Processor Report

Accumulator

Team Name:

Red-2122a-01

Team Members:

Luke Dawdy, Taylor Goldman, Harrison Wight

Table of Contents

Include section & subsection numbers

PLACE IN FinalReport DIRECTORY

1. Intro to Processor
2. Instruction Set Design
    a. Implementation
    b. Final Model
    c. Testing Methodoloy
    d. Comments on Design
3. Project's unique features
    a. Robustness of HW/SW tests
    b. Capabilities of assembler/SW tools
    c. Function of kernel/system code
4. Extra features
5. Conclusion
6. Appendix A (Complete Design Documentation)
7. Appendix B (Design Process Journals)
8. Appendix C (Useful test results collected)
9. Other appendices

Executive Summary

Our group was assigned the task of designing and implementing a scaled-down instruction set processor which can execute a simple range of instructions that are in memory. The processor was thoroughly debugged, tested, and measured based on various parameters. To keep track of progress and the design of our processor, a shared design documentation was kept up to date with milestones and other process updates. The processor we created was a single-cycle accumulator-based processor which included three registers, an instruction memory, a data memory, an ALU, a control unit and other miscellaneous components. Our components were made using a mixture of Verilog and schematic design depending on what we found easier for the type of component. One of the requirements of the project to test functionality was to implement a program which determined a number's relative prime using Euclid's algorithm. The final model of our processor was simulated using the iSim simulator in Xilinx.

Introduction to Processor

For this project, besides the given instruction of creating a small instruction set processor that can execute programs stored in memory, there were other requirements to be met. First, the processor was required to use both an address and data bus that were 16-bit. Basic input and output capabilities were also another requirement for our processor. For the procedure side of things, our processor was required to perform general computations as well as handle parameterized and recursive functions. As stated previously, the one specific program required to run on our processor was computing the relative prime of a number using Euclid's algorithm. Besides all these requirements, we were able to design the processor in whatever way we desired.

Instruction Set Design

The processor's instructions are 16 bits, and the instruction format for each type can be shown in the figure below.

A-Type: immediate addressing type

- add b

Opcode (5 bits)	Immediate (11 bits)
00000	000 0000 0101 (assuming b = 5)

B-Type: PC-relative addressing type

- beq 0, END

Opcode (5 bits)	Immediate (4 bits)	Address (7 bits)
00100	0000	000 1111 (ex. address location of END function)

J-Type: pseudo direct addressing type

- j 0x02F

Opcode (5 bits)	Address (11 bits)
00110	000 0010 1111

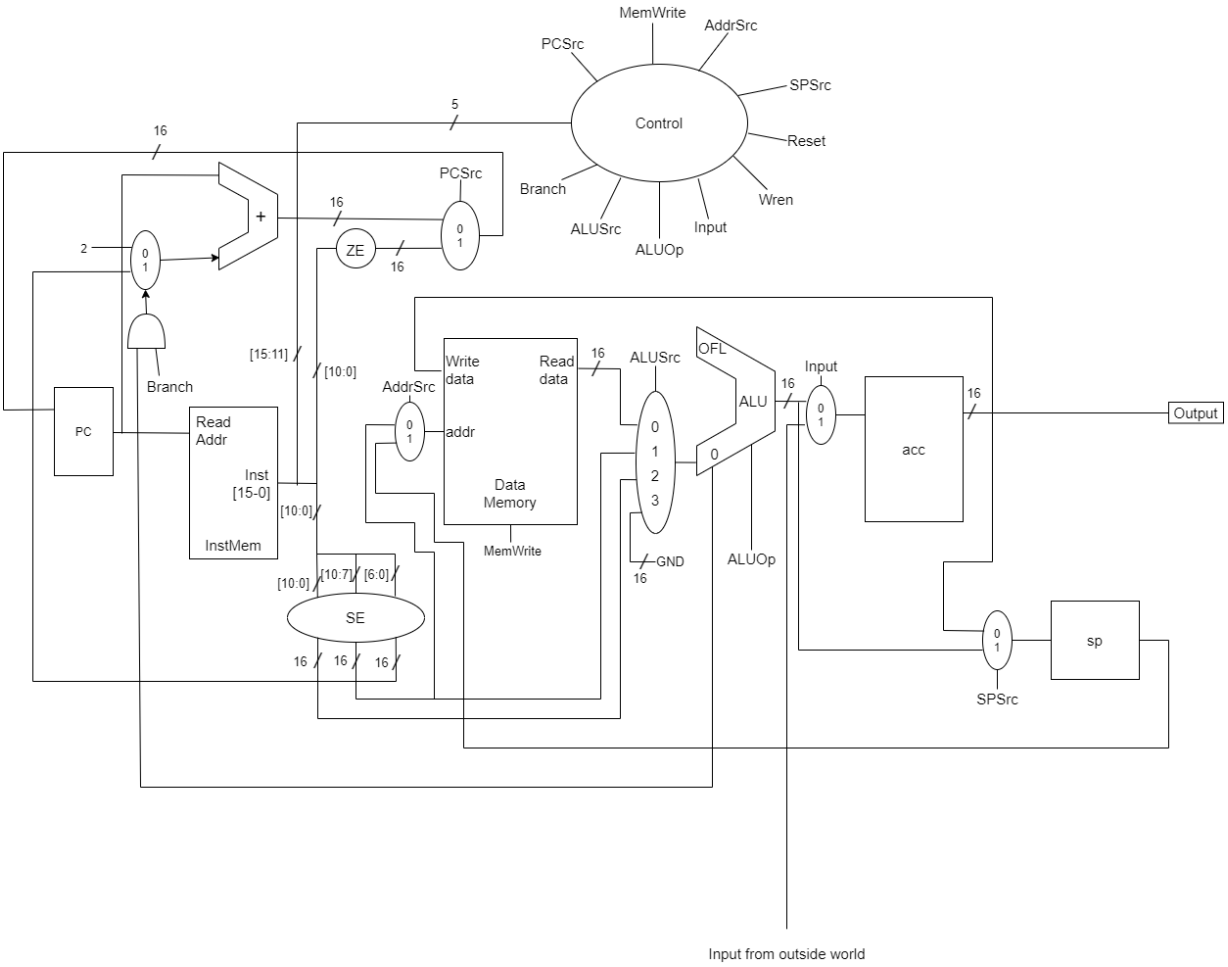
All instructions use a 5-bit opcode so we could implement up to 32 instructions. We wanted to have simple instructions so we were fine with using a 5th bit for the opcode if it meant we could simplify our instructions. Most of our instructions fall under the A-type instruction while the other instructions dealing with more of the addressing side of things are B- and J- type.

Implementation

We made an accumulator-based processor using three registers, two memories (instruction and data), an ALU, an adder, a control unit, and other necessary components. Our implementation was single cycle and used 20 instructions in its instruction set.

Final Model

The figure below is the data path for the processor.



Testing Methodology

For testing our processor, we first began by testing each of the individual components we created. Once we tested each individual component, we implemented a plan to test smaller sections of our data path by beginning with the first two components in a subsystem. We tested this first subsystem and then created a new subsystem using the first subsystem and the next component. We repeated this process by using the previous subsystem in tandem with the next component until we reached our final subsystem which represented the entire

data path. We then tested this overall data path to verify that our processor worked as expected.

Comments on Design

Our design made it difficult to execute more advanced programs, such as recursive function calls, or computations given we had difficulty dealing with our memory initially and maybe oversimplified our instruction set initially. We think it would've been beneficial to create a hybrid of both the accumulator and load-store for our processor. Overall, we believe our design wasn't bad but had much room for improvement and could have been better implemented and completed with better planning.

Appendix A: Design Documentation

Design Document

Team Name:

Red-2122a-01

Team Members:

Luke Dawdy, Taylor Goldman, Harrison Wight

High level design description:

Our team is designing a processor with an accumulator instruction set architecture. Our processor will include a single register, the accumulator, for all instructions to be used by the programmer. All 22 instructions fit into the categories of ALU operations or memory access. Given these two categories, there are three types of instructions available for the programmer to use, A-type, B-type, and J-type instructions. Using an accumulator processor allows for small, quick instructions and very little memory access.

Registers:

Accumulator register ($acc): This register is available for the general purpose of the assembly language programmer. It can be used for any instruction, for example, add, sub, jump, etc.

Input register ($in): This register is used for allowing input values to be used within our processor and is not accessible by the programmer. It is only accessed when the instruction *getin* is called upon.

Output register ($out): This register allows for our processor to output values outside of our processor and is not accessible to the programmer. It is only used when the instruction *getout* is called upon.

$sp: This register holds the address of information that is being stored for a variable.

Machine language format:

A-Type: 2 bytes

- Opcode: denotes the operation and format of an instruction
- Immediate: the immediate / constant value being used for the instruction

Opcode (5 bits)	Immediate (11 bits)

B-Type: 2 bytes

- Opcode: denotes the operation and format of an instruction
- Immediate: the immediate / constant value being used for the instruction
- Address: the address of the register source operand

Opcode (5 bits)	Immediate (4 bits)	Address (7 bits)

J-Type: 2 bytes

- Opcode: denotes the operation and format of an instruction
- Address: the address of the register source operand

Opcode (5 bits)	Address (11 bits)

Syntax / Semantics of instructions:

Name	Instruction	Format	Opcode
Add	add	A	00000
Add Immediate	addi	A	00001
And	and	A	00010
And Immediate	andi	A	00011
Branch Equal	beq	B	00100
Branch Not Equal	bne	B	00101
Jump	j	J	00110
Jump and Link	jal	J	00111
Load	load	A	01000
Or	or	A	01001
Or Immediate	ori	A	01010
Branch Greater Than	bgt	B	01011
Branch Less Than	blt	B	01100
Shift Right	sr	A	01101
Shift Left	sl	A	01110
Store	store	A	01111
Subtract	sub	A	10000
Get Input	getin	X	10001
Clear	clear	X	10010
Return	return	X	10011

English & Symbolic description of instruction behavior:

*var should be replaced with whatever variable is being accessed

Add: Add whatever value is used in the instruction to $acc and store the result in $acc.

R[acc] = R[acc] + Mem[SE(var)]

Add Immediate: Add the immediate value used in the instruction to $acc and store the result in $acc.

R[acc] = R[acc] + SE(Immediate)

And: And the value used in the instruction with $acc and store the result in $acc.

R[acc] = R[acc] & Mem[SE(var)]

And Immediate: And the immediate in the instruction with $acc and store the result in $acc.

R[acc] = R[acc] & SE(Immediate)

Branch Equal: If the value used in the instruction is equal to the value in $acc, then the program counter will branch to the address you specified in the instruction.

if(R[acc] == Mem[SE(var)])

PC = PC + 2+ SE(BranchAddr)

Branch Not Equal: If the value used in the instruction is not equal to the value in $acc, then the program counter will branch to the address you specified in the instruction.

if(R[acc] != Mem[SE(var)])

PC = PC + 2 + SE(BranchAddr)

Jump: The program counter will jump to the address you include in the instruction.

PC = ZE(JumpAddr)

Jump and Link: The program counter will jump to the address you include in the instruction, and the return address will become PC + 2 to return to the current instruction upon function completion.

R[ra] = PC + 2; PC = ZE(JumpAddr)

Load: Load the value used in the instruction into $acc.

R[acc] = Mem[SE(var)]

Or: Or the value used in the instruction with $acc, and store the result in $acc.

R[acc] = R[acc] | Mem[SE(var)]

*Or Immediate***:** Or the immediate used in the instruction with $acc and store the result in $acc.

R[acc] = R[acc] | SE(Immediate)

Branch Greater Than: Checks if the value in $acc is greater than the value you are passing into the instruction. If $acc is greater than the value, the branch will be taken. Else, the program counter will be incremented.

R[acc] = (R[acc] > Mem[SE(var)])? PC = ZE(BranchAddr) : PC = PC + 2

Branch Less Than: Checks if the value in $acc is less than the value you are passing into the instruction. If $acc is less than the value, the branch will be taken. Else, the program counter will be incremented.

R[acc] = (R[acc] < Mem[SE(var)])? PC = ZE(BranchAddr) : PC = PC + 2

Shift Left: Shifts the bits in the $acc to the left by the amount specified in the instruction.

R[acc] = R[acc] << Mem[SE(var)]

Shift Right: Shifts the bits in the $acc to the right by the amount specified in the instruction.

R[acc] = R[acc] >> Mem[SE(var)]

Store: Stores the value from $acc into the variable you specify in the instruction.

Mem[var] = R[acc]

Subtract: Subtracts the value specified in the instruction from $acc and stores the result in $acc.

R[acc] = R[acc] - Mem[SE(var)]

Get Input: Gets the input value from the outside world and stores it in $acc.

R[acc] = input

Clear: Stores the value zero into $acc to clear it.

R[acc] = 0x00

Return: Outputs the value in $acc to the outside world.

output = R[acc]

Push: Puts the value in the accumulator onto the stack and adjusts the stack position.

$sp = $sp - 2

Mem[$sp] = R[acc]

Pop: Takes the value of the top of the stack and puts it in the accumulator.

R[acc] = Mem[$sp]

$sp = $sp + 2

Translating assembly to machine language:

A-Type: immediate addressing type

- add b

Opcode (5 bits)	Immediate (11 bits)
00000	000 0000 0101 (assuming b = 5)

B-Type: PC-relative addressing type

- beq 0, END

Opcode (5 bits)	Immediate (4 bits)	Address (7 bits)
00100	0000	000 1111 (ex. address location of END function)

J-Type: pseudo direct addressing type

- j 0x02F

Opcode (5 bits)	Address (11 bits)
00110	000 0010 1111

Procedure call conventions:

Name	**Use**	**Preserved Across a Call?**
$acc	General use by all	No
$sp	Used to keep track of the top of the stack. Not directly accessible by user.	Yes
$ra	Used to store where the program will return to upon completion of a function. Not directly accessible by user.	Yes
~~$fp~~	~~Used to hold the base address of the function. Not directly accessible by user.~~	~~Yes~~

Memory Map:

0xFFFF XXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXXXXXXX **0x03FF** (top of usable memory) **$sp => 0x0200**
0x01FF **PC => 0x0000**

Example assembly language program (relprime):

(0x00) getin

store n

store a

clear

```
addi 2
store m
store b
clear
LOOP: jal GCD
beq 1, EXIT
load m
addi 1
store m
store b
clear
load n
store a
J LOOP
GCD: load a
beq 0, RETURNB
clear
load b
WHILEB: beq 0, RETURNA
clear
load a
sgt b, ASUBB
clear
load b
sub a
store b
j WHILEB
ASUBB: sub b
```

```
store a
load b
j WHILEB
RETURNB: load b
j ra
RETURNA: load a
j ra
EXIT: clear
load m
return
```

Machine Code:

1000 1xxx xxxx xxxx	(getin, assuming input = 0x13B0)
0111 1000 0000 1110	(store n)
0111 1000 0000 0001	(store a)
1001 0xxx xxxx xxxx	(clear)
0000 1000 0000 0010	(addi 2)
0111 1000 0000 1101	(store m)
0111 1000 0000 0010	(store b)
1001 0xxx xxxx xxxx	(clear)
0011 1000 0000 1001	((0x12) LOOP: jal GCD)
0010 0000 1110 0100	(beq 1, EXIT)
0100 0000 0000 1101	(load m)
0000 1000 0000 0001	(addi 1)
0111 1000 0000 1101	(store m)
0111 1000 0000 0010	(store b)
1001 0xxx xxxx xxxx	(clear)
0100 0000 0000 1110	(load n)

```
0111 1000 0000 0001        (store a)
0011 0000 0000 0100        (J LOOP)
0100 0000 0000 0001        ((0x24) GCD: load a)
0010 0000 0100 0110        (beq 0, RETURNB)
1001 0xxx xxxx xxxx        (clear)
0100 0000 0000 0010        (load b)
0010 0000 0100 1010        ((0x2E) WHILEB: beq 0, RETURNA)
1001 0xxx xxxx xxxx        (clear)
0100 0000 0000 0001        (load a)
0101 1001 0011 1110        (bgt b, ASUBB)
1001 0xxx xxxx xxxx        (clear)
0100 0000 0000 0010        (load b)
1000 0000 0000 0001        (sub a)
0111 1000 0000 0010        (store b)
0011 0000 0010 1110        (j WHILEB)
1000 0000 0000 0010        ((0x3E) ASUBB: sub b)
0111 1000 0000 0001        (store a)
0100 0000 0000 0010        (load b)
0011 0000 0010 1110        (j WHILEB)
0100 0000 0000 0010        ((0x46) RETURNB: load b)
0011 0000 0001 0010        (j ra)
0100 0000 0000 0001        ((0x4A) RETURNA: load a)
0011 0000 0001 0010        (j ra)
1001 0xxx xxxx xxxx        ((0x4E) EXIT: clear)
0100 0000 0000 1101         (load m)
1001 1xxx xxxx xxxx        (return)
```

Common operations in assembly language / machine code translation:

Loading Variable into Register: $acc = a;

```
load a
```

Machine Code:

```
0100 0000 0000 0001        (load a, assuming a = 1)
```

Iteration: while(true) { count = count + 1; }

```
(0x27) LOOP:
    load count
    addi 1
    store count
    j LOOP
```

Machine Code:

```
0100 0000 0011 0110        (load count, assuming count = 54)
0000 1000 0000 0001        (addi 1)
0111 1000 0011 0111        (store count)
0011 0000 0010 0111        (j LOOP)
```

Conditional Statements: if (a < 2) { a = a + 1; }

```
    load a
    blt 2, ADDONE
    j QUIT
(0x08) ADDONE:
    addi 1
    store a
(0x0D) QUIT:
```

Machine Code:

```
0100 0000 0000 0001        (load a, assuming a = 1)
```

0110 0001 0001 0000 (blt 2, ADDONE)

0000 1000 0000 0001 (addi 1)

0111 1000 0000 0001 (store a)

Recursive Function:

```
iterateToZero(n) {
    if ( n == 0) {
        return n;
    }
    return iterateToZero(n – 1);
}
```

Assembly Code:

```
            getin
(0x00) ITERATETOZERO:
            beq 0, CASE0
            sub 1
            j ITERATETOZERO
(0x05) CASE0:
            return
```

Machine Code:

1000 1xxx xxxx xxxx (getin, $acc = n assuming n = 1 in this case)

0010 0000 0000 0101 (beq 0, CASE0)

1000 0000 0000 0001 (sub 1)

0011 0000 0000 0000 (j ITERATETOZERO)

0010 0000 0000 0101 (beq 0, CASE0)

1001 1xxx xxxx xxxx (return value of n)

Reading Data from Input port:

getin

Machine Code:

1000 1xxx xxxx xxxx (getin, gets input from outside world)

Writing to Output port:

return

Machine Code:

1001 1xxx xxxx xxxx (return, returns data to outside world)

RTL:

Add / Sub / And / Or / Sl / Sr	**Addi / Andi / Ori**	**B-Type Instruction**	**J-Type Instruction**
newPC = PC + 2 PC = newPC Inst = Mem[PC] A = Reg[acc] B = SE(Mem[var]) Result = A op B Reg[acc] = Result	newPC = PC + 2 PC = newPC Inst = Mem[PC] A = Reg[acc] B = SE(Immediate) Result = A op B Reg[acc] = Result	newPC = PC + 2 PC = newPC Inst = Mem[PC] A = Reg[acc] B = SE(Immediate) Result = newPC + ZE(BranchAddr) If(A op B) PC = Result	newPC = PC + 2 PC = newPC Inst = Mem[PC] PC = ZE(BranchAddr)
Store	**Load**	**Getin / Return**	**Clear**
newPC = PC + 2 PC = newPC Inst = Mem[PC] A = Reg[acc] B = Mem[var] Mem[B] = A	newPC = PC + 2 PC = newPC Inst = Mem[PC] A = Reg[acc] B = Mem[var] Mem[A] = B	newPC = PC + 2 PC = newPC Inst = Mem[PC] **Getin:** Reg[acc] = Input **Return:** Output = Reg[acc]	newPC = PC + 2 PC = newPC Inst = Mem[PC] A = Reg[acc] B = ZE(0) Result = A && B Reg[acc] = Result

Needed components to implement RTL:

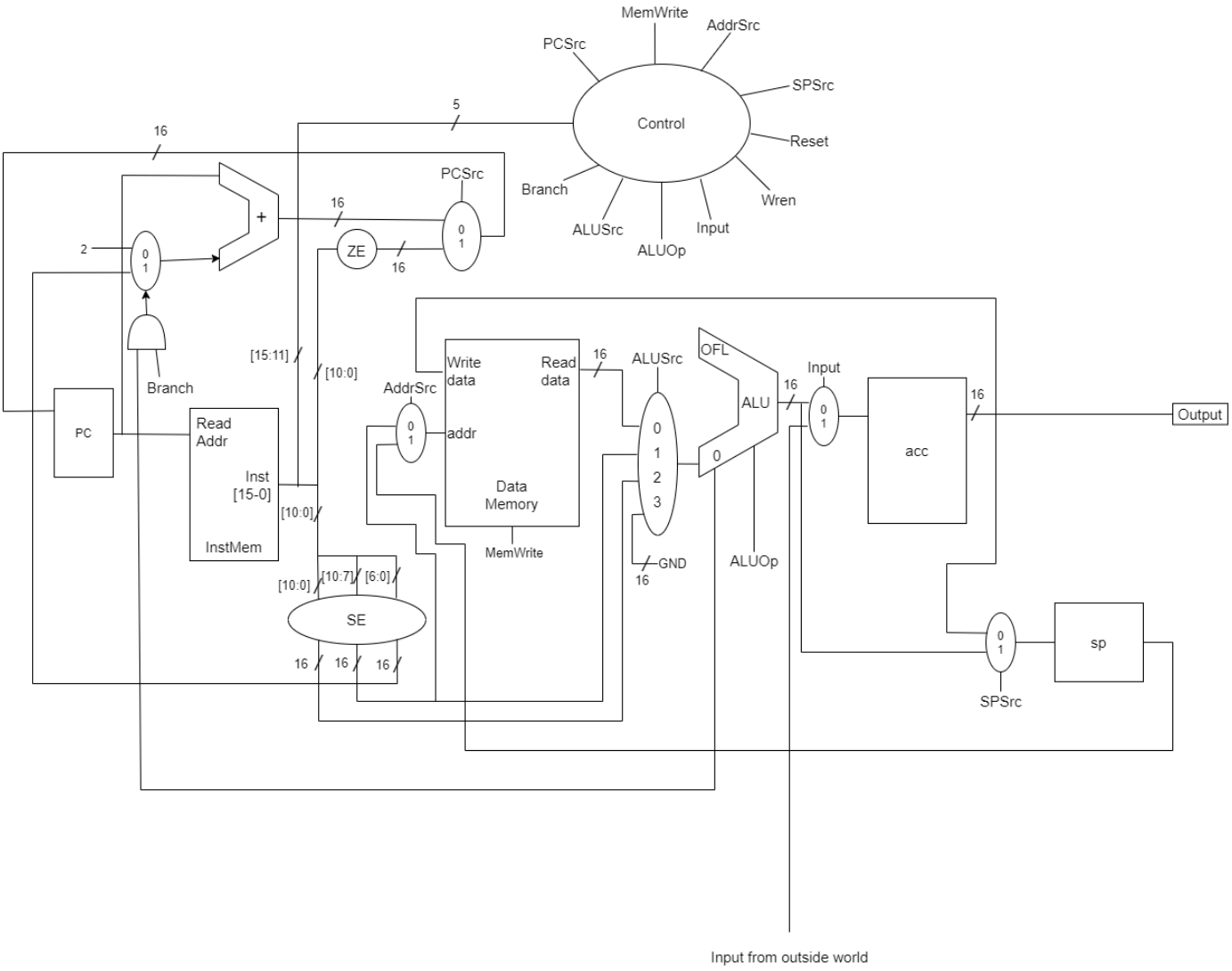
- Instruction Memory:
  - Input: PC, Output: 16-bit instruction, Control: N/A
  - Takes in the PC value and then finds the instruction to output at that address

    - Inst
- Register File:
    - Input: 16-bit instruction, Output: 5-bit opcode, 4/11-bit immediate, 7/11-bit address, Control: IRWrite
    - Takes in instruction and translates the parts into individual components to be handled by ALU
    - Reg[acc]
- Sign Extend:
    - Input: 11-bit immediate, Output: 16-bit immediate, Control: n/a
    - A combinational logic unit that sign-extends an 11-bit input to produce a 16-bit output
    - SE
- Zero Extend:
    - Input: 7/11-bit branch address, Output: 16-bit address, Control: n/a
    - A combinational logic unit that zero-extends a 7/11-bit input to produce a 16-bit output
    - ZE
- ALU:
    - Input: 16-bit A, 16-bit B, Output: result, Control: opcode
    - An arithmetic logic unit that performs operations on our inputs based on the instruction's opcode
    - A, B
- Adder (for PC):
    - Input 1: PC, Input 2: 2, Output: newPC, Control: n/a
    - An ALU that inputs our PC and 1, and adds them together to get our newPC value
    - PC, newPC

Process for double-checking RTL for errors:

- Check that the PC is incremented correctly before/after each instruction
- Verify all control bits are set correctly depending on the opcode of the instruction being run
- Before doing an ALU Operation, verify the registers/immediates are holding the desired value
- Assure the ALU outputs the correct value based on the ALUOp
- Verify the value written to the accumulator is the actual output value from the ALU

Block diagram of datapath:



Description of each control signal:

- *Branch*: the Branch signal is used in combination with the ALU zero flag to determine if a branch should occur; if both jump and the zero flag are high, the PC-relative address will be added to PC rather than just incrementing by 2
- *PCSrc*: the PCSrc signal determines whether PC is going to be using the value from the PC adder or taking the zero-extended branch address (from a J-type instruction)
- *MemWrite*: the MemWrite control signal allows memory to be written to if it is necessary for the instruction and is set to low if nothing is being written to memory
- *ALUSrc*: the ALUSrc control signal chooses which value will be the second input to the ALU; it chooses between the data read from memory, and two different immediate values chosen based off the instruction
- *ALUOp*: the ALUOp control signal goes into the ALU Control to decide which operation is going to be performed on the inputs to the ALU

- *Input*: the Input control signal chooses whether or not the accumulator will receive the ALU result or input from the outside world
- *WrEn*: the WrEn control signal determines if the accumulator can be written to or not
- *Reset*: the Reset control signal will be used to initialize the processor
- *SPSrc*: the SPSrc control signal determines if the stack pointer register gets the output from the ALU or the value in the accumulator
- *AddrSrc*: the AddrSrc control signal determines if the Data Memory uses the sign extended address or the value stored in the stack pointer

Plan to implement each component in hardware:

- Instruction Memory
    - Input: PC[16 bits]
    - Operations: Reads the address saved in the register PC from a memory block and outputs the data read, which is the next instruction
    - Output: Instruction[16 bits]
- Register File (only acc register)
    - Input: ALU output[15:0]
    - Operations: save input to register on clock cycle if wren (write enable) is high
    - Output: acc[15:0]
- Sign Extend
    - Input: inst[10:0], inst[10:7], and inst[6:0]
    - Operations: sets bits 15 – 11, 15 – 11, and 15 – 6 respectively, to the value of the most significant bit from its corresponding instruction (bit 10, 10, or 6)
    - Output: 16-bit sign extension
- Zero Extend
    - Input: inst[10:0]
    - Operations: Sets bits 11 – 15 of input to zero to make input non-negative and 16 bits.
    - Output: inst [10:0] extended to 16-bits with 0 in all positions following bit 10.
- ALU
    - Input: acc[15:0], ALUsrc?
    - Operations: Add, Subtract, And, or Or together both inputs depending on control bits
    - Output: acc OP B [15:0],
- Adder (for PC)
    - Input: pc
    - Operations: newPC = pc + 2
    - Output: newPC

Description of tests for each component:

- Instruction Memory
    - Input range of addresses and verify that the correct instruction sub part values are returned correctly

- Data Memory
    - Test that any address can be properly read from and written to in memory.
- Sign Extend
    - Extend positive integer (most significant bit is 0) to 16 bits to verify is most significant bits are 0
    - Extend negative integer (most significant bit is 1) to 16 bits to verify is most significant bits are 1
- Zero Extend
    - Extend positive integer (most significant bit is 0) to 16 bits to verify is most significant bits are 0
    - Extend negative integer (most significant bit is 1) to 16 bits to verify is most significant bits are 0
- ALU
    - Test adding function by adding simple cases (i.e., 0001 + 0001 = 0010)
    - Test subtraction function with simple cases (i.e., 0001 - 0010 = 1001)
    - Test bitwise AND and OR functions on simple cases (i.e., 0010 & 0011 = 0010 and 0010 | 0011 = 0011)
- Adder (for PC)
    - Set PC at certain values, and verify that newPC is PC incremented by 2
- Accumulator
    - Input values into the accumulator register and verify that it still contains that data on return
- Stack Pointer
    - Input values into the stack pointer register and verify that it is holding the correct value on return

Integration plan for iteratively combining parts into a complete datapath:

- Integrate PC adder and Instruction Memory; this will let us get instructions from memory specified by the spot in memory.
- Add Zero Extend and Sign Extend; this will extend the instructions from memory to 16 bits for use in the ALU
- Add Data Memory: uses the sign-extended address to access data in memory to be read and used in the ALU or writes data from the accumulator to the sign-extended address
- Add ALU; operates on the values taken from the data memory or directly from the sign extender and outputs them based on the chosen operation by the ALUOp
- Add Accumulator; takes the result from the ALU and stores it into the Accumulator.
- Add Stack Pointer; takes a result from the ALU or current value in the accumulator to access a place on the stack which contains data

Tests for each step of integration plan:

- PC adder & Instruction Memory

    - Input PC with values in the adder and verify that the instruction memory is outputting the instruction at the desired address
- Add Sign-Extender and Zero-Extender to PC Adder & Instruction Memory
    - Change PC with values from the adder and verify that all values returned from the sign extender and zero extender are equivalent to the values specified in the instruction with extension
- Add Data Memory to previous combination
    - Change PC with values from the adder and verify that all values read from the data memory are as expected with given instruction and addresses
- Add ALU to previous combination
    - Change PC with values from the adder and verify that the ALU result is correct for the instruction run and values input (correct operation done)
- Add Accumulator to previous combination
    - Change PC with values from the adder and verify that the accumulator is holding the same result as expected from the desired instruction
- Add Stack Pointer to previous combination
    - Change PC with values from the adder and verify that the stack pointer is holding the correct address as expected from the desired instruction

Truth Tables for Combinational Control Units:

Reset

Input:	Output:
0	n/a
1	Initializes processor

AddrSrc

Input:	Output:
0	SE Immediate
1	Address from Stack Pointer

PCSrc

Input:	Output:
0	PC Adder Result
1	ZE BranchAddr

SPSrc

Input:	Output:
0	Accumulator Value
1	ALU Result

MemWrite

Input:	Output:
0	Memory cannot be written
1	Memory can be written

WrEn

Input:	Output:
0	Cannot write to accumulator
1	Can write to accumulator

ALUSrc

Input:	Output:
0	Read data from memory
1	OUT_4 (SE immediate)
2	OUT_11 (SE immediate)
3	0

Branch **Zero Flag**

Input:	Input:	Output:
0	0	2
0	1	2
1	0	2
1	1	SE BranchAddr

Input

Input:	Output:
0	ALU Result
1	Input from Outside World

ALUOp

Input:	Output:
000	Add
001	Sub
010	And
011	Or
100	Shift Left

101	Shift Right

Control Truth Table

Opcode	Inst	Branch	PCSrc	MemWrite	ALUSrc	ALUOp	Input	Wren	RST	SPSrc	AddrSrc
00000	**add**	0	0	0	00	000	0	1	0	x	x
00001	**addi**	0	0	0	10	000	0	1	0	x	x
00010	**and**	0	0	0	00	010	0	1	0	x	x
00011	**andi**	0	0	0	10	010	0	1	0	x	x
00100	**beq**	1	0	0	01	001	0	0	0	x	x
00101	**bne**	1	0	0	01	001	0	0	0	x	x
00110	**j**	0	1	x	x	x	x	x	x	x	x
00111	**jal**	0	1	x	x	x	x	x	x	x	x
01000	**load**	0	0	1	x	x	0	0	0	1	0
01001	**or**	0	0	0	00	011	0	1	0	x	x
01010	**ori**	0	0	0	10	011	0	1	0	x	x
01011	**bgt**	1	0	0	01	001	0	0	0	x	x
01100	**blt**	1	0	0	01	001	0	0	0	x	x
01101	**sr**	0	0	0	10	101	0	1	0	x	x
01110	**sl**	0	0	0	10	100	0	1	0	x	x
01111	**store**	0	0	1	x	x	0	0	0	x	1
10000	**sub**	0	0	0	00	001	0	1	0	x	x
10001	**getin**	0	0	0	x	x	1	1	0	x	x
10010	**clear**	0	0	0	11	010	0	1	0	x	x
10011	**return**	x	x	x	x	x	x	x	x	x	x

System Test Plan:

- Simple programs:
  - Assembly Language for TEST 1 (if statement)

    load a

    blt 2, ADDONE

        j QUIT

    (0x08) ADDONE:

        addi 1

        store a

    (0x0D) QUIT:

  - Hex Representation for TEST 1 (if statement)

0x4001 (load a, assuming a = 1)

0x6108 (blt 2, ADDONE)

0x0801 (addi 1)

0x7801 (store a)

- Assembly Language for TEST 2 (recursive function)

```
getin
(0x04) ITERATETOZERO:
        beq 0, CASE0
        sub 1
        j ITERATETOZERO
(0x0C) CASE0:
        return
```

- Hex Representation for TEST 2 (recursive function)

0x8800 (getin, $acc = n assuming n = 1 in this case)

0x2005 (beq 0, CASE0)

0x8001 (sub 1)

0x3004 (j ITERATETOZERO)

0x2006 (beq 0, CASE0)

0x9800 (return value of n)

- Larger programs:
  - Assembly Language: RelPrime

```
(0x00) getin
store n
store a
clear
addi 2
```

```
store m
store b
clear
LOOP: jal GCD
beq 1, EXIT
load m
addi 1
store m
store b
clear
load n
store a
J LOOP
GCD: load a
beq 0, RETURNB
clear
load b
WHILEB: beq 0, RETURNA
clear
load a
sgt b, ASUBB
clear
load b
sub a
store b
j WHILEB
ASUBB: sub b
```

store a

load b

j WHILEB

RETURNB: load b

j ra

RETURNA: load a

j ra

EXIT: clear

load m

return

- Hex Representation

0x88xx	(getin, assuming input = 0x13B0)
0x780E	(store n)
0xE801	(store a)
0x90xx	(clear)
0x0802	(addi 2)
0x780D	(store m)
0x7802	(store b)
0x90xx	(clear)
0x3809	((0x12) LOOP: jal GCD)
0x20E4	(beq 1, EXIT)
0x400E	(load m)
0x0801	(addi 1)
0x780E	(store m)
0x7802	(store b)
0x90xx	(clear)
0x400E	(load n)

0x7801	(store a)
0x3004	(J LOOP)
0x4001	((0x24) GCD: load a)
0x2046	(beq 0, RETURNB)
0x90xx	(clear)
0x4002	(load b)
0x204A	((0x2E) WHILEB: beq 0, RETURNA)
0x90xx	(clear)
0x4001	(load a)
0x593E	(bgt b, ASUBB)
0x90xx	(clear)
0x4002	(load b)
0x8001	(sub a)
0x7802	(store b)
0x302E	(j WHILEB)
0x8002	((0x3E) ASUBB: sub b)
0x7801	(store a)
0x4002	(load b)
0x302E	(j WHILEB)
0x4002	((0x46) RETURNB: load b)
0x3012	(j ra)
0x4001	((0x4A) RETURNA: load a)
0x3012	(j ra)
0x90xx	((0x4E) EXIT: clear)
0x400D	(load m)
0x98xx	(return)

Performance:

- Total number of bytes required to store Euclid's algorithm and relprime:
  - 84 bytes
- Total number of instructions executed when relprime is called with 0x13B0 (5040):
  -
- Total number of cycles required to execute relprime:
  -
- Average CPI based on data collected in Steps 2 and 3:
  - 1 cycle
- Cycle time for our design:
  - 3.921 ns (255.037 MHz)
- Total execution time for relprime:
  -
- Device utilization summary:

  Selected Device: 3s500efg320-4

  Number of Slices: 190 out of 4656 4%

  Number of Slice Flip Flops: 67 out of 9312 0%

  Number of 4 input LUTs: 349 out of 9312 3%

  Number of IOs: 50

  Number of bonded IOBs: 44 out of 232 18%

  IOB Flip Flops: 16

  Number of GCLKs: 1 out of 24 4%

Change Log:

October 12:

- Crossed out $sp and $fp on procedure call convention diagram. Will remove entirely unused at a later date.
- Updated machine code translations of example assembly language programs.
- Added RTLs as well as the necessary components for implementation and error checks.
- Added functions to program to clear accumulator and to

October 20:

- Additions: Datapath Block Diagram, Control Diagram Description, Component Implementation Plan, Component Test Plans, Integration Plan outline and Integration Test Outline.
- Updated Memory Map: Can only use from 0x0000 to 0x3FFF

October 31:

- Additions: Control Truth Table
- Changed Set Less Than (SLT) and Set Greater Than (SGT) to Branch Less Than (BLT) and Branch Greater Than (BGT), respectively

Appendix B: Luke's Design Process Journal

CSSE 232 Design Journal – Luke Dawdy

Milestone 1 Work:

Sunday, October 3rd, 2021

- Collectively worked on the general design of our processor by coming up with instruction types, specific instructions, and other details (2.5 hours)
    - Worked on high-level design description so we could begin working on the specifics of our instructions, instruction types, etc.
    - Created a table which lists out each instruction and its details including the name, mnemonic, opcode, format, and description.

Monday, October 4th, 2021

- Worked on adding additional instructions into our ISA to allow for differentiation between adding an immediate and adding a variable value from memory. This required us to clarify our instruction types to make sure all our instructions fell within one of our three types. (45 minutes)

Tuesday, October 5th, 2021

- Worked with Taylor Goldman to finalize our fragment programs in both our assembly language and machine code (1.5 hours)
    - Before completing these, we had to adjust the sizes of different elements within our instruction types and clearly identifying what instructions were what types

Milestone 2 Work:

Sunday, October 10th, 2021

- Met with Taylor Goldman to make adjustments to M1 based off meeting comments (2 hours)
    - I fixed the relprime method assembly language and machine code implementations
    - We adjusted instructions to allow proper implementation of all fragments and code
    - Added sign and zero extension to all immediates within instructions
    - Added clear instruction so we can make sure $acc is empty when necessary

Tuesday, October 12th, 2021

- Met with group in class to continue work on M2 (30 minutes)
    - Added the process for checking RTL errors section and checked over RTL chart
- Met with Taylor Goldman to finish up M2 (1.5 hours)
    - Adjusted the process for checking RTL errors section to compensate for new RTL
    - Finished up the RTL chart and added component details for RTL (inputs, outputs, etc.)
    - Completed changes to machine code for relprime method

Milestone 3 Work:

Monday, October 18th, 2021

- Met with Taylor Goldman to fix up M2 problems (1 hour)
    - Changed some RTL to correctly set up data path
    - Decided whether to use sign extension or zero extension for different scenarios
    - Began designing the memory map

Tuesday, October 19th, 2021

- Met with Taylor Goldman and Harrison Wight to work on M3 (1.5 hours)
    - Decided what addresses in the memory map were going to be for what purpose
    - Worked through how variables are stored in memory and explained load/store capabilities more clearly
    - Began data path drawing
- I continued working through the data path drawing and updated the components list accordingly (30 minutes)

Wednesday, October 20th, 2021

- Met with whole team during class time (1.5 hours)
    - Finished/adjusted data path to include control bits and input/output capabilities

    - Wrote descriptions for individual component tests
    - Ideated subsystems of components to be tested for integration testing
    - Small adjustment to RTL to consider data path

Milestone 4 Work:

Monday, October 25th, 2021

- Met with Harrison and Taylor to start working on M4 (30 minutes)
    - Discussed how to split up components to implement and what each of us were going to start working on

Tuesday, October 26th, 2021

- Worked on creating components and test fixtures (1 hour)
    - Created both Instruction Memory and Data Memory components
- Met with Taylor Goldman to finish updating the design document (2 hours)
    - Adjusted the data path to include stack pointer register and additional control bits
    - Added truth tables to design document
    - Fixing integration testing and planning

Milestone 5 Work:

Saturday, October 30th, 2021

- Met with Taylor to keep working on M4 adjustments and start M5 (1 hour)
    - Worked on adjusting the truth tables and creating a table for the overall control unit

Sunday, October 31st, 2021

- Met with whole group to work on subsystems and begin test benches (1 hour)
    - We created the first couple subsystems and talked about how we were going to split up working on test benches

Monday, November 1st, 2021

- Met with whole group in class (0.5 hours)
    - Tried to fix the first subsystem to make sure it passed the tests and worked correctly
- Met with Taylor Goldman (45 minutes)
    - Continued work on the subsystem and adjusting the memory component

Tuesday, November 2nd, 2021

- Met with whole group in class (1 hour)
    - Worked on first and second subsystems to complete them
- Met with Taylor to work on following subsystems (1 hour)

    - Finished up third subsystem and continued working on test benches

Wednesday, November 3rd, 2021

- Met with whole group in class (2 hours)
    - Adjusted memory component and reorganized project to continue building on subsystems
    - Fixed git commit issues with files not being in the right place
- Met with Taylor to work on subsystems and test benches (2 hours)
    - Finished creating subsystems and continued working on test benches for the subsystems

Milestone 6 Work:

Sunday, November 7th, 2021

- Met with Taylor to finish what we didn't complete for M5 (3 hours)
    - I created the final data path and connected it all together so we could begin testing the final data path
    - I also adjusted some of our components to add missing I/O

Monday, November 8th, 2021

- Met with whole group in class (1 hour)
    - We finished the final data path and worked on test benches for the subsystems

Tuesday, November 9th, 2021

- Met with whole group in class (1 hour)
    - Worked on test benches to correct any failures
- Met with Taylor to work on more test benches and figure out errors (1 hour)
    - Met with Dr. Williamson to talk about the bugs in our first test bench
    - Also talked a bit about timing issues

Wednesday, November 10th, 2021

- Met with whole group in class (2 hours)
    - Worked on finishing test benches and added relprime machine code to data path for performance testing
- Met with Taylor to work on subsystems and test benches (2 hours)
    - Worked through all subsystem test benches to verify their outputs and also attempted to get final data path test to work

Thursday, November 11th, 2021

- Met with whole group after milestone meeting (1 hour)
    - Worked on breaking down the final datapath to help find issues with test benches
- Met with Taylor Goldman to finish working on datapath and testing (2 hours)
    - Finished new datapath with broken down parts

- Create testbench to correctly test the ACC_Control component
    - We think this is what is actually causing issues in the final datapath as it was improperly/not tested previously

Milestone 7/8 Work:

Friday, November 12th, 2021

- Met with Taylor to finish what we didn't complete (2 hours)
    - We worked on the tests for our control unit and final datapath
    - I worked on the final report and started to lay it out

Saturday, November 13th, 2021

- Met with Taylor to finish what we didn't complete (3 hours)
    - We continued work on the tests for our control unit and final data path and fixed git issues
    - I continued work on the final report by typing out the executive summary, introduction to our processor and lots of details about our project

Appendix C: Taylor's Design Process Journal

Taylor Goldman, WORK LOG

MILESTONE 1 WORK:

Sunday, October 3, 2021

- Created design document and started working on the layout, and created the GitHub team and cloned the repository (30 minutes)
- Met with team to start working on design document (2.5 hours)
    - Personally worked on the machine language formatting
    - I also worked on the section where we translate the assembly instructions to machine language

Monday, October 4, 2021

- Added to the design document by adding in the English description for each of our instructions we have created thus far (20 minutes)
- Created another instruction and added it to our table (10 minutes)

Tuesday, October 5, 2021

- Met with Luke Dawdy to finish up the design document (1.5 hours)
    - Fixed our machine language format and instruction sizes
    - Added instructions for input / output
    - Completed the common operations in assembly language
    - Completed machine language translations

MILESTONE 2 WORK:

Sunday, October 10, 2021

- Met with Luke Dawdy to make changes to M1 based off our first meeting (2 hours)
    - I added a recursive method to our example assembly language and wrote the machine code translation for it
    - We adjusted instructions to allow for proper implementation
    - I crossed out $fp and $sp for now, we may find a use for them later
    - I added assembly language description next to the machine code translation, so you know what each line of machine code corresponds to

Monday, October 11, 2021

- Worked on starting the RTL (45 minutes)
    - Added the needed sections for M2 onto our design document
    - I created a table for our RTL and split up the instructions based on similarity

Tuesday, October 12, 2021

- Met with Luke Dawdy to finish up the RTL chart and list of components (1.5 hours)
    - I made some corrections and finished up the RTL chart
    - I also added components to our list that we needed implemented for the RTL and specified: inputs, outputs, controls, component description, and necessary RTL symbols for each component.
    - Helped Luke with finishing up correcting our machine code translation for relprime from M1

MILESTONE 3 WORK:

Monday, October 18, 2021

- Met with Luke Dawdy to try to fix up M2 problems (1 hour)
    - Slightly adjusted RTL
    - Fixed the sign extension on some of the variables
    - Worked on creating the memory map for our design document

Tuesday, October 19, 2021

- Met with Luke Dawdy and Harrison Wight to go over M3 (1.5 hours)
    - Finished adding memory map to design document
    - Discussed how we are storing variables to memory
    - Started drawing datapath
- I created the schematic and symbol on Xilinx for the newPC = PC + 2 operation for our datapath (45 minutes)

Wednesday, October 20, 2021

- I met with my team during lab time to finish up M3 (1 hour)
    - I added the Xilinx project directory to the implementation folder
    - I also created the zero extender on Xilinx for our 11-bit instructions

MILESTONE 4 WORK:

Monday, October 25, 2021

- Met during class with Harrison and Luke to start working on M4 (30 minutes)

Tuesday, October 26, 2021

- Worked on creating test fixtures for the components I have made on Xilinx (2 hours)
    - PC Adder, Sign Extender, and Zero Extender
- Met with Luke Dawdy to finish up updates on the design document for M4 (1 hour)
    - I added truth tables for our combinational control units: ALUsrc, WrEn, PCSrc, reset, and MemWrite

MILESTONE 5 WORK:

Saturday, October 30, 2021

- Finished correcting my implementation Verilog code for the sign extender, zero extender, and pc adder, and created tests that work when simulated. (45 minutes)
- Met with Luke Dawdy to work on the truth table for our instructions and control bits (1 hour)

Sunday, October 31, 2021

- Met with Luke Dawdy and Harrison Wight to work on creating subsystems for our implementation for M5 (1 hour)
    - Created subsystem with pc adder and instruction memory
    - Created the next subsystem by adding the zero extender and sign extender to it

Monday, November 1, 2021

- Worked in class to try to get our first subsystem to work properly (30 minutes)
- Met with Luke Dawdy to keep working on changing the first subsystem to get it to properly work (45 minutes)

Tuesday, November 2, 2021

- Worked on finishing up the first and second subsystems, and created testbench programs for them (1 hour)

- Met with Luke Dawdy and finished up the third subsystem, as well as fixed problems for our previous subsystems. We also wrote the first testbench program for our first subsystem. (1 hour)
- Tried working on creating our new ALU (30 minutes)

Wednesday, November 3, 2021

- Worked with Luke Dawdy and Harrison Wight in class (2 hours)
    - Started fixing memory for the subsystem implementation
    - Finished 2nd subsystem testbench program
- Met with Luke Dawdy to finish our datapath and create more testbenches (2 hours)
    - Created the rest of the subsystems for our datapath
    - Created more testbench programs
    - Updated design document

MILESTONE 6 WORK:

Sunday, November 7, 2021

- Worked with Luke Dawdy to start finishing up what we didn't get done for M5 (2 hours)
    - I created a register verilog module so that we could implement the $sp and $acc registers into our final datapath
    - Helped create the final datapath

Monday, November 8, 2021

- Met in class to work on adjusting errors for final datapath (1 hour)
    - Finished final datapath
    - Started working on getting test benches to work properly

Tuesday, November 9, 2021

- Worked on subsystem testbenches in class (1 hour)
    - I also added to the design doc to prep for when we put in performance
- Met with Luke Dawdy to figure out some testbench errors (1.5 hours)
    - Also met with Dr. Williamson to work out bugs for our first subsystem testbench
- Working on second and starting third subsystem testbench (1 hour)

Wednesday, November 10, 2021

- Worked on finishing up tests and adding relprime code to our datapath in class (1.5 hours)
    - I looked over and adjusted the relprime machine code before adding it to our meminst.mif file so that it resembles the instructions in our processor's memory
- Met with Luke Dawdy to finish up the subsystems and work on final datapath testing (2 hours)
    - Got all of subsystem tests to work and run, with minor errors (timing)

- Fixed alu16b.v and the alu16b_tb.v files so that the alu works properly with minor timing issues
- Working through fixing the final datapath test, it is currently not running properly, and the waveform is blank

Thursday, November 11, 2021

- Worked after meeting to try to make our final datapath into more smaller subsystems so the testbenches work (1 hour)
- Added another subsystem with muxes and registers before our final datapath (1 hour)
- Met with Luke Dawdy to create our new final datapath and test it (2 hours)
  - Created new final datapath (still has an empty testbench)
  - Creating test for ACC_Control.v to see if that is our issue

MILESTONE 7 WORK:

Saturday, November 13, 2021

- Met with Luke Dawdy to resolve our git issues and start the presentation/report (3 hours)
  - I created our presentation and put everything we need into it (might add / change some minor details before presentation)
- Worked on getting ACC_Control_tb.v to actually have outputs in a waveform (1 hour)

MILESTONE 8 WORK:

Saturday, November 13, 2021

- Added my work log as well as finishing touches on design document to the final report (1 hour)

Sunday, November 14, 2021

- Met with our team to finish up the presentation and report, and finish up gathering performance data (1 hour)

Appendix D: Harrison's Design Process Journal

Harrison Wight, WORK LOG

MILESTONE 1 WORK:

Sunday, October 3, 2021

Met with team [2.5 hours]

- The team worked on creating the design document for our Accumulator Instruction Set Architecture because it will the easiest to implement. We worked on the design

document to get a general overview of our ISA's design. I wrote the registers available to the user, the syntax/semantics of instructions, and procedure call conventions.

Tuesday, October 5, 2021

- To allow for input and output, we added two registers ($in and $out) to get input and output values. We had to change our instruction types to 16 bits to fit within the project parameters. I added the relPrime example method and hexadecimal representation of reading and writing data to input and output port. [40 min]

MILESTONE 2 WORK:

Friday, October 8, 2021

- Worked on Lab 7, by setting up a block memory module and read through the resources page on the course website. I will be the team's "memory expert". [45 minutes]

Tuesday, October 12, 2021

- Created a list of generic component specifications needed to implement the RTL [20 minutes]
- Continued working on Lab 7 to demo on Wednesday, the 13th [1 hour]

MILESTONE 3 WORK:

Sunday, October 17, 2021

- Finished Lab 7: Created the schematic with the memory and control modules and tested to verify functionality [2 hours]

Tuesday, October 19, 2021

- Met with the Luke Dawdy and Taylor Goldman to work on the datapath and how we wanted to handle storing to memory [1.5 hours]

Wednesday, October 20, 2021

- Did a demo of lab 7 and modified the Memory Map to account for the fact that we can only use 10 bits in our address [2 hours]
    - Updated the memory map with Luke after the lab 7 demo
    - Added the Lab 7 to the git repository

MILESTONE 4 WORK:

Monday, October 25, 2021

- Met with Taylor and Luke to start on M4 (30 minutes)
    - Talked about how to split up components to implement and who was going to create which component

Tuesday, October 26, 2021

- Drew the schematic for 16 bit ALU and the test bench (1 hour)

MILESTONE 5 WORK:

Sunday, October 31, 2021 (Spooky)

- Met with Luke and Taylor to start the integration tests. [1 hour]

Monday, November 1, 2021

- Had to update lab 7 to our specific control needs [3.5 hours]

Wednesday, November 3, 2021

- Needed to update our ALU to work properly[2 hours]
- Started to put the Datapath fully together for the entire system and had to redo the blockmemory generator [1.5 hours]

MILESTONE 6 WORK:

Sunday, November 7, 2021

- Started the final report, and presentation [30 minutes]

Monday, November 8, 2021

- Met in class with Luke and Taylor to fix errors in the datapath[1 hour]

Tuesday, November 9, 2021

- Drew the majority of the datapath using draw.io on google, and had to edit the old datapath drawing a little [1.5 hours]
- Met with Luke and Taylor in class
    - Worked on the subsystem test bench errors [1 hour]

Wednesday, November 10, 2021

- Finished the datapath drawing[30 minutes]
- Worked on integration tests[30 minutes]
- Tried to get the final datapath working [1 hour]

Thursday, November 11, 2021

- Worked with team after the team meeting to get the final datapath working [1 hour]
- Continued working after the meeting to find the problem, but no luck [1 hour]

Friday, November 12, 2021

- Worked on getting the final datapath working [30 minutes]

MILESTONE 7/8 WORK:

Saturday, November 13, 2021

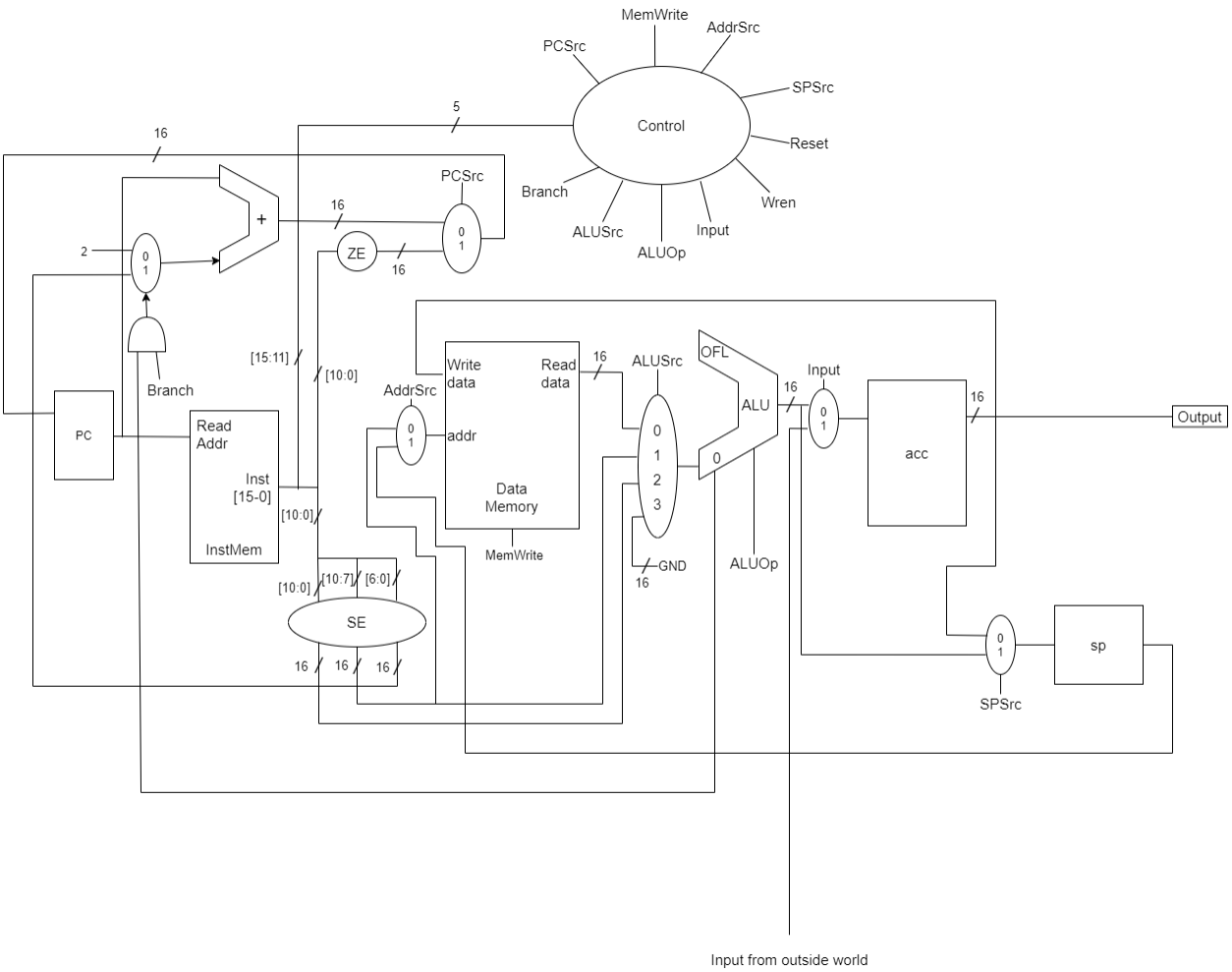
- Added some information to the final report [30 minutes]

Appendix E: RTL, Datapath and Truth Table

RTL:

Add / Sub / And / Or / Sl / Sr	**Addi / Andi / Ori**	**B-Type Instruction**	**J-Type Instruction**
newPC = PC + 2 PC = newPC Inst = Mem[PC] A = Reg[acc] B = SE(Mem[var]) Result = A op B Reg[acc] = Result	newPC = PC + 2 PC = newPC Inst = Mem[PC] A = Reg[acc] B = SE(Immediate) Result = A op B Reg[acc] = Result	newPC = PC + 2 PC = newPC Inst = Mem[PC] A = Reg[acc] B = SE(Immediate) Result = newPC + ZE(BranchAddr) If(A op B) PC = Result	newPC = PC + 2 PC = newPC Inst = Mem[PC] PC = ZE(BranchAddr)
Store	**Load**	**Getin / Return**	**Clear**
newPC = PC + 2 PC = newPC Inst = Mem[PC] A = Reg[acc] B = Mem[var] Mem[B] = A	newPC = PC + 2 PC = newPC Inst = Mem[PC] A = Reg[acc] B = Mem[var] Mem[A] = B	newPC = PC + 2 PC = newPC Inst = Mem[PC] **Getin:** Reg[acc] = Input **Return:** Output = Reg[acc]	newPC = PC + 2 PC = newPC Inst = Mem[PC] A = Reg[acc] B = ZE(0) Result = A && B Reg[acc] = Result

Datapath:



Control Truth Table

Opcode	Inst	Branch	PCSrc	MemWrite	ALUSrc	ALUOp	Input	Wren	RST	SPSrc	AddrSrc
00000	**add**	0	0	0	00	000	0	1	0	x	x
00001	**addi**	0	0	0	10	000	0	1	0	x	x
00010	**and**	0	0	0	00	010	0	1	0	x	x
00011	**andi**	0	0	0	10	010	0	1	0	x	x
00100	**beq**	1	0	0	01	001	0	0	0	x	x
00101	**bne**	1	0	0	01	001	0	0	0	x	x
00110	**j**	0	1	x	x	x	x	x	x	x	x
00111	**jal**	0	1	x	x	x	x	x	x	x	x
01000	**load**	0	0	1	x	x	0	0	0	1	0
01001	**or**	0	0	0	00	011	0	1	0	x	x
01010	**ori**	0	0	0	10	011	0	1	0	x	x
01011	**bgt**	1	0	0	01	001	0	0	0	x	x
01100	**blt**	1	0	0	01	001	0	0	0	x	x
01101	**sr**	0	0	0	10	101	0	1	0	x	x
01110	**sl**	0	0	0	10	100	0	1	0	x	x

01111	**store**	0	0	1	x	x	0	0	0	x	1
10000	**sub**	0	0	0	00	001	0	1	0	x	x
10001	**getin**	0	0	0	x	x	1	1	0	x	x
10010	**clear**	0	0	0	11	010	0	1	0	x	x
10011	**return**	x	x	x	x	x	x	x	x	x	x