Team Yellow 2122a-01

Accumulator Processor

Zeen Wang, Jermaine Brown,

Helen Wang, Athena Henderson

Table Of Contents

Our Processor

Our processor is an accumulator architecture that only has two registers: the accumulator register and the stack pointer register. We load data into the accumulator to do calculations and compare it with data in the memory. The stack pointer register is used to store the address of the current top of the stack. Since we are making changes directly to the data stored in memory when running a function, any data that we need that might be changed will be stored on the stack.

Detailed Overview

We used 16-bit addresses and 16-bit instructions. The first five bits of our instruction are the opcode, then eight bits after that are the address that will be left-shifted by one and then sign-extended to get the 16-bit address the data is stored at. The last three bits are unused by all except the beq and bne instructions, where it is used as the immediate that indicates how many instructions we wish to jump. Since sign extend is used to calculate the address of both data and instruction, we set a rule that the data's address must start with a one and the instruction's address must start with a zero. We used a multicycle approach to implement our instructions. We precalculate the branch address in our decode stage. All of our instructions use 3 cycles except lw and sw.

Testing

To test our datapath, we first tested each component before dividing our datapath into 4 subsystems: PC, memory and control, wires, and the ALU. Each subsystem consists of the main component and the source mux before it except for the wires subsystem which consists of the

accumulator register, the stack pointer register, and the muxes before them. After that, we connected the 4 subsystems and ran our test code which tests each instruction. Then we ran the relPrime code.

Advantages/Disadvantages

Though our approach to using sign extend for addresses limits the amount of memory our program can use, it makes our instruction run a cycle faster since we do not need the ALU to calculate the address. Our design is cheap, since only two registers are used, and relatively fast given our addressing mode. However, the downside to our design is that due to our addressing mode, which uses sign extend, means the number of addresses we can use is limited compared to other possible ways to get a 16-bit address from 8-bit input. This also gets all our data stuck at the top of the memory and instructions at the bottom.

Unique/Extra Features

Memory I/O

Our design uses memory for input and output. We have a magic number set for our I/O, 0x03FC. When the load instruction is given this number as the address to read from, instead of reading the data at that address and putting it into the accumulator register, the memory will read from the input wire and pass that value to the accumulator register. A similar approach is used in the save instruction, where if the address is 0x03FC, the value will be passed to the output of the datapath instead of saving the value in the accumulator register into memory.

Assembler

We have a well-performed assembler, implemented by Python. The user interface is based on the command line, very much similar to the gcc and it can handle the request from other

programs (hypothetically compilers). We used the hashmap to translate the opcodes, and all the other assembler commands, so the speed of the assembler is pretty fast. Also, we implemented the Lable(tag) system for jump, so that we free the programmer from any worries about the addressing. Other than all that's been said, this is a very naive and simple demo of the real-world assembler, so we only support certain formats of the assembler codes (say all numbers need to be decimals, no binary no hex.)

"Real" Accumulator

While we do have two registers total, the accumulator and the stack register. Only the accumulator register is accessible by the programmer. This makes writing code for our processor easy since the programmer will only need to keep in mind the accumulator register. Plus, nearly anything needing to be done can be done using just the accumulator register.

Conclusion

Designing a processor from start to finish would've been immensely more difficult without the examples we had seen in class to build off of and learn from. It was important for us to keep it simple, and it was surprising to see how simple we could really keep an accumulator processor. And even more so, how simple an accumulator processor has to be since you only have one register to rely on. Overall, we learned a lot about design over these past few weeks along with the importance of choosing addressing modes and having a clearly defined direction at the beginning all could've made the design process a bit easier for our group.

The instruction for relPrime is 120 bytes. Running relPrime with 0x13B0 gives output 11 and a total of 12096 instructions are executed, which is a total of 36770 cycles. Out cycle time is 10.9 ns which is 91.54 MHz, so it takes approximately 4 ms to run the whole program. The total logic gates are 410/6272 (7%), the total registers are 136, and the total memory bites are 8192/276480 (3%).

Appendix

Appendix A: Design Document

Design Document

Zeen Wang, Jermaine Brown,
Helen Wang, Athena Henderson
Team: Yellow-2122a-01

Rose-Hulman Institute of Technology
CSSE 232 Computer Architecture I
Prof. Williamson
April 3, 2022

Link to the Google Doc DesignDocument

Description:

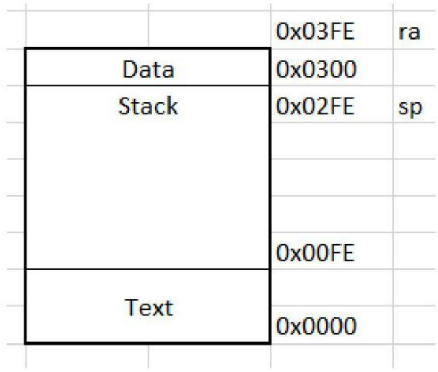
Our design uses a single register accumulator to store data and compare it to inputs. At all times we only use one register and use an allocated space in Memory for data. For our addresses we will use sign extensions to target specific places in memory to receive either data for destination. The input must have the correct first bit to target the proper place in memory.

We are going to use 2 registers, the accumulator($acc) and stack pointer($sp). The accumulator is the only register available by the programmer.

We left shift by 1 bit then we sign extent (the most significant bit will be 1 if it is a data and 0 if it is a instruction)

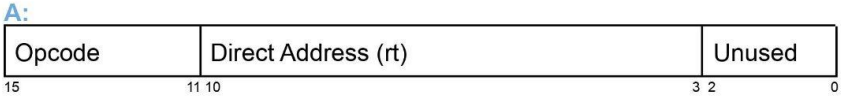
Memory Map:

New:

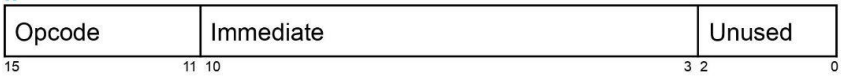


Types:

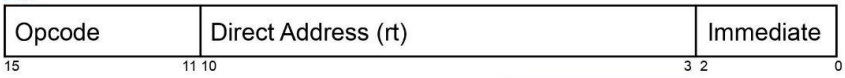
A:



I:

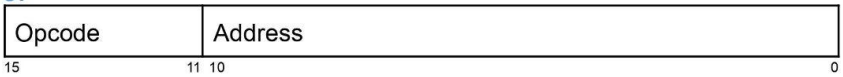


AI:



Link to the Google Doc DesignDocument

J:



Instructions:

Name	Type	Operation	Opcode
load a	A	acc = Mem[getAddr(rt)]	00001
	Takes an 8 bit address a and loads the value at memory address a to the accumulator, using the address rule. If rt = 0xFD = 11111101 (corresponding address 0xFFFC), then the memory is going to read from input		
save a	A	Mem[getAddr(rt)] = acc	00010
	Take an 8 bit address a and save the value in the accumulator into the memory with address a, using the address rule. If rt = 0xFD, then the memory is going to sent to output		
loaduiimm	I	acc = {imm, 8b'0}	00011
	Takes an 8 bit immediate and load it to the upper 8 bits of the accumulator		
bne a, imm	AI	if(acc != Mem[getAddr(rt)]) PC = PC + 2 + getAddr(imm)	00100
	Takes an 8 bit address and a 3 bit immediate. If the value stored at address a is not equal to the value of the accumulator, then jump to the address calculated from the immediate using the branch address rule.		
beq a, imm	AI	if(acc == Mem[getAddr(rt)]) PC = PC + 2 + getAddr(imm)	00101
	Takes an 8 bit address and a 3 bit immediate. If the value stored at address a is equal to the value of the accumulator, then jump to the address calculated from the immediate using the branch address rule.		
slt a	A	acc = acc < Mem[getAddr(rt)] ? 1:0	00110
	Compare the value in the accumulator with the value stored at address a, if the accumulator is less than a then we set the accumulator to 1, else we set the accumulator to 0.		

Link to the Google Doc DesignDocument

slti **imm**	I	acc = acc < SignExtent(imm) ? 1:0	00111
	Compare the value in the accumulator with the immediate, if the accumulator is less than the immediate then we set the accumulator to 1, else we set the accumulator to 0.		
j **a**	J	PC = jumpAddr(rt)	01000
	Jump to the instruction with address a, calculated using the address rule.		
jal **a**	J	Mem[ra] = PC + 2 PC = jumpAddr(rt)	01001
	Jump to the instruction with address a, calculated using the address rule. Store the current PC + 2 to a fixed memory location.		
sw **imm**	I	Mem[sp + SignExtent(imm)] = acc	01010
	Stored the value in the accumulator onto the stack where it is offset imm to the stack pointer.		
lw **imm**	I	acc = sp + SignExtent(imm)	01011
	Stored the value from the stack where it is off offset imm to the stack pointer to the accumulator.		
ms **imm**	I	sp = sp + SignExtent(imm)	01100
	Move the stack pointer with the sign extended immediate.		
sub **a**	A	acc = acc - Mem[getAddr(rt)]	01101
	Subtract the value stored at address a from the accumulator and store the result in the accumulator		
add **a**	A	acc = acc + Mem[getAddr(rt)]	01110
	Add the value stored at address a to the accumulator and store the result in the accumulator		
addi **imm**	I	acc = acc + SignExtent(imm)	01111
	Add the sign extended immediate to the accumulator and store the result in the accumulator		
and **a**	A	acc = acc & Mem[getAddr(rt)]	10000
	And the value stored at address a to the accumulator and store the result in		

Link to the Google Doc DesignDocument

	the accumulator		
or a	A	acc = acc | Mem[getAddr(rt)]	10001
	Or the value stored at address a to the accumulator and store the result in the accumulator		
ori imm	I	acc = acc | ZeroExtent(imm)	10010
	Or the zero extended immediate to the accumulator and store the result in the accumulator		
loadi imm	I	acc = SignExtent(imm)	10011
	Load the sign extended immediate to the accumulator.		
jv ra	I	pc= Mem[ra]	10100
	Jump to the addressed stored at ra		

Address rule: We left shift by 1 bit then we sign extent (the most significant bit will be 1 if it is a data and 0 if it is a instruction)
Branch address: Left shift the immediate by 1, sign extend it to 16 bits then add it to the value of the current PC plus 2.

Call procedure:

For the callers, they are responsible to store the $acc register value, and put the return address on the $acc. For callees, they are responsible to restore the value in the Data memory, and callees will move the stack to store the original value of the data in the stack memory, and restore them back before return. Also, it's callee's responsibility to store the return address in the stack memory and use them for return.

Example program(s):

High Level Code	Assembly	Machine Code	Addresses
n = 6; c = relPrime(n);	load io save n jal relPrime load m save io save c	00001 11111110 000 00010 00000000 000 01001 00000000 110 00001 00000001 000 00010 11111110 000 00010 00000010 000	0x0000 0x0002 0x0004 0x0006 0x0008 0x000A
int **relPrime**(int n) {	relPrime: loadi 2	10011 00000010 000	0x000C

Link to the Google Doc DesignDocument

int m;
m = 2;
while (gcd(n, m) != 1) {
m = m + 1;
}
return m;
}
	save m
	ms -6
	loop2: load m
	sw 0
	save a
	load n
	sw 2
	save b
	load ra
	sw 4
	jal gcd
	save io
	save o
	lw 0
	save m
	lw 2
	save n
	lw 4
	save ra
	loadi 1
	bne o tif2
	j end2
	tif2: load m
	addi 1
	save m
	j loop2
	end2: ms 6
	jv ra
		00010
		01100
		00001
		01010
		00010
		00001
		01010
		00010
		00001
		01010
		01001
		00010
		00010
		01011
		00010
		01011
		00010
		01011
		00010
		10011
		00100
		01000
		00001
		01110
		00010
		01000
		01100
		10100
			00000001
			11111010
			00000001
			00000000
			00000011
			00000000
			00000010
			00000100
			11111111
			00000100
			00000100
			11111110
			00000101
			00000000
			00000001
			00000010
			00000000
			00000100
			11111111
			00000001
			00000101
			00000100
			00000001
			00000001
			00000001
			00000001
			00000110
			11111111
				000
				000
				000
				000
				000
				000
				000
				000
				000
				000
				011
				000
				000
				000
				000
				000
				000
				000
				000
				000
				001
				001
				000
				000
				000
				001
				000
				000
					0x000E
					0x0010
					0x0012
					0x0014
					0x0016
					0x0018
					0x001A
					0x001C
					0x001E
					0x0020
					0x0022
					0x0024
					0x0026
					0x0028
					0x002A
					0x002C
					0x002E
					0x0030
					0x0032
					0x0034
					0x0036
					0x0038
					0x003A
					0x003C
					0x003E
					0x0040
					0x0042
					0x0044

Link to the Google Doc DesignDocument

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;
}
	gcd: loadi 0
	  bne a loop
	  load b
	  jv ra
	loop: loadi 0
	  bne b go
	  j end
	go: load b
	  slt a
	  save i
	  loadi 1
	  beq i tif
	  j else
	tif: load a
	  sub b
	  save a
	  j loop
	else: load b
	  sub a
	  save b
	  j loop
	end: load a
	  jv ra
		10011 00000000 000
		00100 00000011 010
		00001 00000100 000
		10100 11111111 000
		10011 00000000 000
		00100 00000100 001
		01000 00000111 000
		00001 00000100 000
		00110 00000011 000
		00010 00000110 000
		10011 00000001 000
		00101 00000110 001
		01000 00000110 100
		0001 00000011 000
		01101 00000100 000
		00010 00000011 000
		01000 00000100 111
		00001 00000100 000
		01101 00000011 000
		00010 00000100 000
		01000 00000100 111
		00001 00000011 000
		10100 11111111 000
			0x0046
			0x0048
			0x004A
			0x004C
			0x004E
			0x0050
			0x0052
			0x0054
			0x0056
			0x0058
			0x005A
			0x005C
			0x005E
			0x0060
			0x0062
			0x0064
			0x0066
			0x0068
			0x006A
			0x006C
			0x006E
			0x0070
			0x0072

if (n == 0) {
  n++;
} else {
  n = 2;
}
	loadi 0
	  bne n, else
	  load n
	  add 1
	  save n
	  j done
	else: loadi 2
	  save n
	done:
		10011 00000000 000
		00100 10000100 100
		00001 10000100 000
		01110 00000001 000
		00010 10000100 000
		01000 00001001 000
		10011 00000010 000
		00010 10000100 000
			0x 0002
			0x 0004
			0x 0006
			0x 0008
			0x 000A
			0x 000C
			0x 000E
			0x 0010
			0x 0012

while (n != 0) {
  n = n - m
}
	loop: loadi 0
	  beq n, done
	  load n
	  sub m
	  save n
	  j loop
	done:
		10011 00000000 000
		00101 10000100 100
		00001 10000100 000
		01101 10000011 000
		00010 10000100 000
		01000 00000001 000
			0x 0002
			0x 0004
			0x 0006
			0x 0008
			0x 000A
			0x 000C
			0x 000E

int count = 0;
for (int i = 0; i < n; i++) {
  count++;
	loadi 0
	  save count
	  save i
		10011 00000000 000
		00010 10000110 000
		00010 10000010 000
			0x 0002
			0x 0004
			0x 0006

Link to the Google Doc DesignDocument

}	loop: beq n, done add 1 save count save i j loop done:	00101 10000100 100 01110 00000001 000 00010 10000110 000 00010 10000010 000 01000 00000100 000	0x 0008 0x 000A 0x 000C 0x 000E 0x 0010 0x 0012
Data: 0xFF00 a(value = m) 0xFF02 b(value = n) 0xFF04 i 0xFF06 m 0xFF08 n 0xFF0A o 0xFF0C count 0xFF0E c		**Stack:** 0xFEFE	

<u>RTL:</u>

<u>**Name**</u>	<u>**Fetch**</u>	<u>**Decode**</u>		
load a	IR = Mem[PC]; PC = PC + 2	MDR = Mem[SE(IR[10-3]<<1)] ALUOut = PC + SE(IR[2-0]<<1)	Acc = MDR	
save a			Mem[SE(IR[10-3]<<1)] = Acc	
loadui imm			Acc = IR[10-3] << 8	
bne a, imm			if(Acc != MDR) PC = ALUOut	
beq a, imm			if(Acc == MDR) PC = ALUOut	
slt a			if(Acc < MDR) Acc = 1 else Acc = 0	
slti imm			if(Acc < SE(IR[10-3])) Acc = 1 else Acc = 0	
j a			PC = ZE(IR[10-0]<<1)	

jal a			Mem[ra] = PC PC = ZE(IR[10-0]<<1)	
sw imm			ALUOut = sp +SE(IR[10-3])	Mem[ALUOut] = Acc
lw imm			ALUOut = sp +SE(IR[10-3])	Acc = Mem[ALUOut]
ms imm			Sp = Sp + (SE(IR[10-3]))	
sub a			Acc = Acc - MDR	
add a			Acc = Acc + MDR	
addi imm			Acc = Acc + (SE(IR[10-3]))	
and a			Acc = Acc & MDR	
or a			Acc = Acc | MDR	
ori imm			Acc = Acc | (ZE(IR[10-3]))	
loadi imm			Acc = SE(IR[10-3])	
jv ra			PC = Mem[ra]	

RTL Components:

1. 16-bit Registers: reg16

Input wires:

In: 16-bit bus for Input data,
E: 1-bit Enable wire,
CLK: 1-bit clock wire
Reset:

Output wire:

Out: 16-bit bus for output data

Implemented Symbols: Acc, Sp, PC, IR, ALUOut, MDR

Description: We can write to the register only when E = 1. It will read and write on rising edge

Test implementation: Set reset to 1 for 10 cycles then switch it back to one. Test if the output is 0. Set input to 1 while keeping the enable as 0, the output should be 1. Change enable to 1, now the output should be 1. Change the input to 2 between the rising edge and falling edge, the next falling edge should still read 1, while the next rising edge should read 2.

2. **Memory: mem16**

Input wires:

Addr: 16-bit bus for Memory Address,
In: 16-bit bus for Input data,
W:1-bit Memory write control wire,
CLK: 1-bit clock wire

Output wire:

Out: 16-bit bus for output data

Implemented Symbols: Mem

Description: It will write on rising edge

Test implementation: Give a 16-bit address and set read and write to 0, should return 0. Set read to 1, the output should be the data at the given address. Set input data to a random 16 bit number then set read to 0, write to 1, set read to 1, the output should be the new data.

3. **ALU: alu16**

Input wires:

A: 16-bit bus for Input A,
B: 16-bit bus for Input B,
OP: 3-bit bus for ALUopcode,

Output wire:

Out: 16-bit bus for output data
Zero: 1-bit zero wire (high for 0)

Implemented Symbols: ALU

Opcode:

000 for And
001 for Or
010 for Add
011 for Sub
100 for Slt

Test implementation: Input 2 16-bit numbers and change the opcode every cycle, the output should be the result of and, or, add, sub, and slt. Give 0x1111 and 0x1111l and opcode 11, the zero output should be 1. Give 0xFFFF and 0x0001 that will cause an overflow and opcode 00, the overflow output should be 1.

4. **1-bit Mux: mux1b16**

Input wires:

A: 16-bit bus for Input A,
B: 16-bit bus for Input B,
OP:1-bit bus for opcode

Output wire:

Out: 16-bit bus for output data

Implemented Symbols: 1bMux

Opcode:

0 for A
1 for B

Test implementation: Give two different 16-bit numbers, change the opcode every cycle and check the output

5. **1-bit Mux: mux1b16**

Input wires:

A: 16-bit bus for Input A,
B: 16-bit bus for Input B,
C: 16-bit bus for Input C,
D: 16-bit bus for Input D,
OP:2-bit bus for opcode

Output wire:

Out: 16-bit bus for output data

Implemented Symbols: 1bMux

Opcode:

0 for A
1 for B
10 for C

11 for D

Test implementation: Give four different 16-bit numbers, change the opcode every cycle and check the output

6. **16-bit Sign Extend: se16**

Input wires:

In: 8-bit bus for Input,

Output wire:

Out: 16-bit bus for output data

Implemented Symbols:

Test implementation: Give a 8-bit number that starts with 1, should fill the upper bits with 1. Give an 8-bit number that starts with 0, should fill the upper bits with 0.

7. **16-bit Zero Extend: ze16**

Input wires:

In: 8-bit bus for Input,

Output wire:

Out: 16-bit bus for output data

Implemented Symbols:

Test implementation: Give different 8-bit numbers, some starting with 1 and some starting with 0. The zero-extender should fill the upper bits with 0.

8. **1-bit Mux mux1b1**

Input wires:

A: 1-bit for Input A,

B: 1-bit for Input B,

OP:1-bit for opcode

Output wire:

Out: 1-bit for output data

Implemented Symbols:

Opcode:

0 for A

1 for B

Test implementation: Give different 1-bit numbers, set op to 1, then set op to 0

Integration Plan:

1. PC:

The first one will be small, including a 2-bit mux and PC ensuring that the mux is properly selecting the correct inputs based on the PCSrc and that the PC is changing when it should. When doing beq we will set Branch to 1 and bne/beq to 1. When doing bne we will set Branch to 1 and bne/beq to 0. This and PCWrite will select when PC will be changed. PC will also handle shifting the ZE IR input it receives.

Input Wires:

2-bit PCSrc wire

1-bit Zero? Input

1-bit Branch? wire

1 bit bne/beq wire

16-bit bus ALU

16-bit bus ALU Out

bus IR ZE

1-bit PCWrite wire

1-bit clock wire

Output Wires:

16-bit bus for PC out

bus IR ZE << 1

Tests:

Give 3 separate values for the inputs and for each value of PCSrc the clock will do 2 cycles to test when PCWrite is off or on. When PCWrite is on the output should be the selected input from the mux and if PCWrite is off the output should be the previous cycle's output. Next, set PCWrite to 0, Branch to 1 and bne/beq to 1 and check the output. Then switch bne/beq = 0 and check the output.

2. Memory:

The Second will be MEM, a 1-bit mux, 2-bit mux, IR, and MDR. Once again checking the mux functionality. However, this subsystem will ensure that the control of memwrite and memread are working properly with Mux outputs and that the memory block of our Datapath is working.

Link to the Google Doc *DesignDocument*

Input Wires:

16-bit bus IO Input

bus IR SE << 1

16-bit bus ACC

16-bit bus ALU Out

16-bit bus PC

1-bit reset wire

1-bit clock wire

Output Wires:

bus for IR out

16-bit bus for MDR out

16-bit bus for Memory out

1-bit PCWrite wire

1-bit Branch wire

1-bit bneOrbeq wire

1-bit PCSrc wire

1-bit AccWrite wire

1-bit SpWrite wire

1-bitALUSrcA wire

1-bit ALUSrcB wire

3-bit bus ALUOp

16-bit bus for IO Output

Tests:

We will once again create 4 unique values for the bus inputs to use for this test. MemAddr will increment every 8 cycles while MemData increments every 4 clock

Link to the Google Doc DesignDocument

cycles to test MemRead and MemWrite together for all possible combinations of the selected values.

3. **Wires:**

The Third subsystem will take 3 inputs and test output for the SE <<1, SE, ZE, <<8, the ACCSrc 3-bit mux, ACC, and SP. This subsystem ensures a functional modification of data via bit shifts and Zero and Sign extensions, and that this modified data is properly selected and stored in the ACC and SP when it should be.

Input Wires:

3-bit ACCSrc wire

1-bit ACCWrite wire

1-bit SPWrite wire

bus IR

bus MDR

bus Memory Out

1-bit clock wire

Output Wires:

16-bit bus for ACC out

16-bit bus for SP out

bus for IR SE <<1 out

bus for IR SE out

bus for IR ZE output data

Tests:

The main focus of testing the subsystem is not the ZE, SE, or bit shift since they are tested individually. The tests will increment through selections for ACCSrc every 2 cycles to alternate ACCWrite and confirm the proper outputs. At the same time SPWrite will alternate for a secondary check.

4. **ALUPath:**

The Fourth subsystem will include the ALUSrcA, ALUSrcB, ALU, and ALU out. This subsystem guarantees the ALU operations being performed correctly, and the output is stored with no issue.

Link to the Google Doc DesignDocument

Input Wires:

2-bit ALUSrcA wire

3-bit ALUSrcB wire

16-bit bus PC

16-bit bus ACC

16-bit bus SP

16-bit bus MDR

bus IR SE

bus IR ZE

bus IR SE << 1

3-bit bus for ALUopcode

Output Wires:

16-bit bus for ALU out

16-bit bus for ALU Out output

1-bit Zero? output data

Tests:

2 of the 6 inputs are going to be the same to test Zero?. But the 6 inputs will remain the same through the test. We'll increment ALUSrcA every time we try every op code with every selection in ALUSrcB to assure that the ALUOut and ALU are working with proper selection. When we select subtract in the opcode and the 2 inputs that are valued the same we will check that Zero gives a 1 as an output.

5. Overall Integration

Finally you test the four subsystems left as one whole system by connecting the Outputs to the inputs in order that the subsystems are listed like output of PC to inputs of Memory as shown in the Datapath. Lastly connect the last output to the inputs of the PC subsystem. With tests written for the subsystems the tests for the integration are extremely similar.

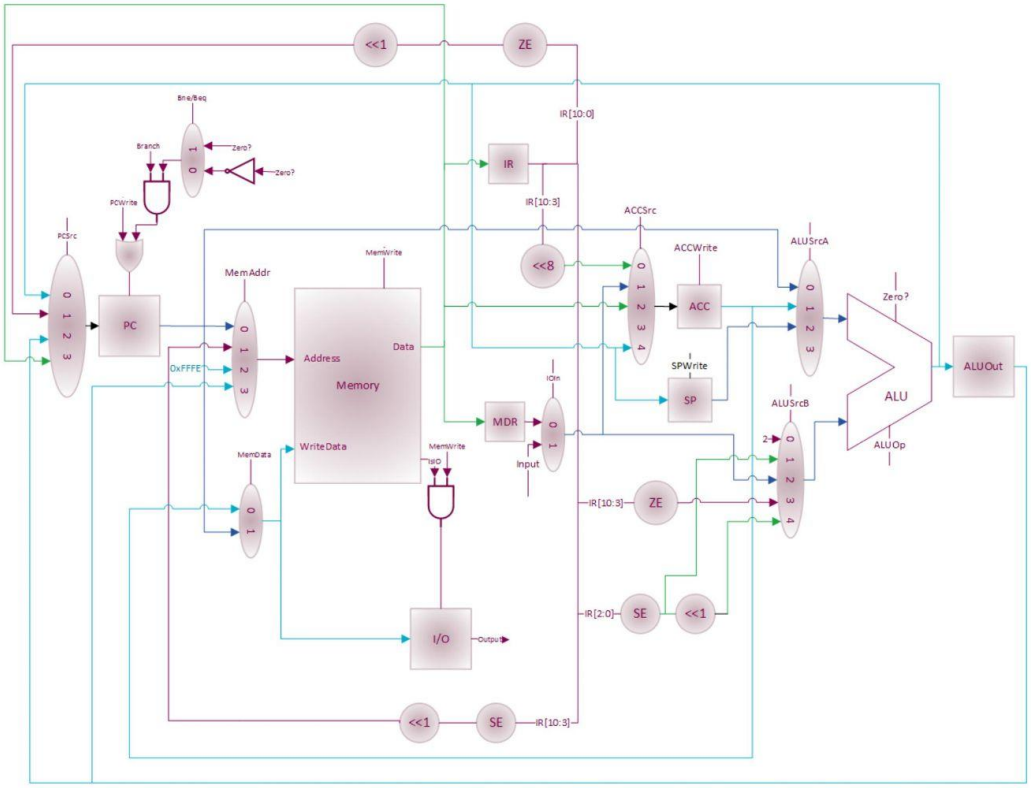
Link to the Google Doc *DesignDocument*

Input/Output/Control:

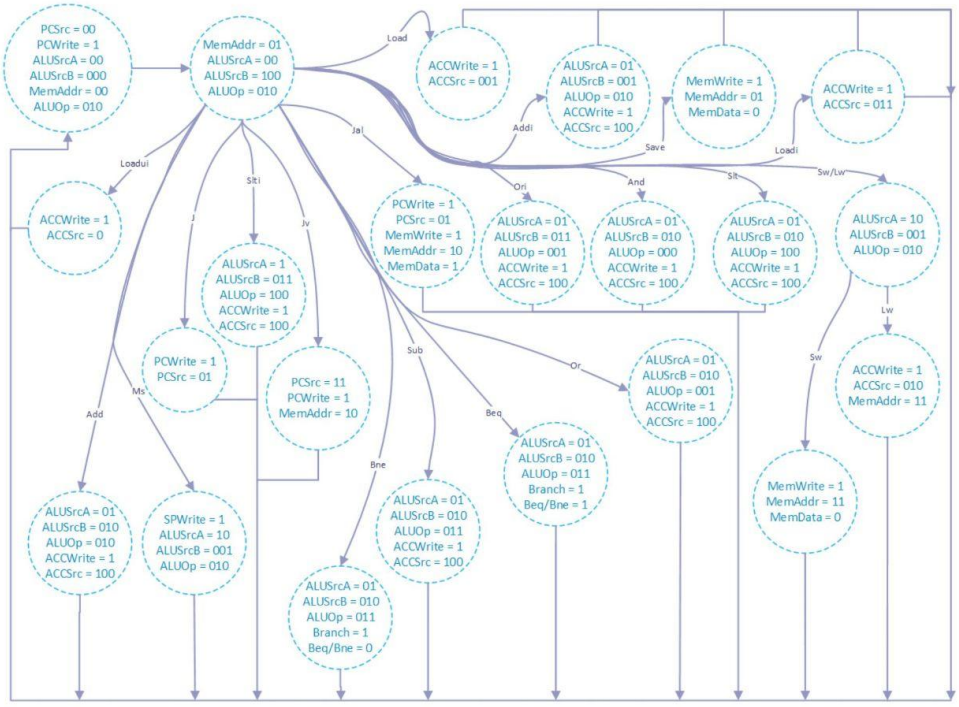
Name	Bit Size	I/O/C	Description
PCSrc	2	Control	Used to determine between ALU output, the zero-extended immediate shifted by 1, and ALUOut for the input to PC.
MemAddr	2	Control	Used to determine between PC, the sign-extended immediate shifted by 1, 0xFFFE, and ALUOut for the address we are writing to in memory
MemData	1	Control	Used to determine between Acc output and PC for input to memory
MemWrite	1	Control	Is 1 when we are writing to memory, 0 otherwise
SPWrite	1	Control	Is 1 when we are writing to Sp, 0 otherwise
ACCSrc	3	Control	Used to determine between the immediate shifted by 8, MDR output, data from memory, the sign-extended immediate, or Sp output for the input to the accumulator
ACCWrite	1	Control	Is 1 when we are writing to the accumulator, 0 otherwise
ALUSrcA	2	Control	Used to determine between PC, Acc output, and Sp output for the input to A
ALUSrcB	3	Control	Used to determine between 2, the sign-extended immediate, MDR output, the zero-extended immediate, and the sign-extended immediate shifted by 1 for the input to for B
Zero?	1	Control	Flag that is set to 1 when the output of the ALU is zero
ALUOp	3	Control	Used to determine which operations should be done by the ALU [000 - And,0 01 - Or, 010 - Add, 011 - Sub, 100 - Slt]
Bne/Beq	1	Control	Used to determine what control bit gets passed into the PC to determine exactly when to write to the PC (specifically for determining bne/beq) [1 - Beq, 0 - Bne]
PCWrite	1	Control	Used to determine what control bit gets passed into the PC to determine exactly when to write
Branch	1	Control	Used to determine what control bit gets passed into the PC to determine exactly when to write (specifically if it's a branch instruction)
IRWrite	1	Control	Used to determine whether we can write to IR or not

Link to the Google Doc DesignDocument

Datapath:



Link to the Google Doc DesignDocument



All read/write enable bits is 0 if not mentioned

Datapath test implementation:

constant io = 1111 1101
constant a = 1000 0000 //a = 20
constant b = 1000 0001 //b = 10

//Test reset

//Test I/O
load io
save io

//Test load/save
load io
save a
load io
save b

Link to the Google Doc DesignDocument

```
load a
save io

//Test loadui
loadui 8
save io

//Test loadi
loadi 30
save io

//Test bne (should print b)
load a
bne b 1
save io
load b
save io

//Test beq (should print a)
load b
beq b 1
save io
load a
save io

//Test slt
load a
slt b
save io         //0
load b
slt a
save io         //1

//Test slti
load a
slti 0
save io         //0
load a
slti 1000 // a number greater than a
save io         //1

//Test j
load a
j TAG
```

```
save io
TAG: load b
save io

//Test jal (should output a)
load b
jal TAG
save io
TAG:load a
j ra

//Test sw/lw/ms
ms 4            3
load a          3
sw 0            4
load b          3
sw 2            4
loadi 0         3
lw 0            4
save io         3
lw 2            4
save io         3

//Test sub
load a
sub b
save io

//Test add
load a
add b
save io

//Test addi
load a
addi 1
save io
load a
addi -1
save io

//Test and
load a
and b
```

```
save io

//Test or
load a
or b
save io

//Test ori
load a
ori 0x0F
save io
```

Appendix B: Zeen's Design Journal

MILESTONE 1 WORK:

Sunday, April 3, 2022

Working on the Milestone 1 [1h] + team meeting [2h]

- We decided to build an accumulator-based processor. We designed all the instructions, the instruction types, and the calling procedures.

Wednesday, April 6, 2022

Finalize the document [15 min]

- Specify the calling procedures.

MILESTONE 2 WORK:

Sunday, April 8, 2022

Working on the Milestone 2 [30min]

- We split the work, got updated with each other, and figured out the basic design idea of Milestone 2.

Wednesday, April 19, 2022

Revise the RTL [15 min]

- Double-checked the work Helen had done and get a better understanding of our proposed RTL design.

Wednesday, April 20, 2022

In-Class Project Meeting [1h15 min] + Evening work [4h]

- Specified the components we are going to use and wrote the assembler program. Also finalized the document.

MILESTONE 3 WORK:

Friday, April 19, 2022

First thoughts on the datapath [50 min]

- We designed the very first few parts of our datapath and decided what next time we should meet. At this stage, we simply started from the multicycle datapath and added a few components as we went through the RTLs & instructions. We adopted this Top-down way of designing so that we are not missing any part.

Sunday, April 24, 2022

Design the RTL datapath [2h]

- We had the team meeting from 5:40 to 6:40 to work on milestone 3. We decided to make the datapath multicycle. We tested all the instructions & their RTLs. Also, we figured that we will put our register on write first and then read so that we don't need to add another register to store the value of our ALU.

Tuesday, April 26, 2022

Revise components design [30 min]

- I updated the design of the components from the last milestone, to specify the name of each input and output wire so that it is easier for other team members to reference it in the later parts.

Wednesday, April 27, 2022

Evening work [3h]

- Implemented the 6 components in the Quartus with Verilog. The 6 basic components include the 16-bit ALU (alu16.v), 16-bit Memory (mem16.v), 16-bit mux with 1-bit opcode (mux1b16.v), 16-bit register (reg16.v), 8-bit to 16-bit sign-extended (se16.v), and 8-bit to 16-bit zero-extended (ze16.v). Aslo, I complied and figured all these components to be logically correct. And I renewed the Python-based assembler to match with our last update of the memory decision and new jump instruction and J type.

MILESTONE 4 WORK:

Saturday, April 30, 2022

M4 Team Meeting [2h]

- Decided to split the work. Also, we separated the full integration into 4 different subsystems - ALU, Mem, Wire, and PC. And each person will be responsible for one subsystem and corresponding components as well as the designs (the

unfinished work from milestone 3.) I will be responsible for the Memory subsystem and Mux, and IR, MDR, also Lab 7. And the assembler needs to be fixed.

Tuesday & Wednesday, May 2 & May 3, 2022

Individual Working [4h]

- Working on the memory. Get done with the memory.

MILESTONE 5 WORK:

Monday, May 9, 2022

Class time working + Individual Working [2h]

- Working on the Lab 7.

Tuesday, May 10, 2022

Class time working + Individual Working [3h]

- Re-implementing the memory system, as we can only use a 10-bit memory instead of 16-bit memory due to client restrictions.

Wednesday, May 11, 2022

Class time working + Individual Working [4h]

- Implementing the control and connected memory in Lab 7.

Thursday, May 12, 2022

Team meeting with prof + Individual Working [3h]

- Note-taker & decided to add the IO input and output to the system, and we will have a magic number to do the IOInput and IOOutput (save io and load io). Also, I demoed Lab 7 to the professor and get a further idea of re-design our memory subsystem.

MILESTONE 6 WORK:

Sunday, May 15, 2022

Team meeting + Working [5h]

- Athena is not coming for the team meeting for a while, and waiting for her reply & system design from her. Re-check the progress in Milestone 5 & figured the progress is not optimistic. Decided on the task re-assignment to catch up on the design for the final few weeks.
- Important! Changed the design of the memory map (since only 10-bit) we put the magic number to be 0x03FC (hardcoded FFFC)
- Designed the save IO and load IO

Monday & Tuesday, May 16 & 17, 2022

Class Time + Working [3.5h]

- Working on the project. Designed the control and made the control components work.

MILESTONE 7 WORK:

Wednesday, May 18, 2022

Class Time + Working [2h]

- Working on the project, debugging the memory substem and making it connected with the control.

Thursday, May 19, 2022

Class Time + Working [3.5h]

- Working on the project.

Saturday, May 21, 2022

Working with Helen [6h]

- Fix the bugs in the full design system, several wires are not connected due to poor naming and typos. Debugging in the ModelSim and tried to fix the most basic full test of each instructions. Until 11 pm, we figured most of the instructions with correct input & outputs.

Sunday, May 22, 2022

Working with Helen [9h] working until 2am

- Debugging the full design testing written by Helen. Fixed the Assembler, and re-assembled the realPrime. Tested the realPrime and made it work. We changed the memory to be working on the negative edge so that it will work fine with the control. Lastly collected all the information needed for the testbench and finished up the report.

Appendix C: Jermaine's Design Journal

Jermaine Brown, WORK LOG

MILESTONE 1 WORK:

Sunday, April 2, 2022

Met with team for [2 hours 30 min]

· We agreed on the accumulator and ran through some example code to figure out needed instructions. We decided to store data in memory and use the stack for securing data after a call to a function. 4 types of instructions identified A, AA, IA, and I.

Monday, April 3, 2022

Solo work done on assigned parts of milestone one.

Tuesday, April 4, 2022

Messaging between team on adjustments to design to account for 16 bits instead of 32 for instructions.

Wednesday, April 5, 2022

Team meeting in class [1-2] hours

· Revisions made to document to account for 16 bit instructions and established information was written down for the milestone. Code turned into instructions from rel-prime.

Revisions made later in the day after classes.

MILESTONE 2 WORK:

N/A

MILESTONE 3 WORK:

Sunday, April 24, 2022

Team meeting

· Discussed and created final Datapath. Decided to go with rising edge clock for now and not falling edge.

· Work distributed to team members, given integration plan and testing.

Tuesday, April 26, 2022

Solo work decided to go with 4 subsystems, then 2 more made up of groups of 2 subsystems for work moving forward

Wednesday, April 27, 2022

In class work

· Decided to go with 4 subsystems before creating whole Datapath and not two more.

Solo work

· Finished writing tests for subsystems.

MILESTONE 4 WORK:

Saturday, April 30, 2022

Team meeting

· Divided work and set deadlines as specified at the top of our document.

Sunday, May 1, 2022

Solo Work

· Implemented 2-bit mux but no testing. File completed but not committed.

Monday, May 2, 2022

Solo Work

- Made tests for 2-bit mux and began making tests for Alu component.

Tuesday, May 3, 2022

Class and Solo Work

- Finalized tests for Alu and 2-bit mux component
- Partial design for ALU subsystem but needs 3-bit mux for completion.
- Work committed and pushed.

MILESTONE 5 WORK:

Sunday, May 8, 2022

Team meeting

- Divided work Zeen and Athena for control and Helen and I finishing the Verilog implementation.

Tuesday, May 10, 2022

Solo Work

- Began working in top level Verilog but only ALU and PC subsystems are done.

Wednesday, May 11, 2022

Solo Work

- Added wires to top level accumulator still missing memory
- Discussed how to approach tests

MILESTONE 6 WORK:

Sunday, May 15, 2022

Team meeting

· Met and discussed testing assignments. While waiting on memory and control from Zeen I promised to work on testing components and adjusting design document.

Monday, May 16, 2022

Solo Work

· Checked documentation for mistakes in syntax and not clear details.

Tuesday, May 17, 2022

Missed class, no work done.

Wednesday, May 18, 2022

Solo Work

· Removed Clock from ALU and tested.

· Removed many syntax errors from PC with shift_left on ZE

· Removed Syntax errors on accumulatorFull and added output from Memory subsystem to Wire subsystem for I

Appendix D: Helen's Design Journal

4/3/2022

Go through some code, decide procedure calling conventions.

TODO:

4/6/2022

Adjust the instructions to fit into 16 bits.

Set all instructions to be 0x00XX and data to be 0xFFXX

Replaced $ra with a specific address 0xFFFE

$sp set at 0x1FFF

4/7/2022

Meeting with the professor

Write minimum code for the procedure call

Add the caller of relPrime

Write an assembler

4/8/2022

Finished writing the RTL

Thinking about changing jump or changing sw lw to make them into 4 cycles

4/16/2022

Tried writing the assembler using java

Get file not found error

Plan to try using Python

4/19/2022

Plan not to change the addressing mode, since sign extent is powerful in a way that it does not need the ALU, may add an addressing mode that uses ALU when we switch to pipelining

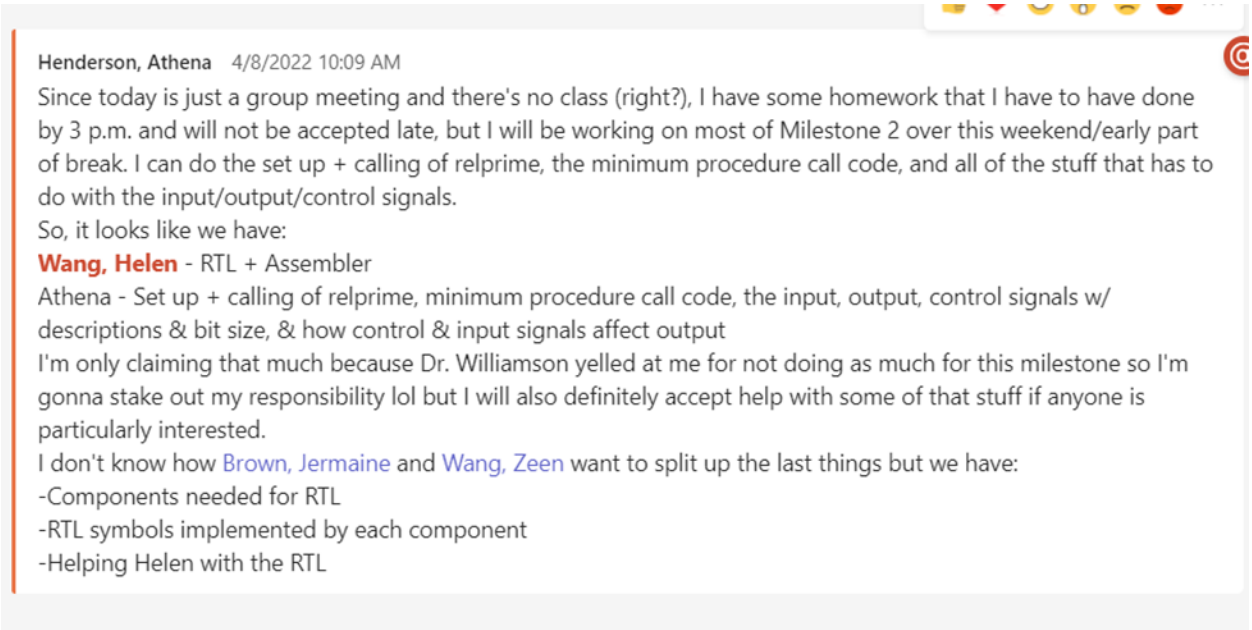
Go through the document of M2 and tell Zeen what to do.

Thinking about having a weekly team meeting time

4/20/2022

Zeen is available Sunday so thinking about 2 pm Sunday.

Current Roles:



Henderson, Athena 4/8/2022 10:09 AM

Since today is just a group meeting and there's no class (right?), I have some homework that I have to have done by 3 p.m. and will not be accepted late, but I will be working on most of Milestone 2 over this weekend/early part of break. I can do the set up + calling of relprime, the minimum procedure call code, and all of the stuff that has to do with the input/output/control signals.

So, it looks like we have:

Wang, Helen - RTL + Assembler

Athena - Set up + calling of relprime, minimum procedure call code, the input, output, control signals w/ descriptions & bit size, & how control & input signals affect output

I'm only claiming that much because Dr. Williamson yelled at me for not doing as much for this milestone so I'm gonna stake out my responsibility lol but I will also definitely accept help with some of that stuff if anyone is particularly interested.

I don't know how Brown, Jermaine and Wang, Zeen want to split up the last things but we have:

-Components needed for RTL

-RTL symbols implemented by each component

-Helping Helen with the RTL

For Zeen:

Zeen takes over Assembler

Thinking about adding pseudo instructions and improving RTL. Think better leave it like that since there is a possibility that we will switch to pipelining. Not may possible pseudo instruction, since we will need to have an extra register or store the value of acc on stack then retrieve it, maybe better without it.

Pseudo Instructions:

sw 0(m): load m, sw 0

Store m on stack with offset 0

slt a,b : load a, slt b

Compare a and b, if a<b, put 1 into acc, else put 0

addi m 1: load m, addi 1, save m

Increment m by imm

We do not have a way to address registers other than acc for most instructions. Can only do the 2nd method.

4/22/2022

Half finished with building datapath.

Added a cycle for sw and lw and do the pre-calculation for beq, bne instead since they are used more often

Tried to use the unused bit to switch between the two but beq, bne are AI type, they do not have the 3 extra bits.

If we are changing the source for ALU, we to choose between the op of the mux to be from IR or control. That means adding a mux to the opcode and using another bit from IR to be the op of this new op mux. But we cannot switch back.

TODO: Add the memory map, increment jump to use 11 bits

TODO: Finish datapath on Sunday

4/24/2022

Finished designing the datapath with Jermaine

Made minor changes to the RTL

Memory map, update j, jal green sheet and RTL

4/26/2022

Finished updating changes to j jal. Finished adding memory map.

Write some plan of testing the implementation, but the implementation plan itself is not yet written so may need to make changes to it.

The test I write basically focused on testing all the possible inputs of the control signals. Tests for flags and clock are also added for those that have these.

Decide to set memory to read and write on rising edge since in jal we want it to store the current PC before PC changes on falling edge. Does not have a cycle where memory do both read and write so it is okay to put read and write on the same edge.

Main should start at 0x0000

ALU op should be 3 since it does add, sub, and, or, slt

Finished the Complex FSM for M4

TODO: For M4, finish implementation of test bench, need to contact Zeen to see if he can meet during Thursday Friday class time.

TODO: Ask Zeen to use the assembler to get the new machine code that has address of main starting at 0x0000

4/29/2022

Plan to set up detail task during work time in class

No one came at 10:00

4/30/2022

Meet at 1 pm. Split the work.

Since all the parts are dependent on each other, maybe better to do the work together in a meeting. But others seem to prefer to split the works into relatively independent part and do it on their own.

DONE: write tests for reg

DONE: integration part PC

TODO: test PC

TODO: test 1 bit mux?

DONE: try to figure out problem with the waveform generator

TODO: push our files at 5 pm Tuesday

5/2/2022

Fixed issues with the waveform

Finished tb for reg, reg has no problem

Implemented PC using the block diagram file, but have problem testing

5/3/2022

Decide to re-implement PC using Verilog. Found that actually need 2 bit mux for that. Waiting for Jermain to finish the 2 bit mux.

Plan to write test for 1 bit mux.

TODO: push our files at 5 pm

Cannot solve the issue, seems to be a problem with PC, tb of reg run normally.

Comment out everything apart from the setup in tb, still crash. Suspect there is a loop in the connection of PC, don't see any error.

Maybe try letting someone else run the test bench

No time to do the test for 1 bit mux

5/6/2022

Added test for mux1b1

Error testing PC, find that it is because I assume that 00 is input A but Jermain implemented it as D instead. Texted Jermain about whether he want to change his design to match the patter of other mux.

We really should come up with a general design before going and implementing it separately.

Updated the document of PC and mux1b1

TODO: Wait for Jermain's reply, edit PC if needed

TODO: Wait for other's reply and schedule a meeting during the weekend.

5/8/2022

Secluded meeting at 1 pm, meeting canceled.

We will meet on Monday class time to make sure M4 is working.

We have 2 main tasks for M5, implementing and testing control, implementing and testing the whole datapath. 2 people will do the control, 2 people will do the datapath.

Decide the split on Monday.

5/9/2022

Split work, Jermaine and I will be doing the implementation and testing of datapath.

Zeen and Athena will be implementing and testing control

Changed the description that our save and load will do I/O when rt is 0x1000

Texted Zeen to add this feature to memory

5/10/2022

TODO: Solve pc test issue

Looked through our implementation files for tb and implementation:

- Memory test still writing by Zeen
- 16bit reg finished both
- Alu finished both
- Alu subsystem implemented no test banch
- PC finished both
- Mux1b1 finished both
- Mux1b16 need testing
- Mux 2b16 finished both
- Mux 3b16 no test file
- Se finished both

- Shift left finished both
- Ze finished both
- Wires subsystem missing both

5/11/2022

Wires subsystem no test file

Mux 3b16 no test file

Solve the problem of 10 bit memory

Control has .v file, no test bench

Write test implementation plan for some instructions, waiting for all subsystems and control to be implemented and tested.

Commit and push journal and document.

5/13/2022

Finished writing the test implementation plan for datapath.

Schedule meeting at Sunday 1 pm, waiting reply from Jermain and Zeen

Asked Jermaine to connect the subsystems

Write tb of mux1b16, but cannot run it due to syntax error in other files.

TODO: figure the error out with the control team and then run the test bentch

5/16/2022

Added a IRWrite bit that is set to 1 only in the fetch stage

Need update bubble diagram

Need update datapath:

Add IRWrite above IR

Delete the MemRead

5/18/2022

Tried to push journal and design doc, cannot due to git merge issue.

Compile failures when trying to test datapath, going to ask Zeen what to add.

TODO: ask Zeen about this tomorrow since I really need some sleep

5/20/2022

Debug our datapath with Zeen.

5/21/2022

Meet with Zeen to debug datapath, it has error loading design issue.

The alu subsystem has wrong number of wires assigned, it declared [2:0] instead of [1:0] for ALUSrcA and [3:0] instead of [2:0] for ALUSrcB.

We are not having the control's testbench.

We get our datapath working and I finished writing the tb. But we have too many log from control to see the message output of our tb.

5/22/2022

Meet at 1pm and work on the report and presentation with Athena and Jermaine. Zeen unable to come because he is stuck at ht.

Split the presentation slides

Need to figure out the cycle time

Meet at 6pm and work with Zeen to fully test our instructions and realPrime

Added a instruction jv ra, that will jump to the return address stored at ra, 0xFFFE or in our memory, 0x03FE.

Correct errors in our datapath and previous code.

Makes realPrime running

TODO: update design document

Finished meeting at 2am.

5/23/2022

Added the cycle time info to final report

Appendix E: Athena's Design Journal

Design Journal – henderae

Milestone 1

04/03/2022 – Group Meeting @ 6:00 p.m.

Was unable to make it to the meeting as some other commitments ran long, and then I had to still get dinner. I found out from Helen the next day that I was put in charge of the last three bullet points of number 1 under heading 3.

04/06/2022 – Individual Work

Was unable to come to lab period because I was still sick from the day prior. Opened the document and edited some formatting. Turned the instruction description and example programs into a table format. I also finished out the code fragments (while loop, for loop, if-else statements) with our assembly translation, machine code translation, and addresses.

Milestone 2

04/18/2022 to 04/21/2022 – Individual Work

Did not make it to the lab on the morning of 04/21, as I did not feel 100%. I did spend the entire lab time in my room working on the document. I created the set-up, calling, and use of

return value for relPrime, and the input/output/control signal names/bit size/description. Spent ~8 hours on Milestone 2.

Milestone 3

04/24/2022 – Group Meeting

Met starting at 4:00 p.m., went until 6:30 p.m. working through the requirements for milestone three. I explained the bullet points about the descriptions of how to implement different components and what it meant by the integration plan. We decided to split the integration into four main parts, one for each person. The PC is separated as the first piece, as it combines some combinational logic plus the mux for the input into the PC. The second piece we decided was memory plus IR and MDR since the complex part of that is the mux going into memory and IR and MDR are just registers. The third piece was the accumulator itself plus SP because those are the core of our processor and how it functions. The fourth piece is the multiplexers that determine the input to the ALU, the ALU itself, and the pathway out of the ALU.

We also decided that I would digitize the data path and update the Input/Output/Control table with the control inputs from the data path plus whatever was added to the RTL descriptions after Zeen updated them. Jermaine oversaw writing up the integration plan and how each piece was going to be tested. Helen oversaw figuring out how to test individual components and the description of how to implement them. Finally, Zeen was to update the RTL symbols based on the feedback from Dr. Williamson at our milestone meeting and work on the partial implementation of some components.

04/27/2022 – Lab Time

Zeen didn't make it to the lab, and Jermaine was late, so Helen and I worked on the control bubble diagram. I spent the lab period digitizing it as well as making any last-minute changes to the data path. We discussed combining some of the stages Jermaine had done for the integration plan, but I didn't catch exactly what we were combining. Helen also clarified some things with Dr. Williamson about how our tests were expected to function in terms of timing since we are reading on the rising edge and writing on the falling edge.

Milestone 4

04/30/2022 – Group Meeting

We decided to split the integration plan up between people and give the components inside those steps to the people who were working on the integration plan step itself to minimize the need to wait for others to finish their work. I took the wires subsystem (Acc + Sp) and volunteered to make the left shift and test the zero extend and the sign extend plus making the 5-to-1 mux.

05/01/2022 – Individual Work

Made progress on my assigned part of the milestone and made the left shift and the 5-to-1 mux. Struggled to get the testbench to work.

05/02/2022 – Individual Work

Finally got the test benches to work and connected properly to test the component. Wrote the test benches for the zero extend and sign extend. At this point, I have all my components individually created and tested effectively. I just need to get the wires subsystem done.

Milestone 5

05/08/2022 to 05/09/2022 – Group Meeting/Class Time

I couldn't make the group meeting, as my mom surprised me by driving to Terre Haute for Mother's Day. Did connect with Helen on what I missed, and she talked with Jermaine and me on Monday about how this milestone was getting divided. She and Jermaine decided to do the full connection and integration of the data path as they had more time later in the milestone timeline. Zeen and I need to implement control.

05/11/2022 – Lab Time

I finished up the wires subsystem with a stupid number of typos on my part. I still need to finish the test bench, but Zeen and Helen told me about the memory incident they ran into. They said that we still have 16 bits, but it only acts and seems as if we have 10 bits, so the stack pointer must get preset to a specific value each time. Plus, the information from the stack pointer must get processed someway to add five zeros to it each time. Zeen said he would tell me what needs to be changed to figure it out when he does the math.

Milestone 6

05/12/2022 – 05/18/2022

Zeen and I were put in charge of the control and its testbench. However, Zeen did not show up to the meeting, and I had no idea where was starting on things. I ended up getting overwhelmed with other classwork and sickness and didn't finish much. Started on a rough control testbench. I still don't have Git working on my computer.

Milestone 7

05/22/2022

We met as a group at the library. Zeen couldn't be there due to moving into his new apartment. We laid out the presentation and the report. I pointed out to Helen some of the control bits were the wrong size, but she said that she and Zeen had already fixed it. Jermaine was put in charge of cleaning up the design document so that it would look nice when it was turned into a series of jpg to be put into the final report. Helen started flushing out some of the writing pieces, and I worked through moving stuff from the final report to the presentation. Helen said she and Zeen would get together later that night to make sure they had everything running, but none of us were sure what we could change without breaking things.

Later that night, I got a series of messages from Helen about changing the data path. I attempted to follow the instructions, but I must've done it wrong.

05/23/2022

Upon arriving at the classroom for presentations, Zeen told me I had mixed up some things for the data path and asked me to fix them before our presentation. When I went to work on it, the file had been corrupted somehow, and so I quickly did some shady editing to make it at least accurate to our new system.

Later that evening, I spent ~2.5 hours updating/recreating the data path in Visio and updating all documentation that had been changed from making the processor work.