MERC-16 Architecture Manual

Nicklas Carpenter, Jake Evans, Craig McGee Jr., Angel Rivera

Published: November 20, 2019

Contents

Executive Summary

Hello, and welcome to our design document! In this document you will learn both the ins and outs of our MERC-16 processor. The MERC-16 processor is a load/store, multicycle, 16-bit processor. There are a total of 22 instructions with 4 different instruction types (J-Type, R-Type, I-Type, L-Type) and 16 registers total. The MERC-16 can perform arithmetic, comparison, jump, and memory operations.

Chapter 1

System Description

1.1 Multi-Cycle RTL

Refer to **rtl.xlsx** in Design Directory

1.2 Components

Component	Inputs	Output	Description
ALU	A(16-bits), B(16-bits), Op(3-bit)	Out(16-bits)	This component is used for the main processing functions done for what is written in code. It included two 16-bit inputs and one 16-bit output. There are individual 1-bit control signals for each of the inputs and a 3-bit operation control signal for the ALU itself so that it can do said operation. Note that the operation control is not the same as the opcode found in the machine code
Register	Din(16-bit), En(1-bit)	Dout(16-bit)	Component that stores values. Writing to the register can only occur when En is acitve
Sign Extender	In(4-bits)	SE(16-bit)	Takes a 4-bit two's complement integer and returns a 16-bit equivalent with the sign of the integer preserved
Zero Extender	In(4-bits)	ZE(16-bit)	Takes a 4-bit two's complement integer and returns a 16-bit equivalent with the sign upper 12 bits set to 0
Left-Shifter	In(16-bit)	LS(16-bit)	Takes a 16-bit binary number and preforms a logical left shift (the most-significant) bit is discarded, each bit is shifted left, and the least-significant bit is set to 0)

1.3 Subsystems

Subsystem	Description
Register File	Composed of registers and combinational logic to handle addressing, input output handling, and control handling
Memory	Composed of a block memory unit and combinational logic to handle input output and control handling
Decode Subsystem	Composed of the Register file, and the combinational logic for register and immediate output (including sign and zero extension as well as left-shifting).

Subsystem	Inputs	Outputs	Controls
Register File	WriteAddr(16-bit), ReadAddrA(16-bit), ReadAddrB(16-bit), WriteData(16-bit)	RegOutA(16-bit), RegOutB(16-bit)	Dest(1-bit), Write(1-bit)
Memory	ReadAddr(16 bit)	MemOut(16-bit)	Read(1-bit), Write(1-bit), ToReg(1-bit), WR_Data(1-bit)
Decode Subsystem	Instr(16-bit), WR_Data(16-bit)	A(16-bit), B(16-bit), ZE(16-bit), SE(16-bit), SE1L(16-bit)	WR_EN(1-bit), Short_rs(1-bit)

1.4 Controls

1.4.1 State Machine

Refer to **control.png** in Design Directory

1.4.2 Signals List

Control Signal	Description
Inst/Data	This control signal is used to tell the multiplexer whether we want to go to our instruction memory or whether we want to go to our data memory to do things like write or read in memory
ALUOp	This control signal is used to tell the ALU what operation we are doing. Such operations include, add, subtract, shift instructions, and logical instructions
ALUSrcA	This control signal is used to determine whether we want the ALU to take whatever value from the register that we read into A or take the PC to do an operation in the ALU
ALUSrcB	This control signal is used to determine whether we want the ALU to take whatever value from the register that we read into B, take the number 2 for incrementing purposes, or the sign/zero extended immediate for other operations
RegData	This control signal is used to determine whether we are taking whatever ALUOut is as the data to write into the register, or we take the data from memory as our write data, or the value of the PC, or the value for lui and llli operations to write into the register
RegDest	This control signal is used to determine which register we want to write data to if we have enabled writing to a register
IRWrite	This control signal is used to enable or disable writing to the instruction register
RegShort	This control signal is used to determine whether we are taking the 4 bit rs value or the shortened 3 bit rs value
PCSrc	This control signal is used to determine whether we are taking the ALUOut value for jump target, the branch value to set PC equal to it, or the next instruction address
PCWrite	This control signal is within the control of the PC and is used to enable or disable writing to the PC

MemWrite	This control signal is used to enable or disable writing to memory
MemRead	This control signal is used to enable or disable reading from memory
RegWrite	This control signal is used to enable or disable writing to a register
ZE/SE	This control signal is used to determine whether we want the sign extended immediate or the zero extended immediate
Upper/Lower	This control signal is used to pick between picking the computation for our lui instruction or our lli instruction
Beq	This control signal is within the control of the PC and is used to enable or disable the writing to the PC the value from the beq instruction
Bne	This control signal is within the control of the PC and is used to enable or disable the writing to the PC the value from the bne instruction
Blt	This control signal is within the control of the PC and is used to enable or disable the writing to the PC the value from the blt instruction
Bgt	This control signal is within the control of the PC and is used to enable or disable the writing to the PC the value from the bgt instruction
Blet	This control signal is within the control of the PC and is used to enable or disable the writing to the PC the value from the blet instruction
Bget	This control signal is within the control of the PC and is used to enable or disable the writing to the PC the value from the bget instruction

1.5 Datapath

Refer to **datapath.png** in Design Directory

Chapter 2

Instruction Set Reference

2.1 Instruction Format

2.1.1 I-Type

This type is using a register and immediate value and store the result into another register. rs is the register being used in conjunction with the immediate value, and rd is the destination register where the result is store when doing the instructions specified in the opcode.

I	opcode	rd	rs	imm
	5	4	3	4

2.1.2 R-Type

This type is using two registers and storing the result from doing the instructions specified in the opcode into a separate register. rs and rt are the two registers that are being used in the initial process; however, rt deals with only the first 8 out of the 16 possible registers due to the smaller bit size the type can handle.

R	opcode	rd	rs	rt
	5	4	4	3

2.1.3 J-Type

This type allows us to jump to a specific address in the code.

J	opcode	psuedo-address
	5	11

2.1.4 L-Type

Used to load half word immediates into registers.

L	opcode	rd	imm.
	5	3	8

2.2 Addressing Modes

2.2.1 Register Direct

Used in all instructions. The value passed to rd, rd, or rt is the address of the specified register in the register file.

2.2.2 Psuedodirect

Used in jumping instructions. The 16-bit value written to the PC is composed as follows:

Jump Address:	PC[15:11]	SE(imm) << 1
	5	11

2.2.3 PC Relative

Used in all branching instructions. The sign-extended value of the immediate is added to the value of the PC and then, if the branch is take, the result is written to the PC.

2.2.4 Base + Offset

Used in memory instructions. The address in memory that is written to or read from is the address specified in the operand register added with the offset specified in the immediate.

2.3 Register Definitions

Address	Register Name	Description
0	zero	This register is always set to 0, or connected to ground, and cannot be changed to any other value.
1	sp	This register is called the stack pointer, which is used to adding/removing stacks in order to insert certain values/registers into said stacks
2-4	t0, t1 t2	These are temporary registers which are not carried through the call, so only use these for brief use in the code
5	ra	This register is used to return to a specific address in the code, which can be overridden when jumping from procedure to procedure
6-8	s0, s1, s2	These are saved registers which are carried through the call, but they can be overridden
9-10	rv0, rv1	These registers are used to store return values in a call; they are carried over after a call has been executed
11-12	arg0, arg1	These registers are used to store argument values before entering a call; they are carried over into the call but can also be overridden within the call
13	at	This register is saved for the assembler and are reserved for handling pseudo-instructions
14-15	k0, k1	This register is saved for the kernel and should not be used by the program

2.4 Instruction List

2.4.1 Add (add)

Description

Adds values of two registers and stores in a register.

Syntax

```
add $rd, $rs, $rt
```

Format

R	0b00000	rd	rs	rt
	5	4	4	3

RTL

newPC = PC + 2
PC = newPC
IR = Mem[PC]
A = Reg[IR[6-3]]
B = Reg[IR[2-0]]
Result = A + B
Reg[IR[10-7]] = result

2.4.2 Add Immediate (addi)

Description

Adds value from a register and an intermediate value and stores in a register.

Syntax

```
addi    $rd, $rs, 3
```

Format

I	0b00001	rd	rs	imm.
	5	4	3	3

RTL

newPC = PC + 2
PC = newPC
IR = Mem[PC]
A = Reg[IR[6-3]]
B = SE(IR[3:0])
Result = A + B
Reg[IR[10-7]] = Result

2.4.3 AND (and)

Description

Bitwise AND between 2 registers and stores result in a register.

Syntax

```
and   $rd, $rs, $rt
```

Format

R	0b01100	rd	rs	rt
	5	4	4	3

RTL

newPC = PC+2
PC = newPC
IR = Mem[PC]
a = Reg[IR[6-3]]
b = Reg[IR[2-0]]
result = a & b
Reg[IR[10-7]] = result

2.4.4 Branch if Equal To (beq)

Description

Branches the specified number of instructions if the compared values are equal.

Syntax

```
beq    $rd, $rs, 5t
```

Format

I	0b00101	rd	rs	imm.
	5	4	3	3

RTL

```
NextPC = PC + 2
IR = M[PC]
Reg[IR[10:7]]
B = Reg[IR[6:4]]
If (A == B)
PC = newPC + SE(IR[0:3])
Else
PC = NextPC
```

2.4.5 Branch if Greater Than (bgt)

Description

Branches the specified number of instructions if the value of the first operand is greater than the second.

Syntax

```
bge    $rd, $rs, 5t
```

Format

I	0b10011	rd	rs	imm.
	5	4	3	3

RTL

```
NextPC = PC + 2
IR = M[PC]
Reg[IR[10:7]]
```

```
B = Reg[IR[6:4]]
If (A ¿ B)
PC = newPC + SE(IR[0:3])
Else
PC = NextPC
```

2.4.6 Branch if Greater Than or Equal To (bget)

Description

Branches the specified number of instructions if the value of the first operand is greater than or equal to the second.

Syntax

```
bget   $rd, $rs, 5t
```

Format

I	0b10101	rd	rs	imm.
	5	4	3	3

RTL

```
NextPC = PC + 2
IR = M[PC]
Reg[IR[10:7]]
B = Reg[IR[6:4]]
If (A ¿= B)
PC = newPC + SE(IR[0:3])
Else
PC = NextPC
```

2.4.7 Branch if Less Than (blt)

Description

Branches the specified number of instructions if the value of the first operand is less than the second.

Syntax

```
ble   $rd, $rs, 5t
```

Format

I	0b10010	rd	rs	imm.
	5	4	3	3

RTL

```
NextPC = PC + 2
IR = M[PC]
Reg[IR[10:7]]
B = Reg[IR[6:4]]
If (A ¡ B)
PC = newPC + SE(IR[0:3])
Else
PC = NextPC
```

2.4.8 Branch if Less Than or Equal To (blet)

Description

Branches the specified number of instructions if the value of the first operand is less than the second.

Syntax

```
blet  $rd, $rs, 5t
```

Format

I	0b10010	rd	rs	imm.
	5	4	3	3

RTL

```
NextPC = PC + 2
IR = M[PC]
Reg[IR[10:7]]
```

B = Reg[IR[6:4]]
If (A ¡= B)
PC = newPC + SE(IR[0:3])
Else
PC = NextPC

2.4.9 Branch if Not Equal To (bne)

Description

Branches the specified number of instructions if the compared values are equal.

Syntax

```
bne   $rd, $rs, 5t
```

Format

I	0b10010	rd	rs	imm.
	5	4	3	3

RTL

NextPC = PC + 2
IR = M[PC]
Reg[IR[10:7]]
B = Reg[IR[6:4]]
If (A != B)
PC = newPC + SE(IR[0:3])
Else
PC = NextPC

2.4.10 Call (call)

Description

BJumps from current address to the address calculated and stores return address in register $5

Syntax

```
call    destination
```

Format

J	0b00100	destination
	5	11

RTL

```
NextPC = PC + 2
IR = M[PC]
Reg[IR[10:7]]
B = Reg[IR[6:4]]
If (A == B)
PC = newPC + SE(IR[0:3])
Else
PC = NextPC
```

2.4.11 Compare (comp)

Description

Adds values of two registers and stores in a register.

Syntax

```
comp    $rd, $rs, $rt
```

Format

R	0b01111	rd	rs	rt
	5	4	4	3

RTL

```
newPC = PC+2
PC = newPC
IR = Mem[PC]
a = Reg[IR[6-3]]
```

```
b = Reg[IR[2-0]]
if (a == b) result = 0
else if (a ¡ b) result = -1
else result = 1
Reg[IR[10-7]] = result
```

2.4.12 Jump (j)

Description

Jumps from current address to the address calculated.

Syntax

```
j  destination
```

Format

J	00011	destination
	5	11

RTL

```
newPC = PC + 2
IR = Mem[PC]
result = PC[15-11] + SE(IR[10-0]) ¡¡ 1
PC = result
```

2.4.13 Jump Register (jr)

Description

Jumps from current address to the address calculated.

Syntax

```
j  $rs
```

Format

R	0b10110	xxxx	rs	xxxx
	5	4	4	3

RTL

newPC = PC + 2
IR = Mem[PC]
a = Reg[IRr[10-0]]
result = a
PC = result
Reg[IR[10-7]] = result

2.4.14 Load Word (load)

Description

Loads word from address in memory and stores in a register

Syntax

```
lw   $rd, $rs, imm
```

Format

I	0b00111	rd	rs	imm.
	5	4	3	3

RTL

NextPC = PC + 2
IR = M[PC]
A = Reg[IR[10-7]]
B= Reg[IR[6-4]]
R = A + SE(IR[3-0])
Mout = M[R]
Reg[B] = Mout

2.4.15 Load Lower Immediate (lli)

Description

Loads the eight bit immediate into the lower eight bits of the specified register.

Syntax

```
lui   $rd , imm
```

Format

L	0b01000	rd	imm.
	5	3	8

RTL

NextPC = PC + 2
IR = M[PC]
D = IR[10-8]
R[D][7:0] = IR[7-0]

2.4.16 Load Upper Immediate (lui)

Description

Loads an eight bit immediate into the upper eight bits of the specified register.

Syntax

```
lui   $rd, immn
```

Format

L	0b01001	rd	imm.
	5	3	8

RTL

NextPC = PC + 2
IR = M[PC]
D = IR[10-8]
R[D][15:8] = IR[7-0]

2.4.17 No Operation (nop)

Description

An instruction that has no effect on when executed.

Syntax

`nop`

Format

0b01110	rd	rs	rt
5	4	4	3

RTL

NextPC = PC + 2
IR = M[PC]
PC = NewPC

2.4.18 NOT (not)

Description

Inverts the bits of the source register and stores the result in the destination register.

Syntax

`nop`

Format

R	0b01101	rd	rs	xxx
	5	4	4	

RTL

NextPC = PC + 2
IR = M[PC]
PC = NewPC

2.4.19 OR (or)

Description

Bitwise OR between 2 registers and stores result in a destination register.

Syntax

```
or    $rd, $rs, $rt
```

Format

R	0b01011	rd	rs	xxx
	5	4	4	

RTL

newPC = PC + 2
IR = M[PC]
A = Reg[IR[6-3]]
B = Reg[IR[2-0]]
D = Reg[10-7]
Reg[D] = A | B

2.4.20 Shift Left Logical (sll)

Description

Shifts a register value left by the amount provided in the immediate and places the result into the destination register.

Syntax

```
sll   $rd, $rs, 2
```

Format

I	0b10000	rd	rs	imm.
	5	4	3	3

RTL

newPC = PC+2
PC = newPC
IR = Mem[PC]
a = Reg[IR[6-4]]
shamt = ZE[IR[3-0]]
result = a ¡¡ shamt
Reg[IR[10-7]] = result

2.4.21 Shift Right Logical (srl)

Description

Shifts a register value right by the amount provided in the immediate and places the result into the destination register.

Syntax

```
sll   $rd, $rs, imm
```

Format

I	0b10001	rd	rs	imm.
	5	4	3	3

RTL

newPC = PC+2
PC = newPC
IR = Mem[PC]
a = Reg[IR[6-4]]
shamt = ZE[IR[3-0]]
result = a ¿¿ shamt
Reg[IR[10-7]] = result

2.4.22 Store Word (store)

Description

Stores value of the source register into an address in memory.

Syntax

```
store   $rd, $rs, imm.
```

Format

I	0b01010	rd	rs	imm.
	5	4	3	3

RTL

newPC = PC + 2
IR = M[PC]
A = Reg[IR[10-7]]
B = Reg[IR[6-4]]
R = A + SE(IR[3-0])
Mem[R] = B

2.4.23 Subtract (sub)

Description

Subtracts value of a register from value of an other register and stores in a register.

Syntax

```
sub   $rd, $rs, $rt
```

Format

R	0b00010	rd	rs	rt
	5	4	4	3

RTL

newPC = PC + 2
PC = newPC
IR = Mem[PC]
A = Reg[IR[6-3]]
B = Reg[IR[2-0]]
result = A - B
Reg[IR[10-7]] = result

Chapter 3

Sample Code

3.1 Common Code Constructs

3.1.1 If Statement

C Code

```
if (a == b) {
    a += 1;

}
else {
    a -= 1
}
```

Equivalent MERC-16 Assembly

Address	Label	Assembly	Machine Code
0x0000		bne $t0, $t1, ELSE	0x0905
0x0002		addi $t0, $0, 1	0x2908
0x0004		j DONE	0x092F
	else:		
0x0006		addi $t0, $t0, -1	0x1802
	done:		

3.1.2 While Statement

C Code

```
int i = 5;
while(i != 0) {
    i--;
}
```

Equivalent MERC-16 Assembly

Address	Label	Assembly	Machine Code
0x0000		addi $t0, $0, 5	0x0905
	while:		
0x0002		beq $t0, $0, DONE	0x2908
0x0004		addi $t0, $t0, -1	0x092F
0x0006		j WHILE	0x1802
	done:		

3.1.3 For Statement

C Code

```
int length = 5;
for (int i = 0) {
}
```

Equivalent MERC-16 Assembly

Address	Label	Assembly	Machine Code
0x0000		addi $t1, $0, 5	0x0905
0x0002		add $t0, $0, $0	0x2908
	for:		
0x0004		beq $t0, $t1, DONE	0x092F
0x0006		addi $t0, $t0, 1	0x092F
0x0008		j FOR	0x1804
	done:		

3.1.4 Euclid's Alogrithm

C Code

```c
// Find m that is relatively prime to n.
int relPrime(int n) {
    int m;
    m = 2;

    // n is the input from the outside world
    while (gcd(n, m) != 1) {
        m = m + 1;
    }

    return m;
}



// The following method determines the Greatest
// Common Divisor of a and b using Euclid's
// algorithm.
int gcd(int a, int b){
    if (a == 0) {
        return b;
    }

    while (b != 0) {
        if (a > b) {
            a = a - b;
        }
        else {
            b = b - a;
        }
    }

    return a;
}
```

Equivalent MERC-16 Assembly

Address	Label	Assembly	Machine Code	Comment
	relprime:			
0x0000		addi $t0, $0, 2	0x0902	#t0 is m
	while1:			
0x0002		add $arg1, $t0, $0	0x0610	# Set m as argument 0
0x0004		addi $sp, $sp, -4	0x089C	# Allocate space
				# on the stack
0x0006		store $sp, $t0, 0	0x48A0	# Store m on the stack
0x0008		store $sp, $ra, 2	0x48D2	# Store return address
0x000A		call gcd	0x200F	# Call gcd function
0x000C		addi $t1, $0, 1	0x0981	# Branch to return if
				# return value is 1
0x000E		beq $rv0, $t1, 6	0x2CB6	# Branch to return1 if
				# return value is 1
0x0010		load $t0, $sp, 0	0x3910	# Loading m into t0
0x0012		add $t0, $t0, $t1	0x0113	# Increment m by 1
0x0014		j while1	0x1801	# Jump to while1
0x0016		add $rv0, $t0, $0	0x0490	# Store result in return
				# value register
	return1:			
0x0018		load $ra, $sp, 2	0x3A92	# Restore the original
				# return address
0x001A		addi, $sp, $sp, 4	0x0894	# Restore the stack
0x001C		jr $ra	0xA805	# Return
	gcd:			
0x001E		bne $arg0, $0, 2	0x3582	# Set return value to b
0x0020		add $rv0, $arg1, $0	0x04D8	# Set return value to b
0x0022		jr $ra	0xA805	# Return
	while2:			
0x0024		add $t1, $arg0, $0	0x01D8	# Puts a into register t1
0x0026		add $t2, $arg1, $0	0x0260	# Puts b into register t2
0x0028		beq $t2, $0, 5	0x2805	# Branch to return2 if b = 0
0x002A		blet $t1, $t2, 2	0x99C2	# Branch to else if a <= 0
0x002C		sub $t1, $t1, $t2	0x119C	# Set a to a - b
0x002E		j while 2	0x1812	# Jump to while2
0x0030		sub $t2, $t2, $t1	0x1223	# Set b to b - a
0x0032		j while 2	0x1812	# Jump to while2
	return2:			
0x0034		add $rv0, $t1, $0	0x0498	# Set return value to a
0x0036		jr $ra	0xA805	# Return

Chapter 4

Verification

4.1 RTL

We put initial values into the RTL and preformed the operations, verifying that our results are what we expect. Included is the an example of this verification

4.1.1 addi

```
addi $t0, $0, 5
```

PC = 0x0008 newPC = PC + 2 = 0x000A
IR = Mem[PC] = Mem[0x000A] = 0x0905 // Machine code of given instruction
A = Reg[IR[6-4]] = Reg[0] = \$0 = 0
B = SE(IR[3-0]) = SE(0101) = 5
Result = A + B = 5
Reg[IR[10-7]] result = 5

```
$t0 = 5
```

4.1.2 lli

```
lli $s0, 16
```

PC = 0x0030 newPC = PC + 2 = 0x0032
IR = Mem[PC] = Mem[0x0032] = 0x4610 // Machine code of given instruction
D = R[10:8] = s0
s0 = 16 `$t0 = 5`

4.2 Components

Component	Testing Plan
ALU	In order to comprehensively test this component, well need to have a thorough process in all the control signals. In order to properly do this, we first have the Op control signal to start at zero (000) and the SrcA and ScrB control signals also at zero (0, 0 - respectfully). From there we increment one of the SrcA/SrcB control signals like and repeat the test.
Left-Shift Module	We decided to test these components using an exhaustive testing method. What is meant by this is that we insert any value into these components and see if they properly sign extended and/or left shifted said value, and we continuously use different values until it is seen that all of the values were sign extended and/or left shifted
Sign Extender	We decided to test these components using an exhaustive testing method. What is meant by this is that we insert any value into these components and see if they properly sign extended and/or left shifted said value, and we continuously use different values until it is seen that all of the values were sign extended and/or left shifted

4.3 Subsystems

Subsystem	Testing Plan
Register File	In order to properly test the register file component, we will need to thoroughly process through two control signals. In this case, we will need to start with the control signal Dest at 0 and then the control signal Write at 0. Then we will need to alternate the bit for Write, with each cycle incrementing the bit for Dest.
Memory	In order to properly test the Memory component, we will need to thoroughly process through four control signals. In this case, we will need to start the control signals Read, Write, ToReg, and WR_Data at 0 respectively. Then we will need to alternate the bit for WR_Data; with each transition from 1 to 0 in WR_Data, ToReg must alternate; with each transition from 1 to 0 in ToReg, Write must alternate; with each transition from 1 to 0 in Write, Read must alternate.
PC Subsystem	In order to properly test this subsystem, well need a 16-bit input that goes into PC_J (jumping), PC_Inc (incrementation), and PC_Br (branching). We do not need three separate 16-bit inputs since all of these inputs should only deal with the same 16-bit input. The result should also come out as 16-bit as well. As for the control signals, there are ten 1-bit inputs that are for checking to see if we need to write something onto the PC, branch out based on comparing values, and seeing if the value is less than/greater than/equal to. In order to test this, we are going to need 210 combinations on each of the 1-bit inputs.

Decode Subsystem	We test this subsystem by having two 16-bit inputs that are used for the Instruction Register and Register Data. These could either be the same values or different ones depending on what is needed for testing. We also have 2 1-bit control bits that allow for the subsystem to enable writing in the register file and if were using the long (4-bit) or short (3-bit) rs register when going through the subsystem. What comes out are four 16-bit outputs, two of which are directly from the register file component (A & B) and two that are sign extended. One of the two 16-bit sign extended values is also left shifted by 1. We can put in any kind of random value onto the two 16-bit inputs as long as theyre the same size, meanwhile we can have a total amount of 3 combinations for WR_EN and Short_Rs when using this subsystem.

4.4 Subsystem Connection

Stage	Testing Plan
PC Memory	Connect the PC subsystem to the Memory subsystem. PC Output will connect to ADDR and Data Inputs. Test and verify that the connection works via the PC output comes out as 16-bits and reaches the Memory subsystem inputs with the same 16-bit size. Be sure that the control signals are connected to their respective subsystems
PC Memory Decode	Connect the first setup with the decode subsystem by having the decoder follow the Memory subsystem. MemOut will connect to the IR and WR_Data. Test and verify that the connection works by checking if the MemOut is 16-bit and goes to the inputs at the same size without any manipulation. Be sure that the control signals are connected to their respective subsystems.
PC Memory Decode ALU Input	Connect the second setup with the ALU subsystem by having the ALU follow the decode subsystem. Connect the outputs to two multiplexers: first multiplexer is connected to the A output and PC; second multiplexer is connected to the B output, SE1, and SEIL1 outputs. The first multiplexer output will go to the A input of the ALU while the second multiplexer output will go to the B input. Test and verify that the connection works by checking if the output values in the Decode stay at 16-bit when entering the ALU. Be sure that the control signals are connected to their respective subsystems and multiplexers.
PC Memory Decode ALU	Connect the outputs of the ALU into the inputs of PC, Memory, and Decode subsystems. ALUOut goes into these inputs via multiplexers (specified in the datapath). Test and verify that the entire system works with the appropriate size values. Be sure that the control signals are connected to their respective subsystems and multiplexers.

4.5 Control

When the Control Unit obtains the opcode, be sure that it is 5-bit long. From there, well start the process by initializing the all of the control signals, then reading the Opcode. When reaching this step, be sure to convert the opcode to their respective instruction and based on that instruction, make sure that the values needed for changing correspond to the instruction. If the opcode contains more than one instruction, check ahead of the next instruction so that it splits up and goes to ultimately the right instruction. Follow through on each instruction until it hits the end of the process and loops back into initializing the control signals all over again. Do this for all instructions available in the instruction set.

4.6 Complete System

A simple test that we could do would be to start running individual instructions through a datapath. If this is successful, small programs can be tested (which will exponentially be followed by larger programs). In other words, we should first test each instruction to make sure their functions create the expected results. If the results of a specific instruction does not match the expected result, there are two options to consider when moving forward: change what is faulty in the system, or recalculate the results by hand (perhaps the machine may be right and the human is wrong). If each instruction appears to function they way that they are intended, we must then combine these instructions to create a simple program (e.g. a counter using a for- or while-loop). If the program gives the results as expected, we can move on to other tests. If not, we will have to debug the system and see where results may have changed and then go back and review the instruction which caused the unexpected catalyst. Once these small tests have passed, we must then move on to larger programs such as finding the relative prime of a number, finding the factorial of a number, or finding the Fibonacci number of a given number.