Final Report

CSSE 232-03

Team RED

Abi Naseem
Michelle Zhou
Darren Zhu
Joshua Giambattista

Architecture name:

Accumulator

Table of Contents

Introduction

Our design is based on accumulator. The highlight of our design is it is easy to program. We a table showing all the opcode, funct for each instruction. We also have detailed 16-bit instructions for the user to read through. Take the instruction “add $wr, $tp, 0” as an example, it will add up the value in the working register and the temporary register. The last bit 0 means the result will go into the first register in the instruction which is the working register. If the last bit is 1, the result will be stored into the temporary register. Take the instruction “addi” and “subi” as another example, both of them will either add or subtract an immediate from the current value stored in the working register. Both of them are sign extended, meaning the user can do “addi -0x6” or “subi -0x7”, these instructions are user friendly.

Registers and Instruction Set Design

Registers

There are 8 registers used in our design: One working register ($wr), one memory address register ($ma), one argument address register ($ar), one number of arguments register ($na), one return value register ($rv), one stack pointer register ($sp), one return address register ($ra) and one temporary register ($tp). More details can be seen on page 7.

Instruction Format

Our instruction design includes three formatting types: R-Type, D-Type and IJ-Type. All of them fit the 16-bit instruction size constraint.

For R-Type Instruction, it uses 4 bits for the opcode, 4 bits for funct, 3 bits for registerSelect to select from which of our 8 registers to use, 3 bits for registerSelect2 and 1 bit for the result location (1 means save the result to registerSelect register, and 0 is the registerSelect2 register). The remaining 1 bit is the dead bit.

For D-Type Instruction, it uses 4 bits for the opcode, 4 bits for funct, 3 bits in the registerSelect to select from which of our 8 registers to use and 4 bits for Delta, which is used for shifting in shr and shl and a small immediate to either add or subtract for push, pop, incr, and decr. The last bit is a dead bit.

For IJ-Type, it uses 4 bits for the opcode and 12 bits for an immediate value. This works in all cases except for goto, which will use a PC-Relative type addressing mode where it shifts the immediate left by 1 and then adds the first three bits of PC onto the MSB of the immediate. Also, if an immediate is too big for the instruction, the user must use a lui and then an ori. But if the user is trying to load a memory address that is too large, then they use luia and orim.

Both the instruction format and the table of instruction set can be seen in more details on page 8.

Implementation

All 32 instructions are converted in to RTL (Register Transfer Language), which can be seen more detailed on page 18. All the instructions are completed in a multicycle datapath. All the components that are used are listed below. They are all being successfully tested for different inputs through testbenches.

Our entire processor has 2 inputs:

1. A reset button
2. A number that is put into the working register on reset

The output is the result of calculation retrieved form the working register.

List of Hardware implementations

- ► Adder
- ► ALU
- ► Left Shifter
- ► Memory
- ► 2-to-1 Mux
- ► 3-to-1 Mux
- ► 4-to-1 Mux
- ► 8-to-1 Mux
- ► 10-to-1 Mux
- ► Register File
- ► Sign Extender
- ► Zero Extender
- ► PC Register
- ► Instruction Register
- ► Controller
- ► Mini Controller

The list above shows all the hardware implementations we used in our design. More details can be seen on page 18-20.

Integration

Our datapath is divided into four integration phases:

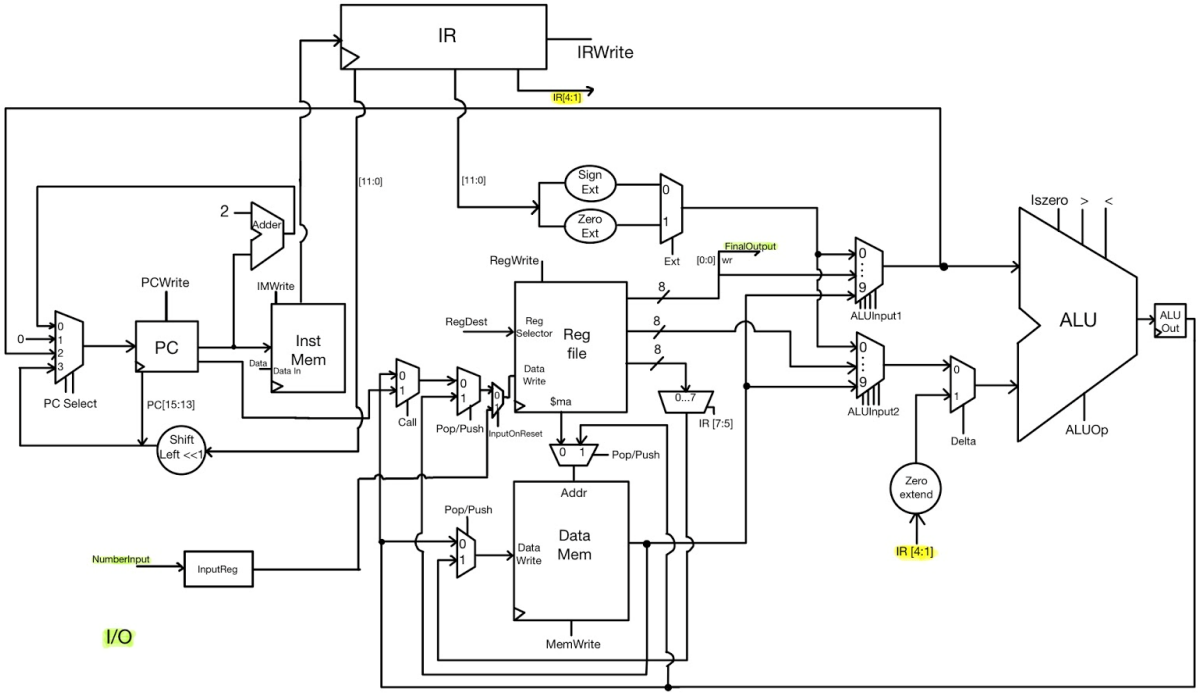
1. Adders and PC
   In this integration phase, we combine 2 adders, a PC register, Shift Left, a PC Select MUX (4 inputs to one output) and an Instruction memory.
2. Register File and Data Memory
   In this integration phase, we connect the register file and data memory with a call MUX, three pop/push MUXs, location select MUX and the data write MUX.
3. Registers

In this integration phase, we add the sign and zero extenders and a wall of 10 registers that will store each register, the immediate selected by the extenders and the data memory (The Wailing Wall).

4. ALU
   In the last integration phase, we add ALU, which is attached to its two inputs using the ALUInput1 and ALUInput2 MUXs. Additionally, the second input to the ALU is also selected between the Zero Extended delta and the registers using a Delta MUX.

We complete all these four phases and transfer them into Schematic. We then test them separately using test benches and form the final version datapath, which is shown below.

Datapath



Testing Methodology

First, we write the test benches for each hardware implementation and validate the output by given several inputs. Then we put components together to form 4 integration phases. We created the test benches for all of them and test if a given input can yield an appropriate output. Finally, we combine integration phases together to form the overall datapath. To test all the parts in our datapath, we'll put at least one instruction in the memory. We're going to test to see if we can successfully get the instruction information, run through the datapath, get the output, and then go to the next instruction accurately until all the instructions are done. In addition, we need to make sure that when we

run branches and jumps, it can go to the correct location and follow the previous steps to complete all the instructions.

Unique Features

The unique features of our processor include 3 things.

One, the use of the memory address register. The register is a pointer to somewhere in data memory so that if the user ever selects the register to do arithmetic, they are actually choosing the memory itself instead of the register. This freed up a lot of space in our instructions so that users could have more options.

The other unique feature is how we can skip instructions. We have seven instructions that can test for equality, inequality, etc. and will skip the next instruction if the condition is met. This makes if/else statements extremely easy in assembly and it is very intuitive to use right away.

Our design can do recursion, we have a sample recursion code in the design document.

Performance Test Results

1. Euclid's algorithm and relPrime	146 bytes
2. Memory variables	8 bytes
3. Total number instructions executed when relPrime is called with 0x13B0	153227
4. Total number of cycles required to execute relPrime under the same conditions as Step 2	612908
5. Average cycles per instruction	4
6. Cycle time	34 ns
7. Total execution time for relPrime under the same conditions as Step 2	19ms

Conclusion

Our design is based on the idea of using an accumulator to run some of the instructions we came up with that are useful. To achieve it, we designed a lot of hardware implementation and control signals, and then combined them to form the overall datapath. Each component is tested through test benches we made before going on to the next step. The advantage of our design is that the instructions are easy to understand even for those who knows nothing about it. However, one flaw of our design is that it has a relatively long running time, which lowers the efficiency.

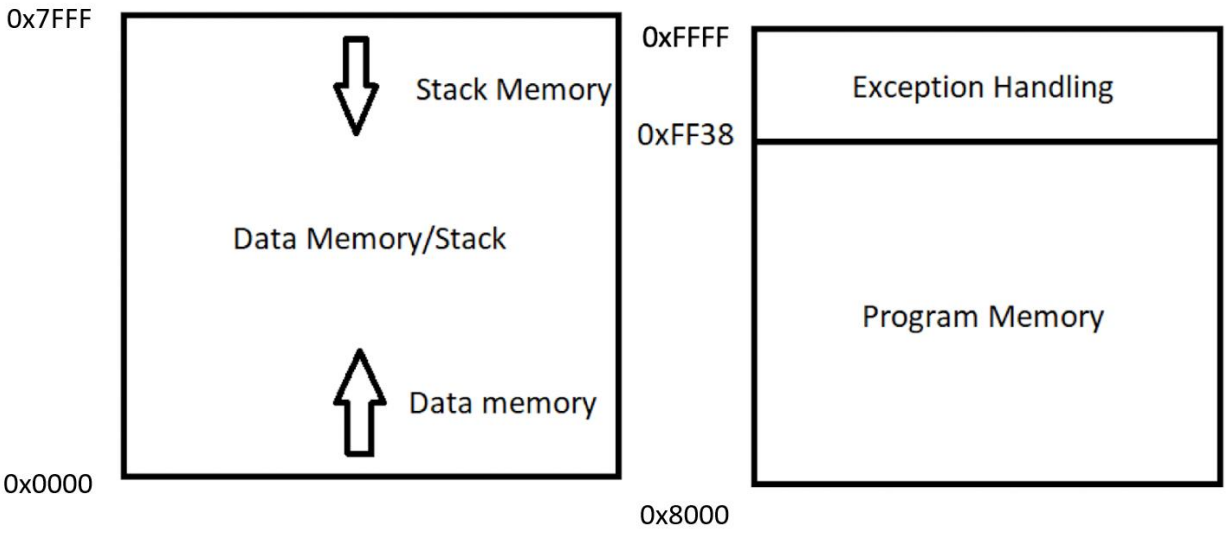
Appendix A

Design Document

Description of Registers

SYMBOL	NAME	NUMBER	DESCRIPTION
$wr	Working Register	0	It is the workspace where the operations happen.
$ma	Memory Address Register	1	It stores the memory address of the value which is used in the operations. For every instruction except for addm, the hardware will grab the value shown by the memory address register to use for the calculations. Addm directly increments the address stored in $ma
$ar	Argument Address Register	2	It stores the address of the first argument.
$na	Number of Arguments Register	3	It stores the number of arguments in total. Hence, the address of the second argument should be right after the address of the first one. With $ar and $na, the total number of arguments will be more flexible.
$rv	Return Value Register	4	It stores the value to be returned by the function. The return value should not exceed 16 bits.
$sp	Stack Pointer Register	5	It points to the space on the stack where data should be stored.
$ra	Return Address Register	6	It stores the return address of the program.
$tp	Temporary Register	7	It stores the temporary for the assembler.

Memory Map



Description of Instruction Format Types

We have three instruction format types: R-Type, D-Type and IJ-Type.

R-Type

OPCODE	FUNCT	REGISTER SELECT	DEAD BIT	REGISTER SELECT 2	RESULT LOCATION
15 12	11 8	7 5	4	3 1	0

For R-Type Instructions, we use 4 bits for the opcode, and 4 bit for funct. There are three bits in the registerSelect to select from which of our 8 registers to use. We also have 3 bit for registerSelect2. We leave one bit to represent the result location (1 is the register select register, and 0 is the register select 2 register) for some instructions.

D-Type

OPCODE	FUNCT	REGISTER SELECT	DELTA	DEAD BIT
15 12	11 8	7 5	4 1	0

D-Type stands for Delta-Type. For D-Type Instructions, we use 4 bits for the opcode, and 4 bits for funct. There are three bits in the registerSelect to select from which of our 8 registers to use. We also have 4 bits for Delta. This is used for shifting in shr and shl and a small immediate to either add or subtract for push, pop, incr, and decr. The last bit is a dead bit.

I/J-Type

OPCODE	IMMEDIATE / ADDRESS
15 12	11 0

For IJ-Type, we leave 4 bit for the opcode and the last 12 bits are used for an immediate value. This works in all cases except for goto will use a PC-Relative type addressing mode where it shifts the immediate left by 1 and then adds the first three bits of PC onto the MSB of the immediate. Also if an immediate is too big for the instruction, the user must use a lui and then an ori. But if the user is trying to load a memory address that is too large, then they use luia and orim.

Description of Instructions

R-Type:
add [R] (RegSelect 0:7), (RegSelect2 0:7),(LocationSelect 1,0)
If LocationSelect == 0, R(RegSelect) = R(RegSelect) + R(RegSelect2)
Else R(RegSelect2) = R(RegSelect) + R(RegSelect2)
{Add the register select register and the register select 2 register. If the user puts a 0 for the location select, the result gets stored into the register select register, if the user puts 1, the result gets put into the selected RegSelect2 register}

and [R] (RegSelect 0:7) , (RegSelect2 0:7),(LocationSelect 1,0)
If LocationSelect == 0,R(RegSelect) = R(RegSelect) & R(RegSelect2)
Else R(RegSelect2) = R(RegSelect) & R(RegSelect2)
{And's the register select register with the value at the register select 2 register. If the user puts a 1, it stores it into the RegSelect2 register, otherwise it gets stored back into the register select register}

jr [R] (RegSelect 0:7) PC = R(RegSelect)
{jumps to the address of the register select register}

move [R] (RegSelect 0:7), (RegSelect 2 0:7) R(RegSelect2) = R(RegSelect)
{Takes the value from the first (RegSelect) register and puts it into the second (RegSelect2) register. If the the user chooses to store in the memory address register, then the value will get stored into memory.}

movem [R] (RegSelect 0:7), $ma = R(RegSelect)
{Takes the value from the first (RegSelect) register and puts it into the $ma.}

or [R] (RegSelect 0:7) , (RegSelect 2 0:7),(LocationSelect 1,0)
If LocationSelect == 0, R(RegSelect) = R(RegSelect) | R(RegSelect2)
Else R(RegSelect2) = R(RegSelect) | R(RegSelect2)
{Or's the register select register with the RegSelect2 register. If the user puts a 0 for the location select, the result gets stored into the register select register, if the user puts 1, the result gets put into the selected RegSelect2 register}

seq [R] (RegSelect 0:7) if($wr == Reg(inst[7:5])) PC = PC + 4

{skips the next instruction if the value of the working register is equal to the value of the register select register}

sgt [R] (RegSelect 0:7) if($wr > Reg(inst[7:5])) PC = PC + 4
{skips the next instruction if the value of the working register is greater than the value of the register select register}

slt [R] (RegSelect 0:7) if($wr < Reg(inst[7:5])) PC = PC + 4
{skips the next instruction if the value of the working register is less than the value of the register select register}

sne [R] (RegSelect 0:7) if($wr != Reg(inst[7:5])) PC = PC + 4
{skips the next instruction if the value of the working register is not equal to the value of the register select register}

sub [R] (RegSelect 0:7) , (RegSelect 2 0:7), (LocationSelect 1,0)
If LocationSelect == 0, R(RegSelect) = R(RegSelect) - R(RegSelect2)
Else R(RegSelect2) = R(RegSelect) - R(RegSelect2)
{Does RegisterSelect - RegSelect2 and if the register location is 0 the result is stored in register select, if it is 1 then it is stored in the register selected by RegSelect2 }

D-Type:
clrr [D] (RegSelect 0:7) R(RegSelect) = 0x0000 {clears the selected register}

decr [D] (RegSelect 0:7), (Delta 0:15) R(RegSelect) = R(RegSelect) - ZeroExt(Delta)
{decrements the value of the selected register by the given immediate from 0 to 15}

incr [D] (RegSelect 0:7), (Delta 0:15) R(RegSelect) = R(RegSelect) + ZeroExt(Delta)
{increments the value of the selected register by the given immediate from 0 to 15}

pop [D] (RegSelect 0:7) (Delta 0:15) R(RegSelect) = Stack(ZeroExt(Delta))
{pops the value specified by the 0:15 number off the stack and into the register specified by the user}

push [D] (RegSelect 0:7) (Delta 0:15) Stack(ZeroExt(Delta)) = R(RegSelect)
{pushes whichever register the user picks from the register file into the stack by the Delta at the end of the instruction}

shl [D] (RegSelect 0:7) (Delta 0:15) Reg(inst[7:5]) = Reg(inst[7:5]) << Delta
{shifts the value of the selected register left by the Delta specified in the instruction and puts it back into the working register}

shr [D] (RegSelect 0:7) (Delta 0:15) Reg(inst[7:5]) = Reg(inst[7:5]) >> Delta

{shifts the value of the selected register right by the Delta specified in the instruction and puts it back into the working register}

I/J-Type:

addi [IJ] (Imm) $wr = $wr + SignExt(Imm)
{adds an immediate to the value of the working register and stores it back in the working register}

addm [IJ] (Imm) $ma = $ma + SignExt(Imm)
{adds an immediate to the memory address register, used to increment memory spaces for arrays and such}

call [IJ] (label) PC = {PC [15:13], label, 1'b0}, $ra = returnAddress
{jumps to the label specified and stores the return address into the return address register}

goto [IJ] (label) PC = {PC+2[15:13], label, 1'b0}
{jumps to the address of the specified label, requires the number in the immediate part of the instruction to be shifted left once and takes the first three bits of PC to do the jump}

la [IJ] (variable/memoryAddress) $ma = (ZeroExt(Imm))
{takes a memory address and stores it into the memory address register}

lui [IJ] (Imm) $wr = (UpperHalf((Imm)))
{takes the upper half of the immediate and stores it into the working register}

luia [IJ] (Imm) $tp = (UpperHalf(Imm))
{takes the upper half of the immediate and stores it into the assembler temporary register}

ori [IJ] (Imm) $wr = $wr | (SignExt(Imm))
{Or's an immediate value with the working register and then puts it back into the working register}

orim [IJ] (Imm) $ma = (SignExt(Imm) | $tp)
{Or's an immediate value with the assembler temporary register and then puts it back into the memory address register}

stop [IJ] termination statement
{It sets every control signal to zero.}

seqi [IJ] (Imm) if($wr == inst[11:0]) PC = PC + 4
{skips the next instruction if the value of the working register is equal to the value given}

slti [IJ] (Imm) if($wr < inst[11:0]) PC = PC + 4
{If the working register is less than the provided immediate, skip the next instruction}

snei [IJ] (Imm) if($wr != inst[11:0]) PC = PC + 4
{skips the next instruction if the value of the working register is not equal to the value given}

subi [IJ] (Imm) $wr = $wr - (SignExt(Imm))
{subtracts an immediate from the value of the working register and stores it back in the working register}

Explanation of Call Conventions

Before a call is made, the user is expected save the arguments into memory, save the address of first argument into the argument register, save the number of arguments into the number of arguments register, and then call the function. Whenever something is saved onto the stack, the stack pointer must move down. When the function is over, the user will unload the stack and add to the stack pointer. Anytime someone accesses an array, they have to increment the memory register. The convention for this is whenever they are done accessing the array element, they must decrement the memory register the same amount as they incremented it.

Rule of Translating into Machine Language

This instruction set has three main addressing modes, Direct, Immediate, and PC relative. All of the [R] type instructions are direct and each opcode and funct is shown below. For the RegisterSelect, RegSelect2, Delta, and LocationSelect, they will either directly receive the value from the user, or if an instruction does not use one of these areas, it will assemble as zeroes. All [I] type instructions are immediate addressing and they only get an opcode and the immediate given by the user. The PC Relative Addressing is used by the goto and call commands. This is done by shifting the immediate in the instruction left by 1 and adding the first three bits of the PC onto the MSB of the immediate. So to find the value that gets added to the instruction you must reverse the process.

- ***Table of Instruction Set***

There are 30 instructions in total. 10 of them are R-type instructions, 7 of them are D-type instructions, and 13 of them are IJ-Type instructions.

NAME	MNEMONIC	FORMAT	OPCODE (hex)	FUNCT (hex)
Add	add	R	0	0
Add Immediate	addi	IJ	2	/
Add $ma	addm	IJ	3	/
And	and	R	0	1

Call	call	IJ	4	/
Clear R	clrr	D	1	0
Decrement R	decr	D	1	1
Go to	goto	IJ	5	/
Increment R	incr	D	1	2
Jump to return address	jr	R	0	2
Load Address	la	IJ	7	/
Load Upper Immediate	lui	IJ	8	/
Load Upper Imm. into $at	luia	IJ	9	/
Move	move	R	0	6
Move $ma	movem	R	0	7
Or	or	R	0	8
Or Immediate	ori	IJ	b	/
Or Immediate put into $ma	orim	IJ	c	/
Pop	pop	D	1	9
Push	push	D	1	a
Skip if equal	seq	R	0	b
Skip if equal to Imm.	seqi	IJ	d	/
Skip if greater than	sgt	R	0	c
Shift left	shl	D	1	d
Shift right	shr	D	1	e
Skip if less than	slt	R	0	f
Skip if less than Imm.	slti	IJ	a	/
Skip if not equal	sne	R	0	3
Skip if not equal to Imm.	snei	IJ	e	/

Subtract	sub	R	0	4
Subtract Immediate	subi	IJ	f	/
Stop the entire CPU	stop	IJ	6	/

Assembly Language Fragments for Common Operations

1. Loading an address

la 0x0002	0111 0000 000 0001 0

2. Conditional statement

Example: seq, skips the next instruction if the value of the working register is equal to the value of the register select register

if ($wr != Mem($ma)){
  $wr = $wr + $ma;
} else {
  $wr = $wr - $ma;
}
	seq $ma
	goto ifcode
	goto elsecode
	ifcode: 0x06
	add $wr, $ma, 0
	elsecode: 0x08
	sub $wr, $ma, 0
		0000 1011 001 0000 0
		0101 0000 000 0001 1
		0101 0000 000 0010 0
		
		0000 0000 000 0001 0
		
		0000 0100 000 0001 0

3. Iteration

Example: for-loop

for (int i = 1; i < 5; i ++){
  a = a+3; // assume i is in working
// register and a is at 0x20
}
	la 0x20
	Loop:
	addm 3
	addi 1
	seqi 5
	goto loop
	NEXT:
		0111 0000 001 0000 0
		
		0011 0000 0000 0011
		0010 0000 000 0000 1
		1101 0000 000 0010 1
		0101 0000 000 0010 0

4. Recursion

int sub5 (int n){
  if(n <= 0){
  return n;
  }
  return n + sub5(n - 5);
}
	Sub5: 0x0000
	  move $sp, $wr
	  addi 0x4
	  move $wr, $sp
	  push $ra, 0
	  move $ar, $wr
	  push $wr, 2
	  slti 0x0
	  goto recpart
	  seqi 0x0
	  goto recpart
	  goto basecase
	
	Recpart: 0x0016
	  addi -0x5
	  move $wr, $ar
	  call Sub5
	  pop $wr, 2
	  add $rv, $wr,1
	
	Basecase: 0x0020
	  move $wr, $rv
	  pop $ra, 0
	  jr $ra
		
		0000 0110 101 0000 0
		0010 0000 000 0010 0
		0000 0110 000 0101 0
		0001 1010 110 0000 0
		0000 0110 010 0000 0
		0001 1010 000 0010 0
		1010 0000 000 0000 0
		0101 0000 000 0101 1
		1101 0000 000 0000 0
		0101 0000 000 1011 0
		0101 0000 001 0000 0
		
		0010 1111 111 1101 1
		0000 0110 000 0010 0
		0100 0000 000 0000 0
		0001 1001 000 0010 0
		0000 0000 100 0000 1
		
		0000 0110 000 0100 0
		0001 1001 110 0000 0
		0000 0010 000 0011 0

Example Assembly Language Program: Find Relative Primes

Relprime: 0x0000
  clrr $ar
  movem $ar
  move $wr, $ma
  clrr $wr
  addi 2
  addm 0x2
  move $wr, $ma
  addm - 0x2
 	
loop: 0x000D
  call gcd
	
	0000 0110 101 0000 0
	0010 0000 000 0001 0
	0000 0110 000 0101 0
	0001 1010 110 0000 0
	0001 0000 000 0000 0
	0010 0000 000 0001 0
	
	0000 0110 010 0001 0
	0011 0000 000 0001 0
	0000 0110 001 0000 0

move $rv, $wr	0011 1111 111 1111 0
seqi 0x1	0100 0000 000 1100 0
goto inloop	0000 0110 100 0000 0
addm 0x2	1110 0000 000 0000 1
move $ma, $wr	0101 0000 000 1010 0
addm -2	0011 0000 000 0001 0
stop	0000 0110 001 0000 0
	0011 1111 111 1111 0
Inloop: 0x0028	0001 1001 110 0000 0
addm 0x2	0000 0010 000 0011 0
move $ma, $wr	
addi 0x1	
move $wr, $ma	0011 0000 000 0001 0
addm - 0x2	0000 0110 001 0000 0
	0010 0000 000 0000 1
Gcd: 0x0030	0000 0110 000 0001 0
incr $sp, 2	
push $ra, 0	
clrr $wr	0000 0110 101 0000 0
movem$ar	0010 0000 000 0001 0
move $ma, $wr	0000 0110 000 0101 0
snei 0x0	0001 1010 110 0000 0
goto ifTrue	0001 0000 000 0000 0
goto else	0000 0110 010 0000 0
	1110 0000 000 0000 0
ifTrue: 0x0042	0101 0000 001 0000 1
movem$ar	0101 0000 001 0011 1
addm 0x2	
move $ma, $rv	
pop $ra,0	0000 0110 010 0001 0
decr $sp, 2	0011 0000 000 0001 0
jr $ra	0000 0110 001 0100 0
	0011 1111 111 1111 0
Else: 0x004E	0001 1001 110 0000 0
movem$ar	0000 0010 000 0011 0
addm 0x2	
move $ma, $wr	
addm - 0x2	0000 0110 010 0001 0
seqi 0x0	0011 0000 000 0001 0
goto Done	0000 0110 001 0000 0
goto inWhile	0011 1111 111 1111 0
	1101 0000 000 0000 0
Done: 0x005C	0101 0000 001 0111 0
move $ma, $rv	0101 0000 001 1000 1

pop $ra, 0 decr $sp, 2 jr $ra	0000 0110 001 0100 0 0001 1001 110 0000 0 0000 0010 000 0011 0

inWhile: 0x0062
move $ma, $wr
addm 0x2
move $ma, $tp
addm - 0x2
sgt $tp
goto WhileIF
goto WhileElse
	0000 0110 001 0000 0
	0011 0000 000 0001 0
	0000 0110 001 0111 0
	0011 1111 111 1111 0
	0000 1100 111 0000 0
	0101 0000 001 1100 0
	0101 0000 001 1101 0

WhileIF: 0x0070
sub $wr, $tp, 0
move $wr, $ma
goto else
	0000 0000 000 0111 0
	0101 0000 001 0011 1

WhileElse: 0x0074
sub $tp, $wr, 0
addm 0x2
move $tp, $ma
addm - 0x2
goto else
	0000 0100 111 0000 0
	0101 0000 001 0011 1

Register Transfer Language Specification

Our design adopts a multicycle datapath.

Summary Table

- The table is separated into three pieces because it is too long.
- The multicycle RTL is broken down into five steps:
  Instruction fetch -> Instruction decode -> Register fetch -> (Optional steps) -> (Optional step only for push/pop)

Examples	add/sub/and/or	call	goto
Instruction fetch	IR = Mem[PC] PC = PC + 2		
Instruction decode	Imm = SE(IR[11:0])		
Reigister fetch	ALUOut = MUX[IR[7 : 5]] op MUX[IR[4 : 1]] (* op = +, -, &, |)	$ra = PC	PC = PC[15 : 13] || MUX([IR[11 : 0]) << 1
Depends on Instr	location = ALUOut	PC = PC[15 : 13] || MUX([IR[11 : 0]) << 1	

(table 1 of 3 - RTL Specification)

Examples	lui	luia	seq / sne	pop / push	jr
Instruction fetch	IR = Mem[PC] PC = PC + 2				
Instruction decode	Imm = SE(IR[11:0])				
Reigister fetch	ALUOut = Imm		ALUOut = $wr - MUX[IR[7:5]] isZero = Wire(ALUOut [0: 1])	ALUOut = $sp + Imm	PC = Reg[IR[7:5]]
Depends on Instr	$wr = ALUOut	$tp = ALUOut	If (isZero == 1): PC = PC + 2	C = Mem[ALUOut]	
Push/Pop Cycle				Reg[IR[7:5]] = C	

(table 2 of 3 - RTL Specification)

Examples	clrr	incr/decr	move	la
Instruction fetch	IR = Mem[PC] PC = PC + 2			
Instruction decode	Imm = SE(IR[11:0])			
Reigister fetch	ALUOut = 0	ALUOut = MUX[IR[7:5]] ± ZE(IR[4:1])	ALUOut = MUX[IR[4:1]]	ALUOut = MUX[IR[11:0]]
Depends on Instr	Reg[IR[7 : 5]] = ALUOut			
Push/Pop Cycle				

(table 3 of 3 - RTL Specification)

Components Needed for RTL Implementation:

Summary table

Name	*Quantity*	*Name*	*Quantity*	*Name*	*Quantity*
Sign Extender	1	Data Memory	1	MUX	10
Zero Extender	1	Register File	1	Instruction Memory	1
ALU	1	Register	22	Controller	1
Adder	3	/	/	/	/

Sign Extender: (SE in Lui and Luia)

Sign Extends a given input A (12 bits) to return a sign extended result R. (16 bit)

For different input A the sign extended result R will be different.

Zero Extender: (ZE in decr, Incr and La)

Sign Extends a given input A (12 bits) to return a zero extended result R. (16 bit)

For different input A the zero extended result R will be different.

ALU:

Main ALU to do the arithmetic operations. (ALUOut = A+B etc.)

For different inputs A (16 bits), B (16 bits) and ALUOp (4 bits) the result R (16 bits) and the zero (1 bit).

Adder:

1. An adder to increment PC by 2 (In instruction fetch, PC = PC+2)
2. An adder to increment PC by an additional 2 if a skip instruction is triggered. (for seq / sne instructions, If(Zero==1) PC = PC+2)

Instruction Memory:

The Instruction Memory stores all the instructions. It takes an input PC (16 bits) and returns the instruction at PC, Inst (16 bits)

For different values of PC we have different instructions returned.

(Each clock cycle fetches the instruction from the address specified by the PC, which wrote as IR = Mem[PC])

Data Memory:

The data memory stores data in memory. It takes MemWrite (1 bit), MemRead (1 bit), $ma (1 bit), Address (16 bits), outputs a data output (16 bits).

If MemWrite is 1, the Write data is written into the address stored by $ma. If MemRead is 1 then the data output has the value of the address stored by $ma.

Register File: (Reg)

The register file stores 8 registers. It takes RegWrite (1 bit), RegRead (1 bit), InputReg (16 bits), WriteData (16 bits) and outputs a data output (16 bits).

If RegWrite is 1, the WriteData is written into the address stored by InputReg. If RegRead is 1 then the data output has the value of the address stored by InputReg.

22 registers:

All registers are 16 bits wide. 8 of the registers will go into the register file for normal use ($wr, $ma, $ar, $na, $rv, $sp, $ra, $tp). One will be used for the PC to hold the current instruction.

The other registers will be used to store values as the instruction progresses since the machine is multi-cycle.

Controller

The controller decides all of the control signals for each instruction.

10 Muxes

The muxes will be placed throughout the datapath to choose what information goes where. The input signal that chooses which information to take is given by the controller.

List of Input Signals, Output Signals, and Control Signals:

Register: ALUOut (16 bits)
Register: PC (16 bits)
Instruction Register (IR) (16 bits)
Program Counter (PC) (16 bits)
isZero (1 bit) (flag)
isGreaterThan(1 bit) (flag)
isLessThan(1 bit) (flag)

Control Bits:
ALUInput1 (4 bits)
ALUInput2 (4 bits)
Delta (1 bit)
Extend (1 bit)
ALUOp (4 bits)
Pop/Push (1 bit)
RegWrite (1 bit)
MemWrite (1 bit)
Call (1 bit)
PC Select (2 bits)
PCWrite (1 bit)
IMWrite (1 bit)
IRWrite (1 bit)

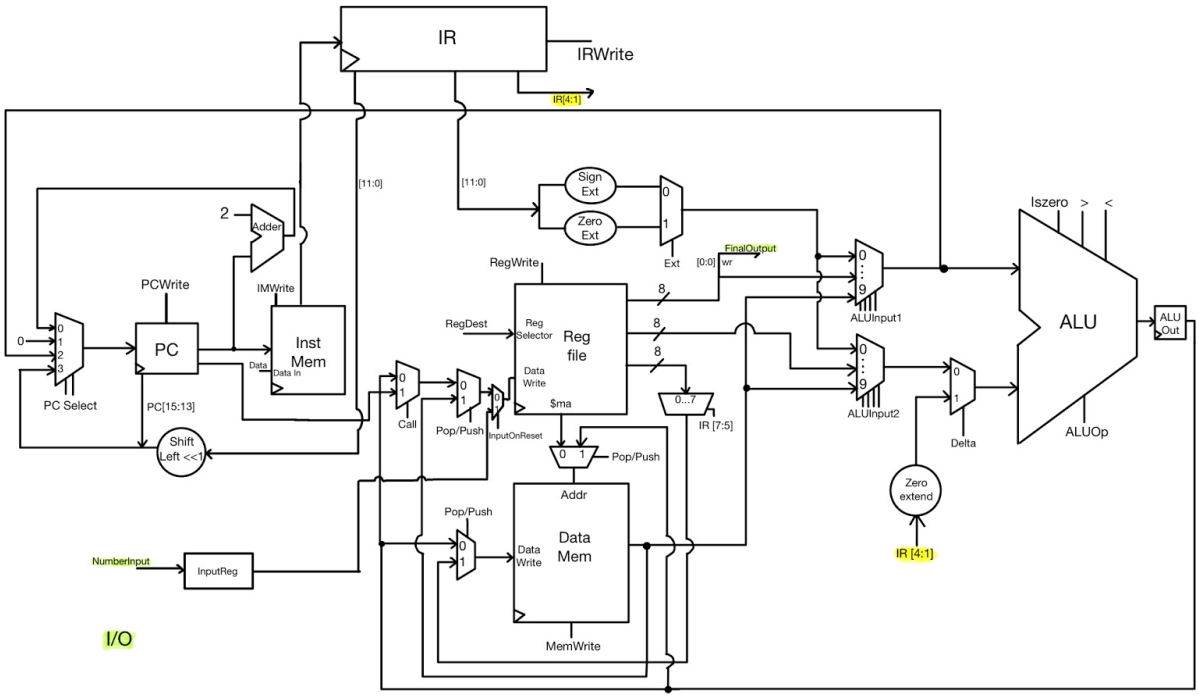
Double Checking the RTL:

To double check our RTL, we drew out a crude datapath and tried to logically follow all of our RTL to make sure that it was not only correct, but also possible with how we decided to use each type of instructions.

A note of changes made to the language specifications

- Jr (jump to return address) was mistakenly set to IJ-type. Now it is changed to R-type and a new set of opcode/funct is assigned.
- Remove the instruction “lv” because the instruction “move” can do the same function as lv could.
- Added “slti” instruction that will skip the next instruction if the working register is less than the immediate. We added this to make life easier for recursion and relprime.

Datapath Diagram



Description of Each Control Signal

ALUOp (3 bits)

The ALU needs to support seven actions: add, subtract, or, and, pass-through, zero-out. The pass-through will not make any changes to the input. The zero-out will output all zeros.

ALUInput1 (4 bits) and ALUInput2 (4 bits)

Both of them are 4 bits because they need to select data from the 10 sources. There are 10 wires in total with 8 data coming from the Register File, one from Zero / Sign Extension and one from the Data Memory. Both ALUInputs select data from the10 wires so that the ALU receives the correct data.

Signal	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001
Source	$wr	Mem	$ar	$na	$rv	$sp	$ra	$tp	Ext*	$ma

*Ext refers to the extended immediate (either sign or zero extended)

Delta (1 bit)

The Delta is either a 0 or 1. If it is set to 0, The ALU takes value from the ALUInput2. If it is set to 1, the ALU takes value from zero-extended delta value of D-Type instructions. The delta value form D-Type instructions works as an immediate.

Ext - Extend (1 bit)

The Extend is either a 0 or 1. If it is set to 0, the ALU takes the sign-extended immediate value from IJ-Type instructions. If it is set to 1, the ALU takes the zero-extended immediate value.

Pop/Push (1 bit)

The Pop/Push is either a 0 or 1. It controls three different MUXes. If it is 0, there will be no pop or push actions. If it is set to 1, pop or push are activated.

RegWrite (1 bit)

RegWrite can be either a 0 or 1. If it is 0, the Register File cannot be modified. If it is 1, the Register File will be open to modifications. The Register File will be modified via the Data Write port. There is no RegRead signal since the Register File will always be read.

MemWrite (1 bit)

MemWrite can be either a 0 or 1. If it is 0, the Data Memory cannot be modified. If it is 1, the Data Memory will be open to modifications. The Data Memory will be modified through the Data Write port. There is no MemRead signal since the Memory will always be read at the input address of memory address register.

Call (1 bit)

Call can be either a 0 or 1. If it is 0, it is not activated. If it is 1, it will pass the PC into the Register File in order to record the $ra.

PC Select (2 bits)

PC Select will select from 4 input signals. Input 0 comes from the value of PC + 2. Almost every instruction needs a PC + 2 since we will execute the next line of instruction every time. Input 1 is the value of PC + 4, for skip instructions like seq and sgt because all the skip instructions will skip one line of instructions if the result is true. Input 2 is the

output of ALUInput1. For the last instruction of every program, “jr $ra”, the address of $ra will pass through the ALUInput1 and then be transferred to PC via Input 2. For Input 3, when there is a big jump to another address, the PC should receive the target address from Input 3.

IRWrite (1 bit)

Tells the IR register to write the value of the instruction memory to hold throughout the instruction.

PCWrite (1 bit)

The PCWrite control bit specifies when the PC can be written to, so it does not constantly update during an instruction cycle.

IMWrite (1 bit)

The IMWrite bit tells the instruction memory when data can be written into the memory through the data in port.

Description of Each Component

PC: (PCRegister)

The PC is a register that holds that value of the current instruction address. It will get incremented by 2 bytes after every instruction is finished since we are using 16 bit addresses.

Instruction Memory:

This is a block of memory that holds the program memory and exception handling memory. The program memory section starts at address 0x4001 and ends at 0xFF38. This leaves 200 bytes for exception handling. The memory block outputs it’s instructions in several busses. The bits [11:0] are outputted twice, one to be used as an immediate value, and another for PC-relative addressing with goto and call commands. Bits [15:12], [11:8] , and [0] are always fed into the controller as the opcode, funct, and location select. Lastly, Bits [7:5] and [3:1] are fed into the register file to determine which register will be written too.

There is a dataIn port controlled by IMWrite signal.

Data Memory:

This block of code is used for holding variables and items on the stack. The data memory starts at 0x0 and ends at 0x4000. The stack starts at 0x4000 and increments downward as described by the programmer. Data memory always takes in the $ma register unless the instruction is a push/pop, since memory is so commonly used. The block also has data write input that takes in the data that wants to be written to memory. It’s only control signal is called MemWrite, which decides whether or not to write to the memory. Lastly, the memory block always outputs to the 2 muxes and can be chosen by the ALUInput’s if the controller decides that it needs to be.

Register File:

The register file holds 8 registers, the working register, memory address register, argument address register, number of arguments register, stack pointer, return value register, return address register, and assembler temporary register. Each register is always fed into the 2 muxes before the ALU and the controller will decide which one is required. The only control bit connected to the register file is the regWrite bit, which controls whether its registers are able to be written too. Its only other inputs consist of a 16 bits data write input that holds the information that will be written to a register, and a 3 reg selector that chooses which register is being written to.

ALU:

The ALU takes in 2 inputs and produces some output based off of its ALUop. Our ALU should be able to support adding, subtracting, anding, oring, zeroing, shifting, and pass through. The ALU also has 3 flags, isZero,isGreaterThan, and isLessThan. The isGreaterThan and isLessThan will determine whether the first input is greater than or less than the other and set the flag accordingly. The isZero flag is set once a computation is completed. It will be 1 if the output is equal to 0. There are two other small ALU's whose only job is to add 2 to the PC. This needs to happen twice, just in case the PC needs to skip an instruction.

ALU Operations			
Opcode	Operation	Opcode	Operation
0	add	4	Shift left
1	sub	5	Shift right
2	and	6	Zero
3	or	7	Pass through

IR: (Instruction Register)

For each instruction, IR only changes once under the control of IRWrite. It is a special register that stores the decoded instruction.

Changes Made to the RTL Descriptions - M3

- Made the second cycle the same for every instruction such that now all eight registers are fetched and passed onto the next stage.
- Add a new push/pop cycle that is used by the push and pop instructions only.

Specifications of the Control Units

- We need 13 signals for 13 different spots: ALUInput1 (4 bits), ALUInput2 (4 bits), ALUOp (4 bits), PC Select (2 bits), Delta (1 bit), Extend (1 bit), Pop/Push (1 bit), RegWrite (1 bit), MemWrite (1 bit), Call (1 bit), PCWrite (1 bit), IMWrite (1 bit), IRWrite (1 bit).
- Most control signals only control one component or one mux. For Pop/Push, it is connected to three different muxes.
- When we implement the control unit, there will be a case statement for all the control bits based on different instructions.

Changes Made to the RTL Descriptions - M4

- The divide, multiply, and xor instructions are gone. The divide and multiply is gone since we already have the shift left and shift right instructions. The xor instruction is seldom used. The main ALU needs fewer operations without these three instructions.
- The wailing wall (referring to the giant wall of 10 registers) is gone. In the instruction decode step, we were going to save all ten values in the wailing wall. However, these ten values are selected immediately after the wailing wall meaning there is no point to save those data. Then we just pass those data into the ALUInput1 (mux) and ALUInput2 (mux) via wires.

Plans to Implement Hardware

Parts	**Plan of how to implement in hardware**
Adder	The 16-bit adder is easy to implement using 16 1-bit adder.
ALU	We can use the 16 bit ALU from the book and modify it to fit our own design.
Left Shifter	It will have a 12 bit input and 13 bit output with an extra zero at the end.
Memory	We will learn from lab 07.
2-to-1 Mux	We can use the Mux from the book and modify it to fit our own design.
3-to-1 Mux	We can use the Mux from the book and modify it to fit our own design.

4-to-1 Mux	We can use the Mux from the book and modify it to fit our own design.
8-to-1 Mux	We can use the Mux from the book and modify it to fit our own design.
10-to-1 Mux	We can use the Mux from the book and modify it to fit our own design.
Register File	We can use the Register file from the Resources page on the course website and modify it to fit our design.
Sign Extender	It takes in a 12 bit number and fills 4 bit 1's or 0's to the top the same as the top digit. The output will be 16 bits.
Zero Extender	It takes in a 12 bit number and fills 4 bit 0's to the top. The output will be 16 bits.
PC Register	It will be a normal register with special outputs: top 3 digits, or full address.
Instruction Register	It will be a normal register which holds the data. It has several special outputs: IR[11:0], IR[7:5], and IR[4:1].
Control Unit	It takes in the opcode and outputs the control components. We can work on all possible conditions and then implement the control unit using verilog.

Descriptions of the Tests to Verify the Control Units

To test the control unit, we will have to exhaustively test each instruction and check that the control signals match to what they should be. So, for each instruction, we can have a for loop that will run through each register that is possible to be called for the R and D types, and then check the waveform to make sure that the control signals are correct. Then, for the IJ types, we have to run each instruction and make sure that the information is being written to the correct register/memory address, and that each control signal is correct.

Unit Test Description

For our components, we have created a lot of tests to see if we can get the output as what we expected by the given inputs. The description of each test and example tests are shown in the table below. Each component is going to be tested by making a test bench in Verilog and we list the corresponding file names for test benches.

Parts	**File Name**	**Description of Tests**	**Example**
Adder	Adder4b.v	It has two inputs and the sum will be the output.	Input1: 0000000000000001 Input2: 0000000000000110 Output: 000000000000111

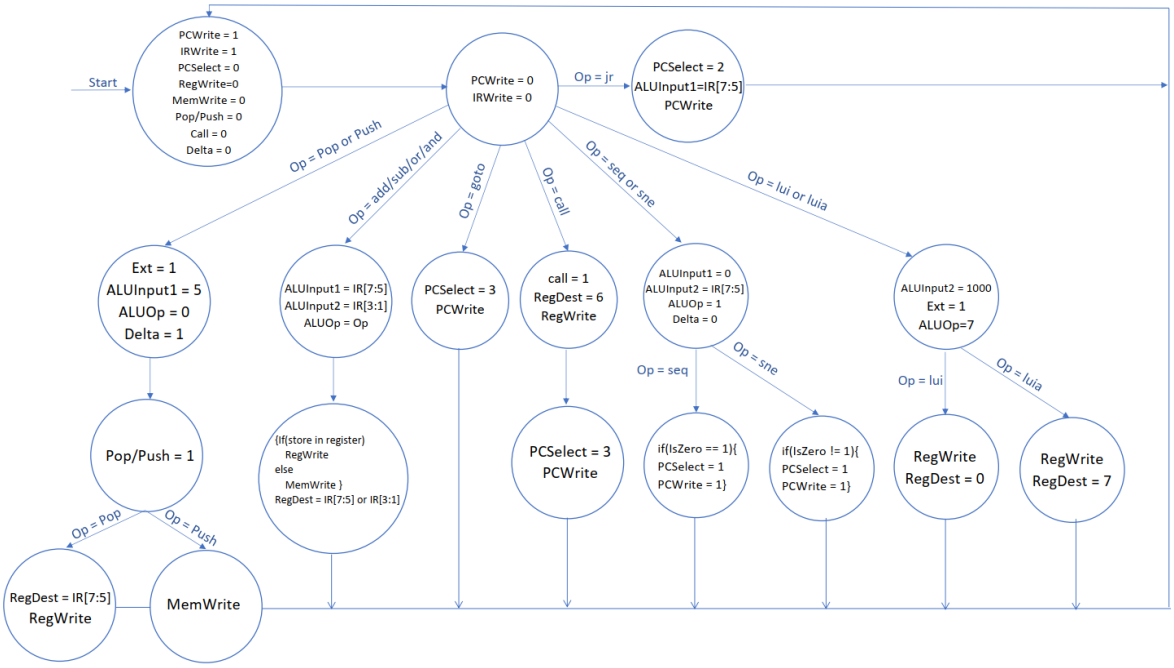
ALU	ALU4b.v	It has three flags at the result to compare input A with input B. It will be capable of doing eight operations: add, sub, and, or, shift left, shift right, output all zeros, and pass through. (Pass through will only pass through the data from input B, always)	ALUOp=000 (ADD) Input1(A) = 0000000000000000 Input2(B) = 0000000000000000 Output1(R) = 0000000000000000 Output2(isZero) = 1 Output3(islessThan)=0 Output4(isgreaterThan)=0 ALUOp=001 (SUB) Input1(A) = 1111111111111111 Input2(B) = 1111111111111111 Output1(R) = 0000000000000000 Output2(isZero) = 1 Output3(islessThan)=0 Output4(isgreaterThan)=0 ALUOp=010 (AND) Input1(A) = 10011100101101010 Input2(B) = 01100011010010101 Output1(R) = 0000000000000000 Output2(isZero) = 0 Output3(islessThan)=1 Output4(isgreaterThan)=0
Left Shifter	Left_Shifter.v	It takes a 12 bit input and conjugates a zero to the end. The outputs will be 13 bits.	Input: 111111111111 Output: 1111111111110 Input: 001100110011 Output: 0011001100110
Memory	Memory.v	It writes on the negative edge with MemWrite is on and reads on the positive edge. The output will be the data stored in the memory at specified address.	If MemWrite == 1 Input = 1010101010101010 Then output = 1010101010101010 If MemWrite == 0 Input = 000000000000000 Output is still 1010101010101010

2-to-1 Mux	mux_16b_2input.v	It takes two inputs value to each 16-bit bus and a 1-bit selector. Check to see if we can get the expected output.	Input1(A) = 0111000101110000 Input2(B) = 0000000101110010 Input3(Op) = 0 Output = 0111000101110000
3-to-1 Mux	mux_3b_3input.v	It takes three inputs value to each 3-bit bus and a 2-bit selector. Check to see if we can get the expected output.	Input1(A) = 011 Input2(B) = 000 Input3(C) = 001 Input4(Op) = 01 Output = 000
4-to-1 Mux	mux_16b_4input.v	It takes four inputs value to each 16-bit bus and a 2-bit selector. Check to see if we can get the expected output.	Input1(A) = 0111000101110000 Input2(B) = 0000000101110010 Input3(C) = 0000000100000000 Input4(D) = 0111000100000000 Input5(Op) = 00 Output = 0111000101110000
8-to-1 Mux	mux_16b_8input.v	It takes eight inputs value to each 16-bit bus and a 3-bit selector. Check to see if we can get the expected output.	Input1(A) = 0111000101110000 Input2(B) = 0000000101110010 Input3(C) = 0000000100000000 Input4(D) = 0111000100000000 Input5(E) = 0000000000000000 Input6(F) = 0000000001110000 Input7(G) = 0101010100000000 Input8(H) = 0111000000001100 Input9(Op) = 001 Output = 0000000101110010
10-to-1 Mux	mux_16b_10input.v	It takes eight inputs value to each 16-bit bus and a 4-bit selector. Check to see if we can get the expected output.	Input1(A) = 0111000101110000 Input2(B) = 0000000101110010 Input3(C) = 0000000100000000 Input4(D) = 0111000100000000 Input5(E) = 0000000000000000 Input6(F) = 0000000001110000 Input7(G) = 0101010100000000 Input8(H) = 0111000000001100 Input9(I) = 1101010100000111 Input10(J) = 1111001110001100 Input11(Op) = 0010 Output = 0000000100000000

Register File	Register_File.v	It will store something in a specific register. Load the data stored in that register. Repeat multiple times. If the data is the same all the time, then it works.	Input1(regDest) = 000 Input2(DataWrite) = 0101010100000000 Input3(regWrite) = 1 Output(wr) = 0101010100000000
Sign Extender	Sign_Extender.v	It takes in a 12 bit number and fills in 4 bit 1's or 0's based on the top digit.	Input= 101010101010 Output = 1111101010101010 Input = 010101010101 Output = 0000010101010101
Zero Extender	Zero_Extender.v	It takes in a 12 bit number and fills 4 bit zeros to make a 16 bit output.	Input = 101010101010 Output = 0000101010101010
PC Register	PCRegister.v	We tests the outputs are the expected digits based on input.	Input1(Din) = 001000010010110 Input2(PCWrite) = 1 Output1(Out) = 001000010010110 Output2(TopOut) = 001
Instruction Register	InstructionRegister.v	We tests the outputs are the expected digits based on input.	Input = 0001011010100110 Output1(RegSelect) = 101 Output2(RegSelect2) = 011 Output3(Imm) = 011010100110 Output4(Delta) = 0011 Output5(LocationSelect) = 0 Output6(Opcode) = 0001 Output7(funct) = 0110
Controller	Control.v	It will input the instruction components for different type of instructions and get the data that represent the information to the corresponding	Input1(Opcode) = 0 Input2(funct) = 0 Input3(RegSelect) = 1 Input4(locationSelect) = 0 Output(writeToMem) = 1

		instruction according to different state	
Mini Controller	miniController.v	It's the controller dealing with seq, seqi, sgt, slt, slti, sne and snei. It will take in several inputs and output if skip or not.	Input1(Opcode) = 0000 Input2(funct) = 1011 Input3(isZero) = 1 Output(skip) = 1

Control Signals (for partial instructions)



System Test Plan

To test all the parts in our datapath, we'll put at least one instruction in the memory. We're going to test to see if we can successfully get the instruction information, run through the datapath, get the output, and then go to the next instruction accurately until all the instructions are done. In addition, we need to make sure that when we run branches and jumps, it can go to the correct location and follow the previous steps to complete all the instructions.

Changes Made to the RTL Descriptions - M5

There were no changes made to the RTL Description in this Milestone.

Changes Made to the Datapath - M5

There were no changes made to the datapath in this Milestone.

Changes Made to the Control Unit Design - M5

There were no changes made to the control unit design in this Milestone.

Facts about our Euclid's Algorithm and relPrime

1.We need 146 bytes to store both Euclid's algorithm and relPrime. We need 8 bytes of memory variables.

2. The total number instructions executed when relPrime is called with 0x13B0: 153227.

3. The total number of cycles required to execute relPrime under the same conditions as Step 2 is 612908.

4. The average cycles per instruction is 4.

5. The cycle time is 34 ns

6. The total execution time for relPrime under the same conditions as Step 2is 19 ms

7. The device utilization summary

Phase4_Verilog Project Status (02/19/2020 - 14:28:38)			
Project File:	Phase4.xise	**Parser Errors:**	No Errors
Module Name:	Phase4_Verilog	**Implementation State:**	Synthesized
Target Device:	xc3s500e-4fg320	**• Errors:**	No Errors
Product Version:	ISE 14.7	**• Warnings:**	51 Warnings (0 new, 0 filtered)
Design Goal:	Balanced	**• Routing Results:**	
Design Strategy:	Xilinx Default (unlocked)	**• Timing Constraints:**	
Environment:	System Settings	**• Final Timing Score:**	

Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	652	4656	14%
Number of Slice Flip Flops	240	9312	2%
Number of 4 input LUTs	1250	9312	13%
Number of bonded IOBs	313	232	134%
Number of BRAMs	16	20	80%

Appendix B

Design Process Journal

Integration:

1. The first task is to combine 2 adders, a PC register, Shift Left, a PC Select MUX (4 inputs to one output) and an Instruction memory.

2. The next addition is to add the register file and Data memory. We connect these with a call MUX, three pop/push MUXs, location select MUX and the data write MUX.

3. The third step is to add the sign and zero extenders (connected with an Ext MUX) and a wall of 10 registers that will store each register, the immediate selected by the extenders and the data memory (The Wailing Wall).

4. The last edition is the main ALU which is attached to its two inputs using the ALUInput1 and ALUInput2 MUXs. Additionally, the second input to the ALU is also selected between the Zero Extended delta and the registers using a Delta MUX.

Affect That Architecture Had On Datapath:

1. Since our architecture was a multicycle implementation of the Accumulator, our datapath is built to be optimised towards the $ma register we had.

2. One of the bigger design decisions was The Wailing Wall. As the controller doesn't finish until the third cycle, we forward all the registers to the next cycle. This doesn't impact efficiency as there are only eight registers and choosing between then doesn't take too long.

Changes to the Integration Plan (M4)

We do not have any update on the Integration Plan for M4.

Changes to the Integration Plan (M5)

- Phase 3 was changed to be the ALU and its various input parts.

- Phase 4 was changed to be the integration the phases 1, 2 and 3 with the control unit