left". So Songyu guesses we need to recreate the whole project by following lab 7 instruction. It's lots of work!

November 2,2015

1. OH GAWD WE ALMOST HAD AN SVN TRAGEDY
2. *shower thought* why did they not teach us how to version control formally
3. Why not use GIT?"!?!?!?!?!?!?
4. made some sweet plans for testing
5. Scratch #1, Shit went crazy. ALU (praise be unto him) is angry with us. Our sinful ways have angered the ALU. Brian, was an asshat and never committed so today we tried making stuff work. He cried a single manly tear for causing his teammates troubles and pain. It should be noted that the tear he cried was bench pressing 250 while it slid down his cheek.
6. Jim will purge the sodom below that is the svn and e-mail the solutions
7. Brian still doesn't update the journal as much as he should.
8. Brian is finishing off the components tests and will integration testing soon.
9. sodemn has been purged

November 3,2015

1. We discovered that the memory in xilinx addresses memory as 16-bit blocks. So we changed our datapath to add/sub "1" to PC and SP (instead of "2").
2. We cannot test Stack in integration test plan because it requires the whole data path to test . So we combine it to the datapath test.
3. All the example test are removed because they are in the Xilinx Project now. Please check them in that folder.
4. No RTL,control unit design changed.
5. datapath is changed slightly

November 8,2015

1. never explicitly said what sp points to now says first empty element in stack
2. SpReal was deleted because we forgot it. It had the same value of pshPop so no biggie
3. Instructions in memory now start at 0x0000 because it's easier to edit the memory and hardcode the instructions. PC now forces 0x0 at it's start.(If 0xFFFF is assigned to PC, it will contain 0x0FFF). We edited the memory layout accordingly.

November 10,2015

1. The ghetto assembler has been fixed to support labels and not mess up (hopefully)
2. The ghetto assembler can now output code as a .coe with the -coe flag

November 11,2015

1. Ghetto assembler patches

November 11,2015

1. Wow
2. What an afternoon / night / morning. We were past documentation. We had broken those shackles. Rebelled against ALU (praise be unto him) and commenced an era of chaos and war. After our people were almost wiped out from the plagues sent from ALU (praise be unto him) and all its wrath (we lost Brian at 7:10am RIP), we decided to willingly go back to our documenting ways. So here's what's up:
3. Instructions that changed the PC such as jump and beq were not working properly since the instruction it fetched seemed to be the one after the one we were supposed to go to. So we added a SubByOne component which decrements the address it's supposed to go to by one so that it ends up in the one we originally planned. This solved that problem.
4. Beq and Jump were still not working as intended. We found out that the next instruction after the beq/jump was not being loaded in PC in time for the fetch stage (the waveform had a small delay, we have no idea why. PC register just doesn't seem to update at the clock edge, just slighly after.) Brian (RIP) changed the control system over and over until they worked despite the delay.
5. When we tried to run Euclid's algorithm we got a simulation error that said "Iteration limit 10000 is reached". We still don't have a clue what that is. After going to Sid we suspected it was due to an output from the ALU going back as an input to the ALU.
6. Following Sid's suggestions we clocked the ALU so as to prevent the "Iteration limit 10000 is reached"

November 12, 2015

1. LETS TEST RELPRIME 8 FIRST

November 15, 2015

2. Brian has failed at updating the journal enough.