Computer Architecture I

Reduced Instruction Set Processor

CSSE232

Team W
Michael Donaghy, Braedyn Edwards,
Emily Hart, Liam Hill, Logan Manthey

July 15, 2023

Rose-Hulman Institute of Technology
Department of Computer Science and Software Engineering

Abstract

This document outlines the processor designed in CSSE232: Computer Architecture by Michael, Braedyn, Emily, Liam, and Logan. This processor uses a reduced instruction set based on accumulators.

Table of Contents

1. Introduction

S.W.H.A.P.
Sid **W**e **H**ave **A** **P**roblem

1.1. High Level Summary

The SWHAP multi-cycle processor was developed by Braedyn Edwards, Liam Hill, Micheal Donaghy, Emily Hart, and Logan Manthey in CSSE232, Fall 2022. This processor will output the relative prime number to the input.

We developed our own Assembly Language, which has 16 registers available for use, 27 instructions, 8 instruction types, detailed Syntax and Semantics, and descriptive calling conventions. Additional Assembly Language details can be viewed in Section 2. We also developed an assembler, which was used to convert our assembly language to machine code quickly. Our RTL is comprised of 4 cycles, with testing and verification included, outlined in Section 3. Section 4 contains our list of components, with specifications, a description, diagram hardware implementation plan, and unit testing plan for each used in the SWHAP Processor.

Our 17 control bits allow us to control when values are read and written, to choose what data to use and where to store it, and to support conditional logic. We also include details on our testing process in Section 7, with Unit Testing, Integration Testing, Subsystem Testing, and System Testing ran on our processor.

Section 8 describes our Subsystems: PC, Memory, Register File, Branch and ALU. Finally our last sections describe the performance of the processor, and our special features: the Memory Mapped IO and our Assembler.

1.1.1. Instruction Set Architecture

This processor uses a multi-accumulator ISA with memory-mapped IO. This allows the programmer to switch easily between which accumulator is currently in use, through use of the swap command. This allows us to offer 27 instructions, with 8 instruction types.

1.1.2. Implementation

Our Assembly Language is detailed in section 2, and our Multi-cycle RTL is detailed in Section 3. We used Verilog in Quartus to create our components, subsystem, and final processor system.

1.1.3. Testing

Each instruction and component was unit tested, and subsystems underwent integration and subsystem testings. The final system went through our System Tests, and we aimed to follow Boundary Value Analysis to ensure we were testing for the right things. Section 4 details our testing plan and processes.

1.1.4. Final Results

Our final results can be seen in Section 9: Performance. Our processor can take in an input, and outputs the closest relatively prime number. Our Average CPI was 3.34, and our total time taken to run relPrime with the input 13b'0 was 20ms.

2. Assembly Language Specifications

2.1. High Level Description

This Reduced Instruction Set Processor uses an accumulator processor design with up to 4 accumulators. It has a dedicated current accumulator register to specify which accumulator to use along with a swap instruction which allows one to swap the current accumulator to one specified in the instruction.

2.2. Registers Available

Registers zero, sp, ra, a0-a3, and p0-p7 are available for the assembly language programmer to use. Register ca is reserved for the current accumulator to be stored. Registers sp and ra can't be edited directly, but their values can be changed through the use of instructions available. See table 2.1.

REGISTER	NAME	USE	SAVER
x0	zero	Zero	N.A
x1	sp	Stack Pointer	Callee
x2	ra	Return Address	Caller
x3-x6	a0-a3	Accumulators	Caller
x7-x8	p0-p1	Function Args/ Return Type	Caller
x9-x14	p2-p7	Func Args	Caller
x15	ca	Current Accum.	—

Table 2.1.: Register Name, Use, Calling Convention

- **Stack Pointer (sp):** The stack pointer is a callee saved register allowing one to increment the amount of data we want to store in our procedure
- **x1 - x14:** These are caller saved because we want to save these values on the stack before we call another procedure
- **Zero and ca:** Zero is hardwired to ground (creating a zero) and Current Accumulator is only accessed by the swap instruction with user input

2.3. Instructions

There are 8 machine language instructions: A, I, M, J, B, L, LI, C. See table 2.2 for the instruction formatting.

Table 2.2.: Instruction Format

		15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
Arithmetic	**A**	—							rs1				f2		Opcode		
Immediate	**I**	Immediate											f2		Opcode		
Memory	**M**	Immediate							rs1				f2		Opcode		
Jump	**J**	Immediate											f2		Opcode		
Branch	**B**	Immediate							rs1				f2		Opcode		
Logical	**L**	—							rs1				f2		Opcode		
Logical Immediate	**LI**	—			Immediate								f2		Opcode		
Control	**C**	–							rs1				f2		Opcode		

2.4. Syntax and Semantics

There are 27 base instructions, and 8 format types. See table 2.6 for more details and Verilog description names, and mnemonics.

Table 2.3.: A, I, M Type Syntax and Semantics

Instruction	Description	Syntax	Where	Usage
		A Type		
ADD	Adds the content of a source register rs1 and the current accumulator and stores the value on the current accumulator.	add rs1	rs1 is the register to add	add x0
SUB	Subtracts rs1 from the current accumulator register value and stores the result in the current accumulator.	sub rs1	rs1 is the register to sub	sub x0
BLASTOFF	Takes the value from the current accumulator and stores it in rs1	blastoff rs1	rs1 is the register to store the blasted off value	blastoff p0
BLASTON	Takes the value from rs1 and stores it to the current accumulator overriding any current value	blaston rs1	rs1 is that's value is taken and blasted onto the current accumulator	blaston p0
		I Type		
ADDI	Adds the value of the Immediate to the current accumulator value and stores the value to the current accumulator	addi imm	imm is the immediate to add to the current accumulator	addi 0x7
SLLI	Bitwise shifts the value of the current accumulator register by the immediate value and stores the value in the current accumulator	slli imm	imm is the immediate amount to shift the current accumulator left by	slli 0x1
LUI	Sets the current accumulator register value to the top half of the imme-diate (bits [15:8])	lui imm	imm is the immediate to which bits [15:8] are loaded into the current accumulator	lui 0xFF4D
ADDSP	Increments the stack pointer by the immediate value and stores the value into the stack pointer	addsp imm	imm is the value to increase the stack pointer by	addsp 2
		M Type		
LW	Loads the value of rs1 plus the immediate from memory and stores it into the current accumulator register	lw imm(rs1)	rs1 is the register used as the base memory address imm is the value added to rs1 as an offset	lw 4(x7)
SW	Stores the value of the current accumulator register into rs1 plus the immediate in memory	sw imm(rs1)	rs1 is the register used as the base memory address imm is the value added to rs1 as an offset	sw 4(x7)

Table 2.4.: J, B Type Syntax and Semantics

Instruction	Description	Syntax	Where	Usage
		J Type		
J	Jumps to the location in memory at the PC + the immediate denoted by the offset and the label	j imm(label)	label is the label which to jump to imm is the value added to the label as an offset	j 0 EXIT
JL	Jumps to the location in memory at the PC + the immediate denoted by the offset and the label and links the original value of the PC to the return address by storing it in the ra register	jl imm(label)	label is the label which to jump *imm* is the value added to the label as an offset	jl 0 EXIT
Jar	Jumps to the address stored in the ra register offset by the immediate	jar imm	imm is the value added to the label as an offset	jar 0
Jarl	Jumps to address stored in the ra register, offset by the immediate, and stores the jumped from location in ra	jarl imm	imm is the value added to the label as an offset	jarl 0
		B Type		
BEQ	If the current accumulator register and rs1 are equal, add the immediate to the PC	beq rs1, label	rs1 is the register to compare the current accumulator to label is the label which to branch to	beq x7 EXIT
BGE	If the current accumulator register is greater than or equal to rs1, add the immediate (label) to the PC	bge rs1, label	rs1 is the register to compare the current accumulator to label is the label which to branch to	bge x7 EXIT
BLT	If the current accumulator register is less than rs1, add the immediate (label) to the PC	blt rs1, label	rs1 is the register to compare the current accumulator to label is the label which to branch to	blt x7 EXIT

Table 2.5.: L, LI, C Type Syntax and Semantics

Instruction	Description	Syntax	Where	Usage
	L Type			
OR	Bitwise ors the current accumulator register with rs1 and stores the resulting value to the current accumulator	or rs1	rs1 is the register to or the current accumulator to	or x7
AND	Bitwise ands the current accumulator register with rs1 and stores the resulting value to the current accumulator	and rs1	rs1 is the register to and the current accumulator to	and x7
XOR	Bitwise xors the current accumulator register with rs1 and stores the resulting value to the current accumulator	xor rs1	rs1 is the register to xor the current accumulator to	xor x7
SLT	If the current accumulator register is less than the value of rs1, set the current accumulator to 1, otherwise set it to 0	slt rs1	rs1 is the register to compare the current accumulator to	slt x7
	LI Type			
ORI	Bitwise ors the current accumulator register with the immediate and stores the value in the current accumulator	ori imm	imm is the immediate to or the current accumulator to	ori 4
ANDI	Bitwise ands the current accumulator register with the immediate and stores the value in the current accumulator	andi imm	imm is the immediate to and the current accumulator to	andi 4
XORI	Bitwise xors the current accumulator register with the immediate and stores the value in the current accumulator	xori imm	imm is the immediate to xor the current accumulator to	xori 4
SLTI	If the immediate is less than the value of the current accumulator register, set the current accumulator to 1, otherwise set it to 0	slti imm	imm is the immediate to compare the current accumulator to	slti 4
	C Type			
SWAP	Changes the current accumlator pointer to the address of rs1	swap rs1	rs1 is the accumulator to swap to	swap a2
NOP	No Operation used as a delay and default instruction loaded	nop		nop

Table 2.6.: Base Instructions

MNEMONIC	FMT	NAME	VERILOG DESCRIPTION
add	A	Add	R[ca] = R[rs1] + R[ca]
addi	I	Add Immediate	R[ca] = imm + R[ca]
addsp	I	Add Stack Pointer	sp = R[sp] + imm
and	L	AND	R[ca] = R[rs1] & R[ca]
andi	LI	AND Immediate	R[ca] = imm & R[ca]
beq	B	Branch Equal	if(R[ca] == R[rs1]) PC=PC+imm
bge	B	Branch Greater Than Equal	if(R[ca] R[rs1]) PC=PC+imm
blt	B	Branch Less Than	if(R[ca] < R[rs1]) PC=PC+imm
jar	J	Jump and Return	PC = Reg[ra] + imm
j	J	Jump	PC = PC + imm + label
jarl	J	Jump and Return Immediate	PC = Reg[ra] + imm Reg[ra] = oldPc
jl	J	Jump Immediate	Reg[ra] = PC PC = PC + imm + label
lui	I	Load Upper Immediate	ca = imm[15:08]
lw	M	Load Word	ca = MEM[rs1] + imm(6:0)
nop	C	No Operation	N/A
or	L	Or	ca = ca | R[s1]
ori	LI	OR with Immediate	ca = ca | imm
blastOff	A	Blast Off	R[rs1] = ca
blastOn	A	Blast On	ca = R[rs1]
slli	I	Shift Left Immediate	ca = ca « imm
slt	L	Set Less Than	ca = (R[ca] < R[rs1]) ? 1 : 0
slti	LI	Set Less Than Immediate	if(imm< R[rs1]) ca = 1
sub	A	Subtract	ca = ca - R[rs1]
sw	M	Store Word	MEM[rs1] + imm(6:0) = ca
swap	C	Swap Current Accumulator	ca = *R[rs1]
xor	L	XOR	ca = R[rs1] ĉa
xori	LI	XOR Immediate	ca = R[rs1] îmm

2.5. Calling Conventions

For this instruction set, the following calling conventions must be observed:

- The current accumulator is caller saved
- The return address is caller saved
- The stack pointer is callee saved/returned to where it was found
- All function arguments/return values are caller saved
  - When calling a function, registers x7-x14 (p0-p7) are reserved for function arguments
  - When returning from a function call, registers x7-x8 (p0-p1) are reserved for return values

To be **caller saved**, it means that the calling function must save the register value on stack before calling the other function if it expects to have access to that same value again.

To be **callee saved**, it means that the called function (i.e. every function) must save those registers' values on the stack before using them because the calling function expects them to be unchanged. Below is a small assembly program to demonstrate calling conventions.

```
int
proc(int num){
      num = num + 2;
      proc2(num);
      return num;
}
```

```
proc:
    swap  a0            # currAccum = a0
    blastOn  p0         # a0 = a1
    addi 2              # a0 = a0 + 2
    addSp -4            # move sp down 4
    sw 0(sp)            # store num on stack
    blastOn ra          # a0 = ra
    sw 2(sp)            # store ra on stack
    jl 0(proc2)         # jump and return from proc2
    nop

    lw 0(sp)            # restore num from stack
    blastOff p0         # set return value to num
    lw 2(sp)            # restore ra from stack
    blastOff ra         # set return address to original value

    addSp 4             # move sp back up 4
    jar 0               # return to original caller
    nop

```

2.6. Translating Assembly Language into Machine Language

- **Opcode:** For all types, the opcode of the instruction is stored in inst[2:0]. Instructions of the same format share the same opcode since they can be differentiated by their funct2s.
- **Funct2:** For all types, the funct2 of the instruction is stored in inst[4:3]. The funct2, in combination with the opcode, allows us to distinguish between which operation is being used since the opcode doesn't give us that level of specificity.
- **RS1:** For the A, M, B, L, and C types, rs1 represents the register used in the instruction and is stored at inst[8:5].
- **Immediate:** For the I and J types, the immediate is stored at inst[15:5]. For the M and B types, the immediate is stored at inst[15:9]. For the LI type, the immediate is stored at inst[12:5].
- **Unused Bits:** For the A, L, and C types, inst[15:9] represent unused bits. For the LI type, inst[15:13] represent unused bits.

Table 2.7.: Opcodes

MNEMONIC	FMT	OPCODE	FUNCT2
add	A	001	00
sub	A	001	01
blastOff	A	001	10
blastOn	A	001	11
addi	I	010	00
slli	I	010	01
lui	I	010	10
addsp	I	010	11
lw	M	011	00
sw	M	011	01
jar	J	100	00
j	J	100	01
jl	J	100	10
jarl	J	100	11
beq	B	101	00
bge	B	101	01
blt	B	101	10
or	L	110	00
and	L	110	01
xor	L	110	10
slt	L	110	11
ori	LI	111	00
andi	LI	111	01
oxri	LI	111	10
slti	LI	111	11
swap	C	000	00
nop	C	000	01

2.7. Assembly Translations

Example assembly language program translations and fragments demonstrating that our instruction set can find relative primes and perform other common operations.

2.7.1. RelPrime

Assembly program to find relative primes (relprime).

```
int
relPrime(int n)
{
    int m;
    m = 2;

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

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;
}
```

```
relPrime:
    swap a0          # currAccum = a0
    blastOn p0       # a0 = n

    addsp -4         # move sp down 4
    sw 0(sp)         # store n on stack
    blastOn zero     # clear a0
    addi 2           # a0 = 2
    blastOff p1      # p1 = a0
    sw 2(sp)         # store p1 on stack

WHILE:
    jl 0 GCD         # call GCD
    nop              # Jump delay slot

    swap a1          # currAccum = a1
    blastOn p0       # a1 = p0
    swap a2          # currAccum = a2
    blastOn zero     # clear a2
    addi 1           # a2 = 1

    beq a1 DONE      # if (a1 == a2) goto DONE
    nop              # Branch delay slot

    swap a0          # currAccum = a0
    lw 2(sp)         # currAccum = stored M on stack
    addi 1           # currAccum = currAccum + 1
    blastOff p1      # p1 = currAccum
    sw 2(sp)         # store currAccum (M) on stack
    lw 0(sp)         # load n off of stack into currAccum
    blastOff p0      # p0 = currAccum
    j 0 WHILE        # continue loop
    nop              # Jump delay slot

DONE:
    swap a0          # currAccum = a0
    lw 2(sp)         # currAccum = stored M on stack
    blastOff p0      # p0 = currAccum
    addsp 4          # reset sp

    sw 8(zero)       # put result in IO Output
    jar 0            # jump back to caller
    nop              # Jump delay slot

GCD:
    swap a2          # Swap currAccum to a2
    blastOn p1       # blastOn p1 onto a2
```

```
48      swap a1          # Swap currAccum to a1
49      blastOn p0       # blastOn p0 onto a1
50
51      beq zero DONEB   # if (a == 0) goto DONEB
52      nop              # Branch delay slot
53
54  LOOP:
55      swap a2          # Swap currAccum to a2
56      beq zero DONEA   # if(a2 == 0) goto DONEA (b == 0)
57      nop              # Branch delay slot
58
59      bge a1 SUBA      # if(a2 >= a1) goto SUBA (if statement)
60      nop              # Branch delay slot
61
62      swap a1          # Swap currAccum to a1
63      sub a2           # a1 = a1 - a2 (a = a - b)
64      j 0 LOOP         # continue loop
65      nop              # Jump delay slot
66
67  SUBA:
68      sub a1           # a1 = a1 - a2 (a = a - b)
69      j 0 LOOP         # Jump to LOOP
70      nop              # Jump delay slot
71
72  DONEA:
73      swap a1          # Swap currAccum to a1
74      j 0 END          # Jump to END
75      nop              # Jump delay slot
76
77  DONEB:
78      swap a2          # Swap currAccum to a2
79      j 0 END          # Jump to END
80      nop              # Jump delay slot
81
82  END:
83      blastOff p0      # p0 = currAccum
84      jar 0            # return to caller
85      nop              # Jump delay slot
86
```

2.7.2. SumArray

Assembly program to demonstrate array iteration, loops, and summation.

```c
int
sumArr(int len, int* arr)
{
    int total = 0;
    for ( int i = 0; i < len; i++ ) {
        total = total + arr[i];
    }
    return total;
}
```

```
SumArr:
    swap a0          # currAccum = a0
    blastOn p0          # a0 = p0

    swap a1          # currAccum = a1
    blastOn zero        # a1 = 0
    swap a0          # currAccum = a0

LOOP:
    beq  zero EXIT # if ( a0 == 0 ) goto EXIT
    nop

    swap a0          # currAccum = a0
    blastOn p0       # a0 = p0
    addi -1          # a0 = a0 - 1
    blastOff  p0     # p0 = a0
    slli 1           # a0 = a0 * 2
    add  p1          # a0 = a0 + p1

    lw   0(a0)       # load currAccum with value at a0[0]
    swap a1          # currAccum = a1
    add  a0          # a1 = a1 + a0;
    jl   0 LOOP      # continue loop
    nop

EXIT:
    blastOn p0       # a1 = p0
    jar 0            # return to caller
    nop
```

2.7.3. SumArray (Recursive)

Assembly program to demonstrate recursion and conditionals using a recursive implementation of the previous program.

```
int
sumArrRec(int len, int* arr)
{
    if ( len == 0 ) { return 0; }

    return arr[0] + sumArrRec( len - 1, (arr + 1) );

}
```

```
SumArrRec:
    swap a0          # currAccum = a0
    blastOn zero     # a0 = 0
    swap a1          # currAccum = a1
    blastOn p0       # a1 = p0

    beq zero EXIT    # if ( a1 == 0 ) goto EXIT
    nop
    addi -1          # a1 = a1 - 1
    blastOff p0      # p0 = a1
    addsp -4         # sp = sp - 4
    blastOn p1       # a1 = p1
    sw 0(sp)         # store p1 on stack

    addi 2           # a1 = a1 + 2
    blastOff p1      # p1 = a1
    swap a0          # currAccum = a0
    blastOn ra       # a0 = ra
    sw 2(sp)         # store ra on stack
    jl 0, SumArrRec # recursively call SumArrRec
    nop

    blastOn p0       # a0 = p0
    swap a1          # currAccum = a1
    lw 0(sp)         # address of arr is restored from stack
    lw 0(a1)         # a1 = arr from stack at address

    swap a0          # currAccum = a0
    add a1           # a0 = a0 + a1
    swap a1          # currAccum = a1
    lw 2(sp)         # address of ra is restored from the stack
    blastOff ra      # ra = ra from stack
```

```
33
34 EXIT:
35     swap a0          # currAccum = a0
36     blastOff p0      # p0 = a0
37     jar 0            # return to caller
38     nop
39
```

2.7.4. If

Assembly program to demonstrate an if statment

```
int
addIf1(int num)
{
    if ( num=1 ) { num+=1 }

    return num;

}
```

```
AddIf1:
    swap  a0            # currAccum = a0
    blastOn  zero       # a0 = 0
    swap  a1            # currAccum = a1
    blastOn  p0         # a1 = p0
    addi -1             # a1 -= 1
    beq   zero ADD      # if ( a1 == 0 ) goto EXIT
    nop

EXIT:
    swap  a0            # currAccum = a0
    blastOff   p0       # p0 = a0
    jar  0              # return to caller
    nop
ADD:
    addi 2
    blastOff   p0       # p0 = a1
    blastOn zero
    beq zero EXIT
    nop

```

2.7.5. While Loop

Assembly program to demonstrate a while loop

```
int
WhileExample()
{
    int num = 0;
    while(num < 10) {
        num++;
    }

    return num;
}
```

```
WhileExample:
    swap  a0            # currAccum = a0
    blastOn  zero       # a0 = 0
    swap  a1            # currAccum = a1
    blastOn  zero       # a1 = p0
    addi  10            # a1 = 10
    swap  a0

WHILE:
    bge   a1 DONE       # if a0 > a1 (num > 10) goto DONE
    nop
    addi  1             # a0 += 1
    j     0 WHILE       # loop again
    nop

DONE:
    blastOff   p0       # p0 = a1
    jar   0             # return to caller
    nop
```

2.8. Machine Language Translations

Machine language translations of our assembly programs (relprime and your fragments).

2.8.1. RelPrime

Machine language translation of relprime.

```
0x0000    0000000001100000    //swap a0 # currAccum = a0 from line2
0x0002    0000000011111001    //blastOn p0 # a0 = n from line3
0x0004    1111111110011010    //addsp -4 # move sp down 4 from line4
0x0006    0000000000101011    //sw 0(sp) # store n on stack from line5
0x0008    0000000000011001    //blastOn zero # clear a0 from line6
0x000a    0000000001000010    //addi 2 # a0 = 2 from line7
0x000c    0000000100010001    //blastOff p1 # p1 = a0 from line8
0x000e    0000010000101011    //sw 2(sp) # store p1 on stack from line9
0x0010    0000011101010100    //jl 0 GCD
0x0012    0000000000001000    //nop
0x0014    0000000000001000    //nop # Jump delay slot from line11
0x0016    0000000010000000    //swap a1 # currAccum = a1 from line12
0x0018    0000000011111001    //blastOn p0 # a1 = p0 from line13
0x001a    0000000010100000    //swap a2 # currAccum = a2 from line14
0x001c    0000000000011001    //blastOn zero # clear a2 from line15
0x001e    0000000000100010    //addi 1 # a2 = 1 from line16
0x0020    0011010010000101    //beq a1 DONE
0x0022    0000000000001000    //nop
0x0024    0000000000001000    //nop # Branch delay slot from line17
0x0026    0000000001100000    //swap a0 # currAccum = a0 from line18
0x0028    0000010000100011    //lw 2(sp) # currAccum = stored M on stack from line19
0x002a    0000000000100010    //addi 1 # currAccum = currAccum + 1 from line20
0x002c    0000000100010001    //blastOff p1 # p1 = currAccum from line21
0x002e    0000010000101011    //sw 2(sp) # store currAccum (M) on stack from line22
0x0030    0000000000100011    //lw 0(sp) # load n off of stack into currAccum from l
0x0032    0000000011110001    //blastOff p0 # p0 = currAccum from line24
0x0034    1111101110001100    //j 0 WHILE
0x0036    0000000000001000    //nop
0x0038    0000000000001000    //nop # Jump delay slot from line25
0x003a    0000000001100000    //swap a0 # currAccum = a0 from line27
0x003c    0000010000100011    //lw 2(sp) # currAccum = stored M on stack from line28
0x003e    0000000011110001    //blastOff p0 # p0 = currAccum from line29
0x0040    0000000010011010    //addsp 4 # reset sp from line30
0x0042    0001000000001011    //sw 8(zero) # put result in IO Output from line31
0x0044    0000000000000100    //jar 0
0x0046    0000000000001000    //nop
0x0048    0000000000001000    //nop # Jump delay slot from line32
0x004a    0000000010100000    //swap a2 # Swap currAccum to a2 from line34
0x004c    0000000100011001    //blastOn p1 # blastOn p1 onto a2 from line35
0x004e    0000000010000000    //swap a1 # Swap currAccum to a1 from line36
0x0050    0000000011111001    //blastOn p0 # blastOn p0 onto a1 from line37
0x0052    0101110000000101    //beq zero DONEB
0x0054    0000000000001000    //nop
0x0056    0000000000001000    //nop # Branch delay slot from line38
```

```
0x0058    0000000010100000    //swap a2 # Swap currAccum to a2 from line40
0x005a    0011110000000101    //beq zero DONEA
0x005c    0000000000001000    //nop
0x005e    0000000000001000    //nop # Branch delay slot from line41
0x0060    0010000010000101    //beq a1 SUBA
0x0062    0000000000001000    //nop
0x0064    0000000000001000    //nop # Branch delay slot from line42
0x0066    0000000010000000    //swap a1 # Swap currAccum to a1 from line43
0x0068    0000000010101001    //sub a2 # a1 = a1 - a2 (a = a - b) from line44
0x006a    1111110111001100    //j 0 LOOP
0x006c    0000000000001000    //nop
0x006e    0000000000001000    //nop # Jump delay slot from line45
0x0070    0000000010001001    //sub a1 # a1 = a1 - a2 (a = a - b) from line47
0x0072    1111110011001100    //j 0 LOOP
0x0074    0000000000001000    //nop
0x0076    0000000000001000    //nop # Jump delay slot from line48
0x0078    0000000010000000    //swap a1 # Swap currAccum to a1 from line50
0x007a    0000000111001100    //j 0 END
0x007c    0000000000001000    //nop
0x007e    0000000000001000    //nop # Jump delay slot from line51
0x0080    0000000010100000    //swap a2 # Swap currAccum to a2 from line53
0x0082    0000000011001100    //j 0 END
0x0084    0000000000001000    //nop
0x0086    0000000000001000    //nop # Jump delay slot from line54
0x0088    0000000011110001    //blastOff p0 # p0 = currAccum from line56
0x008a    0000000000000100    //jar 0
0x008c    0000000000001000    //nop
0x008e    0000000000001000    //nop # Jump delay slot from line57
```

2.8.2. SumArray

Machine language translation of sumArray.

```
0x0000    0000000001100000    //swap a0 # currAccum = a0 from line2
0x0002    0000000011111001    //blastOn p0 # a0 = p0 from line3
0x0004    0000000010000000    //swap a1 # currAccum = a1 from line4
0x0006    0000000000011001    //blastOn zero # a1 = 0 from line5
0x0008    0000000001100000    //swap a0 # currAccum = a0 from line6
0x000a    0011110000000101    //beq zero EXIT
0x000c    0000000000001000    //nop
0x000e    0000000000001000    //nop from line8
0x0010    0000000001100000    //swap a0 # currAccum = a0 from line9
0x0012    0000000011111001    //blastOn p0 # a0 = p0 from line10
0x0014    1111111111100010    //addi -1 # a0 = a0 - 1 from line11
0x0016    0000000011110001    //blastOff p0 # p0 = a0 from line12
0x0018    0000000000101010    //slli 1 # a0 = a0 * 2 from line13
0x001a    0000000100000001    //add p1 # a0 = a0 + p1 from line14
0x001c    0000000001100011    //lw 0(a0) # load currAccum with value at a0[0] from l
0x001e    0000000010000000    //swap a1 # currAccum = a1 from line16
0x0020    0000000001100001    //add a0 # a1 = a1 + a0; from line17
0x0022    1111110100010100    //jl 0 LOOP
0x0024    0000000000001000    //nop
```

```
0x0026    0000000000001000    //nop from line18
0x0028    0000000011111001    //blastOn p0 # a1 = p0 from line20
0x002a    0000000000000100    //jar 0
0x002c    0000000000001000    //nop
0x002e    0000000000001000    //nop from line21
```

2.8.3. SumArray (Recursive)

Machine language translation of sumArrayRec.

```
0x0000    0000000001100000    //swap a0 # currAccum = a0 from line2
0x0002    0000000000011001    //blastOn zero # a0 = 0 from line3
0x0004    0000000010000000    //swap a1 # currAccum = a1 from line4
0x0006    0000000011111001    //blastOn p0 # a1 = p0 from line5
0x0008    0110010000000101    //beq zero EXIT
0x000a    0000000000001000    //nop
0x000c    0000000000001000    //nop from line6
0x000e    1111111111100010    //addi -1 # a1 = a1 - 1 from line7
0x0010    0000000011110001    //blastOff p0 # p0 = a1 from line8
0x0012    1111111110011010    //addsp -4 # sp = sp - 4 from line9
0x0014    0000000100011001    //blastOn p1 # a1 = p1 from line10
0x0016    0000000000101011    //sw 0(sp) # store p1 on stack from line11
0x0018    0000000001000010    //addi 2 # a1 = a1 + 2 from line12
0x001a    0000000100010001    //blastOff p1 # p1 = a1 from line13
0x001c    0000000001100000    //swap a0 # currAccum = a0 from line14
0x001e    0000000001011001    //blastOn ra # a0 = ra from line15
0x0020    0000010000101011    //sw 2(sp) # store ra on stack from line16
0x0022    1111101111010100    //jl 0, SumArrRec
0x0024    0000000000001000    //nop
0x0026    0000000000001000    //nop from line17
0x0028    0000000011111001    //blastOn p0 # a0 = p0 from line18
0x002a    0000000010000000    //swap a1 # currAccum = a1 from line19
0x002c    0000000000100011    //lw 0(sp) # address of arr is restored from stack fro
0x002e    0000000010000011    //lw 0(a1) # a1 = arr from stack at address from line2
0x0030    0000000001100000    //swap a0 # currAccum = a0 from line22
0x0032    0000000010000001    //add a1 # a0 = a0 + a1 from line23
0x0034    0000000010000000    //swap a1 # currAccum = a1 from line24
0x0036    0000010000100011    //lw 2(sp) # address of ra is restored from the stack
0x0038    0000000001010001    //blastOff ra # ra = ra from stack from line26
0x003a    0000000001100000    //swap a0 # currAccum = a0 from line28
0x003c    0000000011110001    //blastOff p0 # p0 = a0 from line29
0x003e    0000000000000100    //jar 0
0x0040    0000000000001000    //nop
0x0042    0000000000001000    //nop from line30
```

2.8.4. If

Machine language translation of If.

```
0x0000    0000000001100000    //swap a0 # currAccum = a0 from line2
0x0002    0000000000011001    //blastOn zero # a0 = 0 from line3
0x0004    0000000010000000    //swap a1 # currAccum = a1 from line4
0x0006    0000000011111001    //blastOn p0 # a1 = p0 from line5
0x0008    1111111111100010    //addi -1 # a1 -= 1 from line6
0x000a    0010000000000101    //beq zero ADD
0x000c    0000000000001000    //nop
0x000e    0000000000001000    //nop from line7
0x0010    0000000001100000    //swap a0 # currAccum = a0 from line9
0x0012    0000000011110001    //blastOff p0 # p0 = a0 from line10
0x0014    0000000000000100    //jar 0
0x0016    0000000000001000    //nop
0x0018    0000000000001000    //nop from line11
0x001a    0000000001000010    //addi 2 from line13
0x001c    0000000011110001    //blastOff p0 # p0 = a1 from line14
0x001e    0000000000011001    //blastOn zero from line15
0x0020    1111000000000101     //beq zero EXIT
0x0022    0000000000001000    //nop
0x0024    0000000000001000    //nop from line16
```

2.8.5. While Loop

Machine language translation of While Loop.

```
0x0000    0000000001100000    //swap a0 # currAccum = a0 from line2
0x0002    0000000000011001    //blastOn zero # a0 = 0 from line3
0x0004    0000000010000000    //swap a1 # currAccum = a1 from line4
0x0006    0000000000011001    //blastOn zero # a1 = p0 from line5
0x0008    0000000101000010    //addi 10 # a1 = 10 from line6
0x000a    0000000001100000    //swap a0 from line7
0x000c    0001110010000101    //beq a1 DONE
0x000e    0000000000001000    //nop
0x0010    0000000000001000    //nop from line9
0x0012    0000000000100010    //addi 1 # a0 += 1 from line10
0x0014    1111111100001100    //j 0 WHILE
0x0016    0000000000001000    //nop
0x0018    0000000000001000    //nop from line11
0x001a    0000000011110001    //blastOff p0 # p0 = a1 from line13
0x001c    0000000000000100    //jar 0
0x001e    0000000000001000    //nop
0x0020    0000000000001000    //nop from line14

```

3. Register Transfer Language

3.1. Multi-Cycle RTL

The Multi-cycle RTL was created and designed off of the Single-Cycle RTL to make the system faster and more efficient, and can be seen below.

All instructions go through the same first two cycles: *Instruction Fetch* and *Instruction Decode/Register Fetch*. In the *Instruction Fetch* cycle, the instruction is fetched from memory based on where the PC is currently, and then the PC is incremented by two since our instructions are all 2 Bytes. The instruction is stored in a register named IR. In the *Instruction Decode/Register Fetch* cycle, register A is set to the value of the Current Accumulator, register B is set to the value of the register denoted by bits 8:5 of the instruction, newPC is set to the current PC + immGen to allow for any quick jumps/branches, and the current Stack Pointer is stored in a register aptly denoted SP. Even though each instruction does not use all of these registers, it is wise to set them all up in this cycle rather than adding more cycles to do them later for specific instructions. All immGen values in this RTL are sign extended to 16 bits using an immGen generator.

In the Execution and Memory Access/Completion cycles, individual instructions/instruction groups have unique RTL actions. For instance, all add/sub/and/or/xor operations use the ALU to perform an operation between A and B and the result is stored in the current accumulator. All memory reference instructions (sw and lw) add A and the immGen together and stores the value in the ALUOut Register. If it is a load, the value in memory at ALUOut is stored into the current accumulator; if it is a store, the value in the current accumulator is store in memory. For the push operation, the value of register B is store in the current accumulator. For the pop instruction, the register passed into the instruction is given the value of the current accumulator.

For branch operations, if the conditional is true, the PC obtains the value in ALUOut. For jump instructions, Jar and Jari store save the current location in the return address register, and then the PC is set to the value in ALUOut. For all immGen operations, the RTL is the same; however the B register is replaced with the immGen. For the slt operation, if the current accumulator is less than the passed in operand, the current accumulator is set to 1; otherwise, it is set to 0. For lui, the current accumulator is set to the top-most 8 bits of the immGen passed in. For the swap operation, the current accumulator is pointed to the same register as the operand (B). Finally, for the addsp operation, the ALUOut register is set to the stack pointer plus the provided immGen, which is then stored in the stack pointer register.

Table 3.1.: Multi-Cycle RTL Part 1

StepName	Action for branches	Action for Jumps	Action for addi / andi / ori/xori	Action for slt
Instruction Fetch	IR <= Memory[PC] oldPC <= PC PC <= PC + 2			
Instruction Decode / Register Fetch	A <= Reg[CA] B <= Reg[IR[8:5]] ALUOut <= oldPC + immGen(IR) Sp <= Reg[SP] RA <= Reg[RA]			

Execution, address comp, branch/ jump completion	if((A==B) |
	(A>=B) | (A<B))
	PC <= ALUOut
		J : PC <= ALUOut
		JL : Reg[RA] <= PC
		PC <= ALUOut
		Jar & Jarl: PC <= RA + immGen(IR)
			ALUOut <= A OP immGen(IR)	if(A<B)
				Reg[CA] <= 1
				else
				Reg[CA] <= 0

Memory Access or A-Type completion		Jarl: PC <= oldPC		

Note: immGen represents a combinational logic component that takes in the instruction, locates the immediate, and outputs a sign extended 16 bit immediate if it was not already.

Table 3.2.: Multi-Cycle RTL Part 2

StepName	Action for Add/Sub/And/Or/Xor	Action for Memory Reference Instructions	Action for BlastOn	Action for BlastOff
Instruction Fetch	IR <= Memory[PC] oldPC <= PC PC <= PC + 2			
Instruction Decode / Register Fetch	A <= Reg[CA] B <= Reg[IR[8:5]] ALUOut <= oldPC + immGen(IR) Sp <= Reg[SP] RA <= Reg[RA]			
Execution, address comp, branch/ jump completion	ALUOut <= A op B	ALUOut <= A + immGen(IR)	Reg[CA] <= B	Reg[IR[8:5]] <= A
Memory Access or A-Type completion	Reg[CA] = ALUOut	Load: Reg[CA] <= Memory[ALUOut] or Store: Memory[ALUOut] <= A		

Table 3.3.: Multi-Cycle RTL Part 3

StepName	Action for slti	Action for lui	Action for swap	Action for addsp
Instruction Fetch	IR <= Memory[PC] PC <= PC + 2			
Instruction Decode / Register Fetch	A <= Reg[CA] B <= Reg[IR[8:5]] ALUOut <= oldPC + immGen(IR) Sp <= Reg[SP] RA <= Reg[RA]			

Execution, address comp, branch/ jump completion	if (a < immGen(IR))
	Reg[CA] <= 1
	else
	Reg[CA] <= 0
		Reg[CA] <= immGen(IR)	CA <= B	ALUOut <= SP + immGen(IR)

Memory Access or A-Type completion				Reg[SP] <= ALUOut

Note: No Op was not included because because it has no actions that happen besides the base 2. Instruction Fetch and Instruction Decode

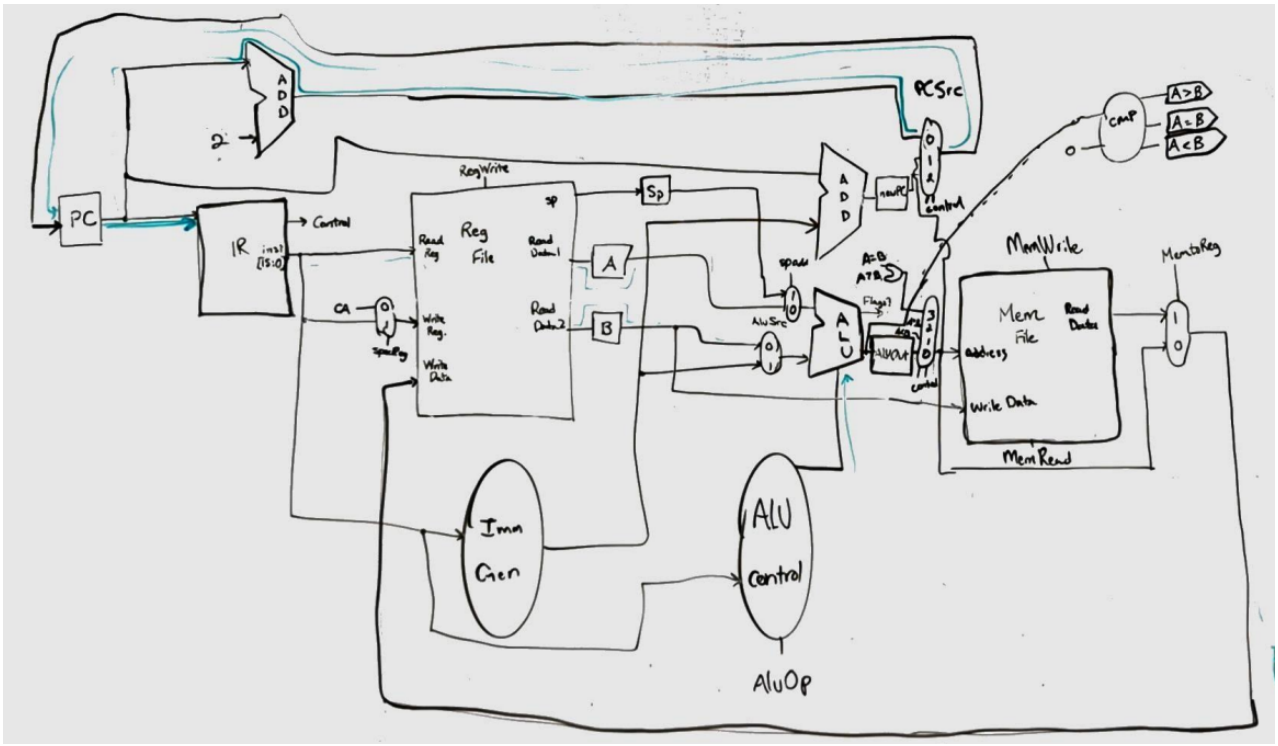


Figure 3.1.: Rough Draft Data Path

3.2. RTL Verification

To verify our RTL, we went ahead and drafted a rough draft of our datapath based on the textbook example specified (Figure 4.19). Before we began testing individual instructions, we edited the register file to only read in one register, since the first argument for each of instructions would be implicit. Despite this, we still included a write register input since we need the flexibility to write to other registers, such as our stack pointer as well as any function argument registers for our addsp and pop instructions. With the basic implementation complete, we physically began to draw out the RTL instructions step-by-step, taking special care to note any additional registers we need to store our data in between the multiple actions our multi-cycle implementation takes. The rough draft of the of the data path can be seen in figure 3.1. Once our data path is complete we will be writing RTL tests in code form by having starting values and predicted end values. We will test that these match at the end after the RTL actions are preformed manually.

3.3. RTL Tests

RTL tests will show the relevant registers before and after instructions are ran and at each step allowing one to compare the actual to expected values which will determine if the test is successful.

3.3.1. Add/Sub/And/Or/Xor Tests

add/sub/and/or/xor (sub p0):

```
PC = 0x0080
CA = 0x0003
a0 = 0x0040
p0 = 0x0008
sp = 0xFFFE

Expected: a0 = 0x0038

Sub subtracts the value of the current accumulator by the value in the specified
register.

IR <= Memory[PC]  #IR = XXXX XXX0 1110 1001
oldPC <= PC       #oldPC = 0x0080
PC <= PC + 2      #PC = 0x0082

A <= Reg[CA]      #A = 0x0040
B <= Reg[IR[8:5]  #B = 0x0008

ALUOut <= oldPC + immGen(IR)  #ALUOut = 0x0080
Sp = Reg[sp] <=   #Sp = 0xFFFE

ALUOUT <= A op B  #op = sub
                  #ALUOut = 0x0038

Reg[CA] <= ALUOut #a0 = 0x0038
```

3.3.2. Memory Reference Tests

lw/sw (lw 0(sp)):

```
PC = 0x0080
CA = 0x0003
a0 = 0xFEDC
sp = 0xFF00
Memory[sp] = 0x1234

Expected: a0 = 0x1234

LW puts the value stored in memory at the specified address + offset into the
current accumulator.

IR <= Memory[PC]  #IR = 0000 0000 0010 0011
oldPC <= PC       #oldPC = 0x0080
PC <= PC + 2      #PC = 0x0082

A <= Reg[CA]      #A = 0xFEDC
B <= Reg[IR[8:5]  #B = 0xFF00

ALUOut <= oldPC + immGen(IR)  #ALUOut = 0x0080
Sp = Reg[sp] <=   #Sp = 0xFF00

ALUOUT <= B + immGen(IR)  #ALUOut = 0xFF00

Reg[CA] <= Memory[ALUOut] #a0 = 0x1234
```

3.3.3. BlastOn Tests

blaston p0:

```
PC = 0x0080
CA = 0x0003
a0 = 0xFEDC
p0 = 0x1234
immediate = 0x0000
sp = 0xFF00

Expected: a0 = 0x1234

BlastOn reads the value in the specified register and puts it as the value of
the current accumulator.

IR <= Memory[PC]  #IR = XXXX XXX0 1111 1001
PC <= PC + 2      #PC = 0x0082

A <= Reg[CA]      #A = 0xFEDC
B <= Reg[IR[8:5]  #B = 0x1234

ALUOut <= PC + immediate  #ALUOut = 0x0082
Sp = Reg[sp] <=   #Sp = 0xFF00

Reg[CA] <= B      #a0 = 0x1234
```

3.3.4. BlastOff Tests

```
PC = 0x0000
CA = 0x0003
a0 = 0x0001
p1 = 0x0000

BlastOff reads the value of the current accumulator and puts it into the
specified register

IR = Memory[PC]   IR[8:5]= 1000 (p1)
PC = PC + 2

A = Reg[CA]       A=0X0001
IR = IR

p1 = 0X0001


```

3.3.5. Branch Type Tests

```
branches take the value of a register and updates PC if it is (greater than or equal
Beq

PC = 0X0000
CA = 0X0003
p0 = 0X0000
a0 = 0X0000

IR = Memory[PC]
oldPC = PC
PC += 2

A = Reg[CA]      =0X0000
B = Reg[IR[8:5]]  =0X0000
ALUOut = oldPC + immGen(IR)

A==B --> PC=ALUOut
```

3.3.6. Jump Type Tests

```
Jumps add an immediate to pc and store the current pc to RA or zero
Jar

PC = 0X0000
RA = 0XFFFF
Imm= 0X000F

IR = Memory[PC]
oldPC = PC         =0X0000
PC += 2           =0X0002

ALUOut = oldPC + immGen(IR)     =0X000F
oldPC = PC        =0X0002

PC = ALUOut        = 0X000F
Reg[RA] = oldPC         =0x0002
```

3.3.7. Addi/Andi/Ori/Xori Tests

addi/andi/ori/xori (andi 0x0000):

```
PC = 0x0080
CA = 0x0003
a0 = 0xFEDC
sp = 0xFF00

Expected: a0 = 0x0000

Andi performs a bitwise and operation against the value in the current accumulator
and the specified immediate and stores that value in the current accumulator.

IR <= Memory[PC]  #IR = XXX0 0000 0010 1111
oldPC <= PC       #oldPC = 0x0080
PC <= PC + 2      #PC = 0x0082

A <= Reg[CA]      #A = 0xFEDC
B <= Reg[IR[8:5]  #B = 0x0000

ALUOut <= oldPC + immGen(IR)  #ALUOut = 0x0080
Sp = Reg[sp] <=   #Sp = 0xFF00

ALUOUT <= A op immGen(IR)  #op = and
                           #ALUOut = 0x0000

Reg[CA] <= ALUOut #a0 = 0x0000
```

3.3.8. Slt Test

```
SLT: sets the current accumulator to 1 if it is less
than the register, 0 otherwise.

slt p0
=========================================================
PC = 0x0000
CA = 0x00CA, value 0x0001
SP = 0x00BB, value 0x0000
Reg[IR[8:5]] = p0, value 0x0000
---------------------------------------------------------
IR <= Memory[PC]  #get the instruction from Memory Location 0x0000
oldPC <= PC       #set oldPC to currentPC: 0x0000
PC <= PC + 2      #increment PC by 2 to become 0x0002

A <= Reg[CA]      #A gets the contents of the CA register, 0x00CA, value 0x0001
B = Reg[IR[8:5]]  #B gets the contents of the instruction at register p0,
                    bits 8-5, which are 0x0000.
ALUOut = oldPc + immGen(IR)  #ALUout gets value of the oldPC, 0x0000,
                    added to the immediate from the immediate gen, which is 0x0002
Sp = Reg[SP]      #Sp gets the value of register SP, 0x00BB, value 0x0000

if(A < B)     # A (1) < B (2), so CA will be 1
  Reg[CA] <= 1         # CA gets overwritten as 1
else
  Reg[CA] <= 0
---------------------------------------------------------------------
Expected Values:
PC = 0x0002
oldPC = 0x0000
CA = 0x00CA, value 0x0001
SP = 0x00BB, value 0x0000
Reg[IR[12:5]] = 0x0002
A = 0x0001
ALUOut = 0x0002
Sp = 0x0000
immGen(IR) = 2

Actual Values:
PC = 0x0002
oldPC = 0x0000
CA = 0x00CA, value 0x0001
SP = 0x00BB, value 0x0000
Reg[IR[12:5]] = 0x0002
A = 0x0001
ALUOut = 0x0002
Sp = 0x0000
immGen(IR) = 2
```

3.3.9. Slti Test

```
SLTI: sets the current accumulator to 1 if it is less
than the immediate, 0 otherwise.

slti 2
=========================================================
PC = 0x0000
CA = 0x00CA, value 0x0001
SP = 0x00BB, value 0x0000
Reg[IR[12:5]] = 0x0002
---------------------------------------------------------
IR <= Memory[PC]  #get the instruction from Memory Location 0x0000
oldPC <= PC       #set oldPC to currentPC: 0x0000
PC <= PC + 2      #increment PC by 2 to become 0x0002

A <= Reg[CA]      #A gets the contents of the CA register, 0x00CA, value 0x0001
B = Reg[IR[8:5]]  #B gets the contents of the instruction at 0x0000,
                    bits 8-5, which are useless in this case.
ALUOut = oldPc + immGen(IR)  #ALUout gets value of the oldPC, 0x0000,
                    added to the immediate from the immediate gen, which is 0x0002
Sp = Reg[SP]      #Sp gets the value of register SP, 0x00BB, value 0x0000

if(A < immGen(IR))      # A (1) < immGen(IR) (2), so CA will be 1
  Reg[CA] <= 1          # CA gets overwritten as 1
else
  Reg[CA] <= 0
---------------------------------------------------------------------
Expected Values:
PC = 0x0002
oldPC = 0x0000
CA = 0x00CA, value 0x0001
SP = 0x00BB, value 0x0000
Reg[IR[12:5]] = 0x0002
A = 0x0001
ALUOut = 0x0002
Sp = 0x0000
immGen(IR) = 2

Actual Values:
PC = 0x0002
oldPC = 0x0000
CA = 0x00CA, value 0x0001
SP = 0x00BB, value 0x0000
Reg[IR[12:5]] = 0x0002
A = 0x0001
ALUOut = 0x0002
Sp = 0x0000
immGen(IR) = 2

```

3.3.10. Lui Test

```
LUI: sets the current accumulator to the top 8 bits of the immediate.

lui 2570
=====================================================================
PC = 0x0000
CA = 0x00CA, value 0x0001
SP = 0x00BB, value 0x0000
Reg[IR[15:5]] = 0x0A0A
---------------------------------------------------------------------
IR <= Memory[PC]  #get the instruction from Memory Location 0x0000
oldPC <= PC       #set oldPC to currentPC: 0x0000
PC <= PC + 2      #increment PC by 2 to become 0x0002

A <= Reg[CA]      #A gets the contents of the CA register, 0x00CA, value 0x0001
B = Reg[IR[8:5]]  #B gets the contents of the instruction at 0x0000,
                   bits 8-5, which are useless in this case.
ALUOut = oldPc + immGen(IR)  #ALUout gets value of the oldPC, 0x0000, added to the
                     immediate from the immediate gen, which is 0x0002
Sp = Reg[SP]      #Sp gets the value of register SP, 0x00BB, value 0x0000

Reg[CA] <= immGen(IR)[15:7]
---------------------------------------------------------------------
Expected Values:
PC = 0x0002
oldPC = 0x0000
CA = 0x00CA, value 0x000A
SP = 0x00BB, value 0x0000
Reg[IR[12:5]] = 0x0002
A = 0x000A
ALUOut = 0x0A0C
Sp = 0x0000
immGen(IR) = 2570

Actual Values:
PC = 0x0002
oldPC = 0x0000
CA = 0x00CA, value 0x000A
SP = 0x00BB, value 0x0000
Reg[IR[12:5]] = 0x0002
A = 0x000A
ALUOut = 0x0A0C
Sp = 0x0000
immGen(IR) = 2570
```

3.3.11. Swap Test

```
PC = 0x0000
CA = 0x00CA, value 0x0001
SP = 0x00BB, value 0x0000
Reg[IR[8:5]] = 0x0002

Swap: switches the current accumulator to rs1

swap a2

IR = Memory[PC]      #get the instruction from Memory Location 0x0000
oldPC <= PC          #set oldPC to currentPC: 0x0000
Pc = PC + 2          #increment PC by 2 to become 0x0002

a = Reg[CA]          #A gets the contents of the CA register, 0x00CA, value 0x0001
B = Reg [IR[8:5]]    #B gets the contents of the instruction at 0x0000, bits 8-5.
ALUOut = oldPc + immGen(IR)  #ALUout gets value of the oldPC, 0x0000,
                              and the immediate gen, which is empty
Sp = Reg[SP]         #Sp gets the value of register SP, 0x00BB, value 0x0000

CA = B               #CA becomes B, the contents of the instruction at 0x0000,
                     bits 8-5, and the current value is overwritten
```

3.3.12. AddSP Test

```
PC = 0x0000
CA = 0x00CA, value 0x0001
SP = 0x00BB, value 0x0000
Reg[IR[8:5]] = 0x0002

Addsp: increments the sp by the imm and stores new value into sp
addsp 4

IR = Memory[PC]         #get the instruction from Memory Location 0x0000
oldPC <= PC             #set oldPC to currentPC: 0x0000
Pc = PC + 2             #increment PC by 2 to become 0x0002

a = Reg[CA]             #A gets the contents of the CA register, 0x00CA,
                         value 0x0001
B = Reg [IR[8:5]]       #B gets the contents of the instruction at 0x0000, bits 8-5.
ALUOut = oldPc + immGen(IR)  #ALUout gets value of the oldPC, 0x0000,
                              and the immediate gen, which is 4, to become 0x0004
Sp = Reg[SP]            #Sp gets the value of register SP, 0x00BB, value 0x0000

ALUOut = SP + immGen[IR] #AluOut becomes 0x0000 + 4, to become 0x0004
Reg[sp] = ALUOut        #the value of Register SP becomes 0x0004.
```

3.4. Components

Components needed to implement the RTL, with the input, output, and control signals included for each, as well as a list of the RTL symbols that will be implemented with each component.

- **Registers:**
  - **Input Signals:** in[15:0], defaultValue[15:0]
  - **Output Signals:** out[15:0]
  - **Control Signals:** RegWrite[0:0], reset[0:0], clk[0:0],
  - **Symbols Implemented:** PC, oldPC, ALUOut, SP, IR, A, B, CA
- **Register File:**
  - **Input Signals:** TReg[3:0], WriteReg[15:0], WriteDat[15:0], CA[3:0]
  - **Output Signals:** CADat[15:0], SPDat[15:0], TRegDat[15:0]
  - **Control Signals:** Write[0:0], Reset[0:0]
  - **Symbols Implemented:** Reg[]
- **ALU:**
  - **Input Signals:** A[15:0], B[15:0]
  - **Output Signals:** ALU_Out[15:0], ALU_Flag[1:0]
  - **Control Signals:** ALU_Op[3:0]
  - **Symbols Implemented:** +, OP
- **Memory:**
  - **Input Signals:** addr[15:0], writeData[15:0]
  - **Output Signals:** readData[15:0]
  - **Control Signals:** memRead[0:0], memWrite[0:0], clk
  - **Symbols Implemented:** Memory[]
- **Immediate Generator:**
  - **Input Signals:** Inst[15:0]
  - **Output Signals:** IMM[15:0]
  - **Control Signals:** OP[1:0]
  - **Symbols Implemented:** immGen()
- **Comparator:**
  - **Input Signals:** A[15:0], B[15:0]
  - **Output Signals:** aLTb[0:0], aEQb[0:0], aGEb[0:0]
  - **Symbols Implemented:** <, ==, >=
- **ALU Control:**
  - **Input Signals:** IR[15:0]
  - **Output Signals:** ALUOp[3:0]

4. Component Specifications

4.1. Comparator

The comparator component takes in the value of 2 different 16 bit inputs and outputs a single logical value which corresponds to their comparison. The schematic symbol for the comparator can be seen in Figure 4.1 and a example truth table for a simple 1 bit comparator which gives an overview of the funcionality of a comparator can be seen in Table 4.1

List of Outputs

- $A > B$: This will output high when the value of A is greater than the value of B
- $A = B$: This will output high when the value of A is equal to the value of B
- $A < B$: This will output high when the value of A is less than the value of B

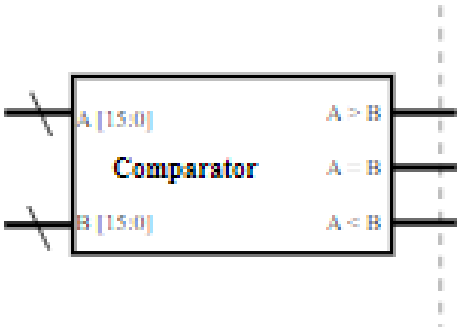


Figure 4.1.: Comparator Schematic Symbol

Table 4.1.: Example 1 Bit Comparator Truth Table

A	B	A > B	A = B	A < B
1	0	1	0	0
0	1	0	0	1
1	1	0	1	0

4.1.1. Hardware Implementation Plan

The following hardware implementation as seen in figure **??** will be used to create a comparator. Each one of the inputs will be passed into a separate adder, the second input is inverted. The carry bit of each is transferred to the next adder and the final bit can be used to determine if A > B, The sums of all of the adders are ANDed together and that output will determine if the inputs are equal to each other. To determine if A < B one can simply NOR the AND gate output and the final carry bit.

4.1.2. Unit Tests

To unit test the comparator, we will develop a Verilog test bench that tests the functionality of each comparison: <, >, =. Once the comparison has been done, the test bench will determine whether the correct output signal is set to 1. If not, the failure will be marked in the console, which will then be used for debugging purposes.

4.2. ALU

The ALU takes has 3 inputs A, B, and OP. A and B are the operands and the OP allows for a selection of what operation is to be done on the operands. One can see a list of ALU operations in table 4.2. The schematic symbol can be seen below in Figured 4.2

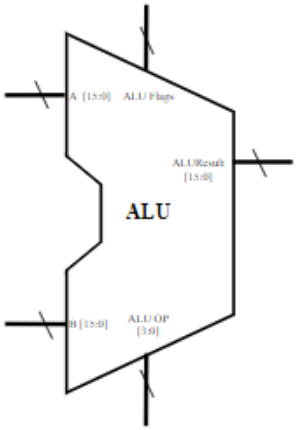


Figure 4.2.: ALU Schematic Symbol

The ALU OP input takes in a 4 bit input and uses that to determine the given operation. A 4 bit input was used to allow for future expand-ability for more operations.

Table 4.2.: ALU Operations

Operation	Code	Description
A + B	0000	Add
A - B	0001	Subtract
A « B	0010	Shift Left
A & B	0011	Bitwise AND
A | B	0100	Bitwise OR
A ⊕ B	0101	Bitwise XOR

The ALU also has status flags which will occur depending on the result of the operation. This can be seen in Table 4.3.

Table 4.3.: ALU Status Flags

Flag	Code	Description
N	00	Set when the result of the operation was Negative
Z	01	Set when the result of the operation was Zero
V	10	Set when the operation caused overflow

4.2.1. Hardware Implementation Plan

Case statements in Verilog will used to implement this unit as seen below.

```
case(ALU_OP)
    3'b0000: // Addition
       ALU_Result = A + B ;
    3'b0001: // Subtraction
       ALU_Result = A - B ;
    3'b0010: // Shift Left
       ALU_Result = A << B;
    3'b0011: // Bitwise AND
       ALU_Result = A & B;
    3'b0100: // Bitwise OR
       ALU_Result = A | 1;
    3'b0101: // Bitwise XOR
       ALU_Result = A ^ 1;
    default: ALU_Result = A + B ;
endcase

```

4.2.2. Unit Tests

To test the ALU, we would write a Verilog test bench to test all of the different possible ALU Op code inputs to determine whether or not the ALU performed the operation correctly.

4.3. Registers

The individual registers throughout the data path store values for use in-between cycles since they will be lost, otherwise.

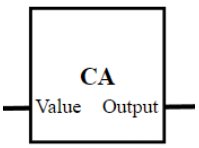


Figure 4.3.: Register Schematic Symbol

4.3.1. Hardware Implementation Plan

To implement a Register in hardware, we will create a Verilog module that has 1 input, value, and 1 output, output. Since this module acts only as data holder, there is nothing more this module will need to do. There will also be a version of this register that has a reset input to it. This reset input will set the register back to a default value as specified by a paramter during it's creation. This version of the register will be used for IR to hold instructions, PC along with old PC to hold to the program count.

4.3.2. Unit Tests

To test the Verilog implementation of this component, we will test writing a value to the register and ensure the value is held until it is changed again. We will also be testing the required clock cycles needed to change the value in the register. For the registers with the default values we will need to test that the reset will reset back to this default value.

4.4. Register File

The register file will store the bulk of the registers for the processor. The register file has the following inputs and outputs.

Inputs

- **Reg Write:** Enables the ability to write to the register file. Set input to low to do read only
- **Select CA:** With a given immediate (1-4) it will select it will select which accumulator to set as the current accumulator
- **Write Data:** Determines the data to write to a given register from write reg
- **Write Reg:** Determines which register to write to
- **Read Reg:** Determines which register to read from a given instruction

Outputs

- **SP:** Stack pointer register output
- **Read Current Accumulator:** Outputs the current accumulator value
- **Read Data 2:** Outputs the second register value
- **RA:** Return Address Register Output

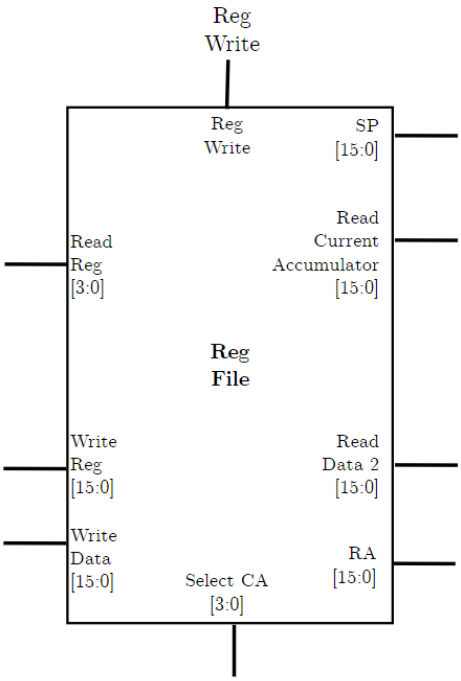


Figure 4.4.: Register File Schematic Symbol

4.4.1. Hardware Implementation Plan

To implement the Register File in hardware, we will create a Verilog module that has 3 inputs: Read Reg, Write Reg, Write Data; 3 outputs: SP, Read Current Accumulator, Read Data 2; and 2 control signals: Reg Write, Select CA. If Reg Write is set to 1, we will write the data from the Write Data

input to the register denoted in Write Reg. Registers are stored in an array and accessed as such. Select CA changes the register that the current accumulator points to.

4.4.2. Unit Tests

To test the Verilog implementation of this component, we will test each individual function of the register file: reading, writing, and swapping. To test reading, we will give the register file a register with a value to read and ensure that the value comes out of Read Data 2. To test writing, we will give the register file a register to write to and a subsequent value and ensure that value is what is in the register after running. To test the swapping of accumulators, we will provide the register file with a new register identifier to point the current accumulator to. After this, we will ensure that the current accumulator points to where it's supposed to be.

4.5. Memory

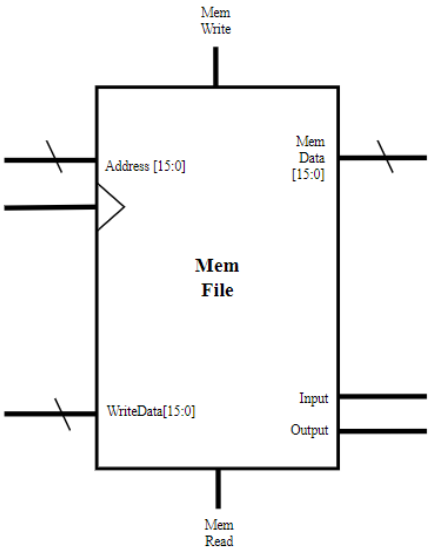


Figure 4.5.: Memory File Schematic Symbol

4.5.1. IO

The memory module includes input and output. For our processor this will be used for memory mapped IO. There will be a dedicated section of memory for memory mapped IO as seen in Figure 4.6. The input is address with 0x00 and the output is addressed with 0x08.

Using memory mapped IO in our processor allows us to more easily pass in input and output using already created instructions. Examples of input and output can be seen below.

Example of Loading Input to P0

```
1 blastOn zero
2 lw 0(zero)
3 blastOff p0
```

Example of Storing output on to the stack

```
1 sw 8(zero)
```

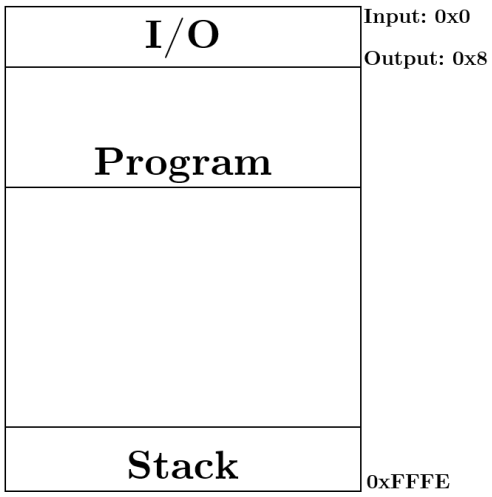


Figure 4.6.: Memory File Layout

4.5.2. Hardware Implementation Plan

To implement the Memory File in hardware, we will create a Verilog module that has 2 inputs: Address, Write Data; 3 outputs: Read Data, Input, Output; and 1 control signal: Mem Write. If Mem Write is set to 1, we will write the data from the Write Data input to the address denoted in address. Memory is stored in a single cell RAM template as recommened by quartus.

To add IO to the memory a wrapper will be created around memory that will detect when an input or output address is selected. When an input address is selected the input will be redirected to the read data output. When an output is selected the write data input will be redirected to the output.

4.5.3. Unit Tests

To test the Verilog implementation of this component, we will test each individual function of the memory file: reading and writing. To test reading, we will give the memory file an address with a value to read from and ensure that the value comes out of Read Data. To test writing, we will give the memory file an address to write to and a subsequent value and ensure that value is what is in memory at that address after running. To test input a given input will be sent in with an input address and that input will be confirmed to be the read data output. To test output a given write data input will be sent in with an output address and the output will be confirmed to be equal to the given input.

4.6. Immediate Generator

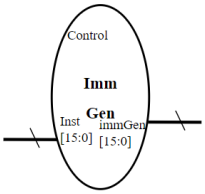


Figure 4.7.: Immediate Generator Schematic Symbol

4.6.1. Hardware Implementation Plan

The immediate generator takes a 2 bit control signal and an instruction and outputs a sign extended Immediate. The generator is implemented in Verilog by taking bytes of the instruction and copying into the extended immediate. The control signal is used to determine where bytes are read from.

4.6.2. Unit Tests

To test the Verilog implementation of this component, instructions that had (in base 10) 0, 0, 16, and -1 as the translation of their immediate bytes were passed into the generator with the appropriate control signal. For the first 0 input all bits unused by the generator were set to 0 and for the second all were set to 1. All valid control signals were tested under these conditions.

4.7. ALU Control

The ALU Control unit takes in the 16 bit instruction as the input, and decodes the instruction to determine a 4 bit op code that will control the ALU. The schematic symbol for the control unit can be seen in figure 3.7.

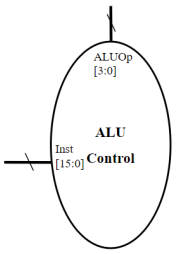


Figure 4.8.: ALU Control Schematic Symbol

4.7.1. Hardware Implementation Plan

To implement the ALU Control unit in Verilog, we will break apart the 16 bit input to isolate the instruction's op code and funct2 to determine which operation to use. Once the operation is determined, the ALUOp output signal will be set to the appropriate code, which are described in Table 3.2.

4.7.2. Unit Tests

To test the ALU Control unit, we will create a Verilog test bench that tests each type of operation that the control unit could output. For example, we would test the unit with a 16 bit input that is identical to an add instruction, and then affirm that the unit produces a 0000 Op code. The other tests would test the other operations and affirm their outputs as well.

5. Multi-Cycle Data Path

The diagram below represents the multi-cycle data path implementation containing the components outlined above.

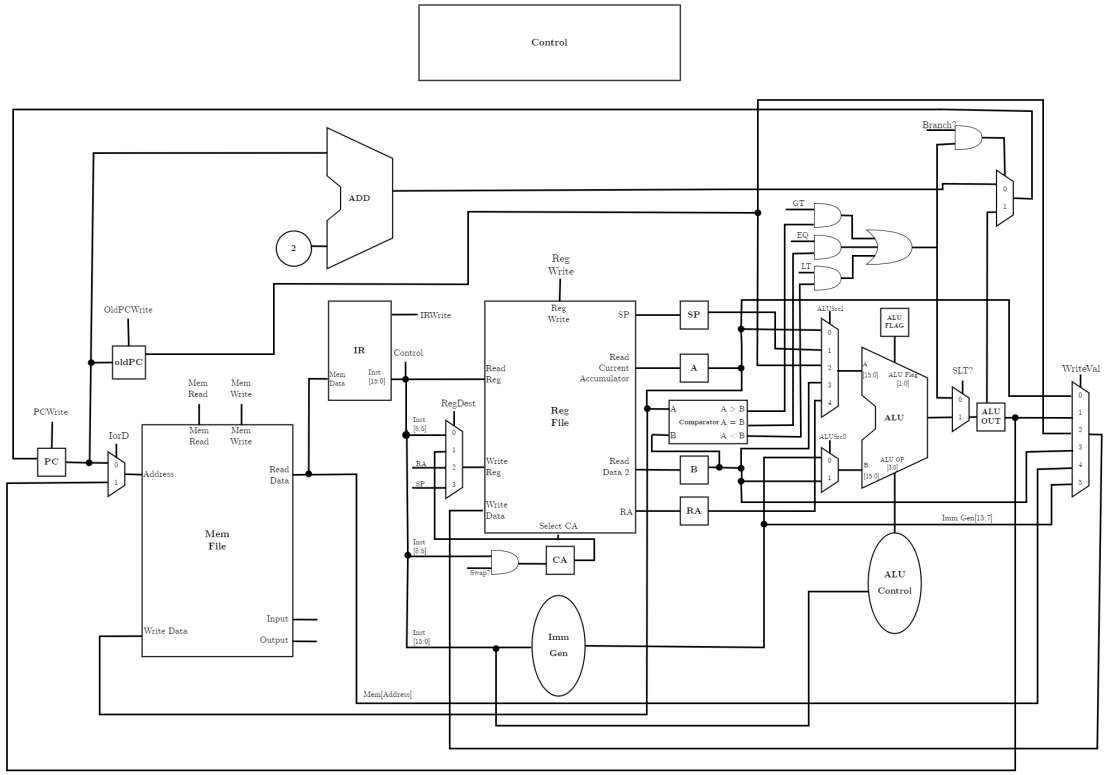


Figure 5.1.: Multi-Cycle Data Path

6. Control Specifications

6.1. Control Signals

- **IorD:** 1 bit. Determines whether we use the value from PC (instruction) or ALUOut (data) to perform memory operations at.
- **MemWrite:** 1 bit. Determines whether we want to write data into memory at the specified address.
- **RegWrite:** 1 bit. Determines whether or not we are able to write into the registers stored in our register files.
- **RegDest:** 2 bits. Determines the destination register the result of our instruction will write into (if applicable). Currently there are four options:
  - **00:** Write into the specified register in the instruction (rs1).
  - **01:** Write into the current accumulator.
  - **10:** Write into the return address register.
  - **11:** Write into the stack point register.
- **GT:** 1 bit. Determines whether there is a valid branch if A is greater than B.
- **LT:** 1 bit. Determines whether there is a valid branch if A is less than B.
- **EQ:** 1 bit. Determines whether there is a valid branch if A is equal to B.
- **Swap?:** 1 bit. Determines whether or not we are performing our swap instruction, and need to update the value in our CA (current accumulator) register.
- **SLT?:** 1 bit. Determines whether or not we are performing a Set Less Than instruction. Saves the logic from the comparator into ALUOut instead of the output of the ALU.
- **Branch?:** 1 bit. Determines whether or not we are currently performing a branch instruction. Used in conjunction with GT, LT, and EQ to determine the value of the next PC address.
- **ALUOp:** 4 bits. Determines the operation our ALU will perform on its two inputs. Currently the four bit size allows for a potential sixteen different operations we can perform, although currently we are only utilizing six.
- **ALUSrc1:** 3 bits. Determines the value of the first input for our ALU. Currently there are four options:
  - **000:** The value stored in the register A.

  - **001:** The value stored in the register Sp.
  - **010:** The value stored in the register oldPC.
  - **011:** The value stored in the register B.
  - **100:** The value stored in the register Ra.

- **ALUSrc2:** 1 bit. Determines the value of the second input for our ALU. Currently there are two options:
  - **0:** The immediate generated by our immGen.
  - **1:** The value stored in the register B.

- **WriteVal:** 3 bits. Determines the result we would like to write into either our memory file or our register file. Currently there are five options:
  - **000:** The value stored in the register A.
  - **001:** The result of our ALU operations (found in the register ALUOut).
  - **010:** The value stored in the register oldPC.
  - **011:** The value stored in the register B.
  - **100:** The value pulled from memory at the address in ALUOut (Mem[ALUOut]).
  - **101:** The immediate generated by our immGen.

- **PCWrite:** 1 bit. Determines whether we can overwrite the value currently stored in the PC register with the value from our PC subsystem.

- **IRWrite:** 1 bit. Determines whether we can overwrite the value currently stored in the Instruction Register with the instruction fetched from the memory file.

- **Reset:** 1 bit. Cannot be activated through an instruction. When activated, all registers' contents are reset to its specified default value.

6.2. Control Unit Specification

The control unit takes in the instruction parsed from the instruction register (IR), and decides based on the instruction (parsed from the combination of our op code and f2 fields) the various signals to the other components and registers in our multi-cycle implementation.

The mapping of our control signals for each instruction per cycle for our multi-cycle implementation can be found in the appendix.

In addition, the control unit contains the ability to activate a reset signal, which when activate will wipe all of the contents stored in registers. While this capability cannot be activated through an instruction, it can be utilized through a single bit input to the control unit, which will in turn activate the reset signal.

6.3. Control Unit Testing

The control unit will be tested through procedurally going through each instruction and testing each of the possible outputs through each cycle of the instruction.

Certain signals change per cycle of the instruction (ALUOp and ALUSrc1 being good examples of this). Special care will be used to ensure that the signals change to their correct values at the appropriate times. The handling of these control signals will be handled through the individual components and subsystems and their own unique testing.

In addition, to handle the issue of "dangling" control signals, certain signals will be defaulted to 0 after each state change. Such signals are: PCWrite, IRWrite, MemWrite, RegWrite.

6.4. ALU Control Unit Specification

ALU Control functions as a sub-unit to the main control unit. It is separated to help simplify the logic and clarify its role and process in our visual datapath. Similar to the control unit, ALU Control will received the parsed instruction from the instruction register, and depending on the received opcode and f2 combination, the appropriate operation will be performed on the operands.

6.5. ALU Control Unit Testing

Like the control unit, ALU control will be tested on an instruction basis. The logic is similar, albeit smaller in scope.

Like the control unit, ALUOp is subject to change over the multiple cycles of an instruction. While the first instruction will always be add for our PC + immediate calculation, the second instruction can be one of the other five operations it can perform. Sometimes the result of the ALU is not used. In these cases, the ALUOp will be defaulted to add, even if the summation is not used.

7. Testing

7.1. Unit Testing

Unit Testing for each component can be seen in Section 2.4:

Each component will be tested using Boundary Value Analysis, with the minimum value, a nominal-minimum value, a nominal value, a nominal- maximum value, and maximum value checked for each. We will also take into special consideration edge cases that could cause the instruction to fail, and test for them in our cases. Components will be modified until the tests pass.

7.2. Integration Testing

Components will first be integrated into subsystems, and then into the system as a whole. Integration Testing will be used to catch faults in the components not already found with the Unit Tests, and will be done with an iterative approach. We will first start by isolating components into subsystems, and once no faults are found in the subsystems, the subsystems will be combined to create each individual cycle, until the entire system has been created and is being tested.

We will integrate the components based on their functionality, making small subsystems that can be combined and tested to create the entire system. For example, we could test the combination of the comparator and branching logic to ensure the branching subsystem works. Another subsystem that could be tested could be the Immediate Generator and the ALU, to verify immediate arithmetic.

7.3. Subsystem Testing

Subsystems will be defined based on the components needed for specific instructions. Subsystem tests will still follow BVA; however, the tests themselves will focus on the output of the whole system itself rather than the individual components of the system. For instance, when testing the branching subsystem, we would ensure the subsystem outputs the correct branch signal instead of the individual comparator outputs.

Initially, we will begin by developing and testing small distinct subsystems, that are then combined to create bigger subsystems once the smaller subsystems are tested thoroughly. Once we have gotten to the point where there are no more subsystems, we will advance to System Testing.

7.4. System Testing

All Unit Tests, Integration Tests, and Subsystem tests will be ran on the system as a whole. Each individual subsystem will be combined together and tested using both a Verilog testbench, to test the combined subsystems, and the RelPrime algorithm to ensure our system works to specification.

With RelPrime being our final benchmark, we will start with checking the output of a simple arithmetic instructions: simple addition and subtraction with values in registers along with immediates. After arithmetic instructions are proven to be functional, we will then begin testing instructions that put values into registers such as blastOn and blastOff. Once we have verified that all arithmetic and register based instructions are working, we will begin to focus on testing pulling data and storing data into memory.

To test memory, we will test instructions such as lw and sw to ensure that they are correctly inserting values onto the stack and other memory addresses. After ensuring that these instructions are functional, we will then begin to test input and output since we using memory mapped IO. Next, we will start running and verifying that basic loops are functional, which will test our branch and jump instructions. With the basis of our instruction set now defined and tested as a whole, we can start to test multiple procedure calls, along with validation of our calling conventions by loading small programs into memory such as the defined calling procedures example and GCD. Once we have verified that our procedure calls are working correct, we will then begin to test and debug RelPrime until the expected output is received.

8. Subsystems Specifications

The different subsystems of our data path are indicated by the different colored sections seen in Figure 8.1.

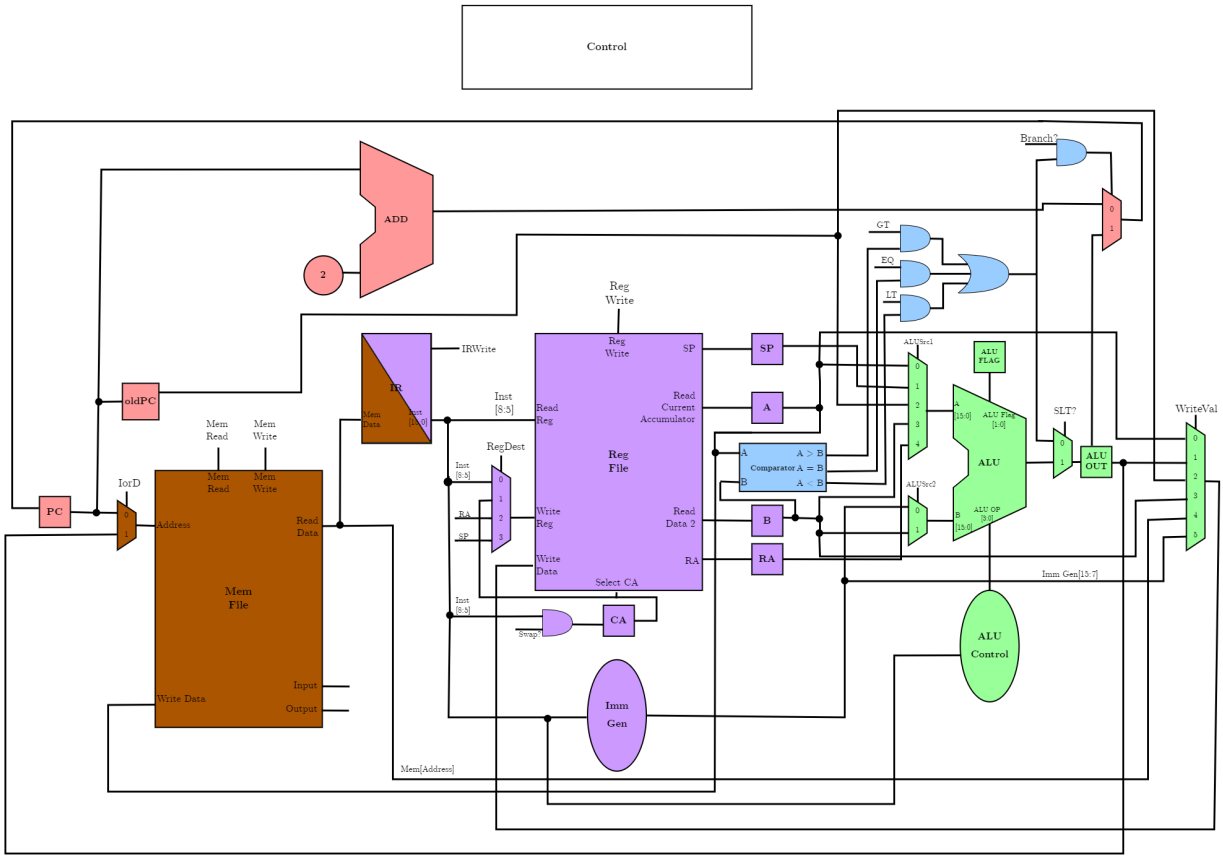


Figure 8.1.: Subsystem Specification on Data Path

Based on the colors in the image, the denoted subsystems are as follows:

- **Red** - PC Subsystem
- **Brown** - Memory Subsystem
- **Purple** - Register File Subsystem
- **Blue** - Branch Subsystem
- **Green** - ALU Subsystem

8.1. PC Subsystem

This subsystem represents the components in the data path that handle the changing of the PC for either a jump, branch, or move to a next instruction.

The PC_Dest "control signal" comes from the branch subsystem, and that connection will be tested when the two subsystems are combined for further subsystem testing. WritePC determines whether the value in PC can be overwritten, determined by the current state of the control unit.

The PC subsystem uses a multiplexor, controlled by the branch subsystem logic, to determine whether the PC is rewritten to PC + 2 or PC + immediate.

Components List
- PC register
- oldPC register
- PC + 2 Adder
- ALUOut register
- PC_Dest "control signal"
- PCWrite control signal

Inputs
- ALUOut (16 bits)
- PC_Dest (1 bit)
- PCWrite (1 bit)

Outputs
- PC (16 bits)
- oldPC (16 bits)

8.2. Memory Subsystem

This subsystem represents the components in the data path that handle the use of memory for either reading, writing, instruction fetching, and external input and output.

The Memory subsystem uses an address determined by a multiplexor, controlled by the IorD control signal, to either read from or write to in the memory array based on the MemRead and MemWrite control signals. The read data in memory is then stored in the instruction register for decoding. Similar to the contents of the PC register only being able to written to while PCWrite is asserted, the Instruction Register's contents can only be changed if the IRWrite control signal is asserted.

Component List
- Memory File
- Instruction register
- IorD control signal
- MemWrite control signal
- IRWrite control signal.

Inputs
- PC (16 bits)
- ALUOut (16 bits)
- IorD (1 bit)
- MemRead (1 bit)
- MemWrite (1 bit)
- IRWrite (1 bit)
- IOInput (16 bits)

Outputs
- Mem[Address] (16 bits)
- IO Output (16 bits)
- IR (16 bits)

8.3. Register File Subsystem

This subsystem represents the components in the data path that handle the use of the register file for either reading from or writing to a register in the file.

The Register File subsystem uses the address provided by the instruction register to either read from in the register array, which is stored into B. The subsystem also uses a write address, determined by a multiplexor, controlled by RegDest, to write the data passed into it. This subsystem always outputs the current stack pointer into the SP register, and the value of the current accumulator in A.

Components List
- Register File
- RA register
- SP register
- A register
- B Register
- CA Register
- Immediate Generator
- RegDest control signal
- Swap control signal
- RegWrite control signal

Inputs
- Swap (1 bit)
- RegDest (1 bit)
- RegWrite (1 bit)
- Instruction (16 bits)

Outputs
- Sp (16 bits)
- A (16 bits)
- B (16 bits)
- RA (16 bits)
- ImmGen (16 bits)

8.4. Branch Subsystem

This subsystem represents the components in the data path that handle the branch comparisons needed to determine whether or not the PC should be affected by ALUOut.

The Branch subsystem uses the values provided by the A and B registers in the comparator to output relevant signals as to whether A is either less than, equal to, or greater than B. These singles are then combinationally combined with the control signals to determine whether or not the PC should be overwritten.

Components List
- Comparator component
- A register
- B register
- Branch control signal
- GT control signal
- EQ control signal
- LT control signal

Inputs
- A (16 bits)
- B (16 bits)
- GT (1 bit)
- LT (1 bit)
- EQ (1 bit)
- Branch (1 bit)

Outputs
- PC_Dest (1 bit)
- SLT_Val (1 bit)

8.5. ALU Subsystem

This subsystem represents the components in the data path that handle the operations and values that go through the ALU.

The ALU subsystem uses the values provided two multiplexors, controlled by the ALUSrc1 and ALUSrc2 control signals, as inputs to the ALU. The operation between those inputs is determined by the ALUOp control signal determined by the ALU Control unit. The output of the ALU is stored in the ALUOut register unless the SLT control signal is 1, then the input SLT_Val is written to ALUOut. Any flags thrown by the ALU are stored in the ALUFlag register for use later. Finally, the control signal WriteVal determines which output (OutVal) will be sent to the Register File.

Components List
- ALU component
- ALU Control unit
- ALUOut register
- ALUFlag register
- ALUOp control signal
- ALUSrc1 control signal
- ALUSrc2 control signal
- SLT control signal
- WriteVal control signal

Inputs
- A (16 bits)
- Sp (16 bits)
- oldPC (16 bits)
- B (16 bits)
- ImmGen (16 bits)
- ALUSrc1 (3 bits)
- ALUSrc2 (1 bit)
- SLT (1 bit)
- SLT_Val (1 bit)
- WriteVal (3 bits)

Outputs
- OutVal (16 bits)
- ALUOut (16 bits)
- ALUFlag (2 bits)

9. Performance

When running the S.W.H.A.P. processor with an input of 0x13B0, the processor was observed to run a total of 61,384 instructions in roughly 20.47 ms, producing the output 0x000B as seen in Table 9.1 and Figure 9.2. Considering the amount of looping that must be done to find the correct M that is relatively prime to 0x13B0, it is understandable that 61 thousand instructions were run. Additionally, the processor was observed to go through 204,726 different cycles. This combined with the number of instructions produces the average Cycles Per Instruction (CPI) to be 3.34 cycles. Considering most of our instructions in the instruction set are 4 cycles and the relPrime program contains a lot of swap instructions which are 3 cycles, a CPI between 3 and 4 was highly expected.

Table 9.1.: Performance Summary

Metric	Result
Bytes Needed for Relprime	142
Total Registers	407
Total Mem Bits	524,288/608,256 (86%)
Total Logical Elements	1,338/28,848 (5%)
Instructions	61,384
Cycles	204,726
CPI	3.34
Total Exec Time for 0x13B0 (ms)	20.47295
Clock Frequency	76.46 MHz

Overall, the relPrime program consumes 142 bytes of the memory file, which includes all instructions, memory variables, and constants. Though this is larger than we initially would have hoped, it is understandably this large considering the addition of a branch delay slot after each branch or jump. It was decided that it would be best to trade some of our memory space for performance, so 142 bytes makes sense. Our clock speed was found to be 76.46 MHz as seen in Figure 9.1. The total number of registers used throughout the entire design was found to be 407 registers. Additionally, the total number of memory bits used is 524,288 which is 86% of the total memory bits available. Finally, the total number of logical elements was found to be 1,338, which is 5% of the total logical elements available to be used.

	Fmax	Restricted Fmax	Clock Name
1	76.46 MHz	76.46 MHz	clk
2	161.03 MHz	161.03 MHz	Control:control|ALUSrc1[0]
3	173.73 MHz	173.73 MHz	Control:control|CurrentState[0]
4	178.7 MHz	178.7 MHz	Control:control|WriteVal[1]
5	178.57 MHz	178.57 MHz	MemorySubsystemWithIO:memSub|reg...ithResetAndWrite:IR|outResult
6	181.49 MHz	181.49 MHz	Reg_Subsystem:register|immop[1]

Figure 9.1.: Quartus Screen Captures

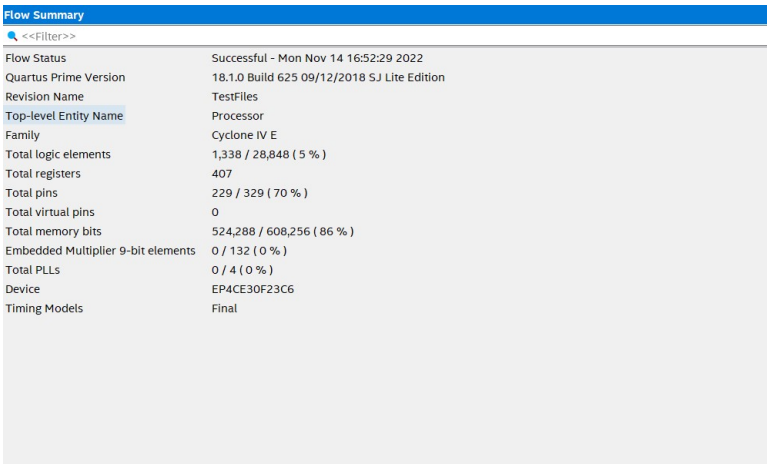


Figure 9.2.: Quartus Screen Captures

10. Machine Code Assembler

10.1. Instructions for Basic Usage

1. Go to `baseinput.txt`
2. Write assembly using pseudo instructions, labels and changing where binary is written to as needed
3. Ensure no errors
4. Copy output from `output.txt`

10.2. Assembly Language Tokens and Usage

10.2.1. Enable/Disable memory locations and comments

By default the assembler will output comments and the assumed hex values of the address in memory where an instruction will lie. If disabling this is desired to, say, copy binary directly into a test file, writing a line that says "disableExtras" will disable these extra printouts.

10.2.2. Set Memory Address %

% - sets memory address. Can be changed mid-file if needed. sets the memory address where the next instruction will be written. Starts writing to 0X0000 by default.

Example

```
% 48879
```

In this example, the next instruction will be written at address 0XBEEF

```
% 0
Inst1
Inst2
% 18
Inst3
```

In this example Inst1 is written at 0X0000, Inst2 is written at 0X0002 and Inst3 is written at 0X0016.

10.2.3. Add a Label $

$ - add a label. Labels will interpret all text (including colons) after $ and before a space as a the label

Examples

```
$ loop
Inst
J loop
```

In this example j jumps to "loop"

```
$ loop:
Inst
J loop:
```

In this example j jumps to "loop:"

If % and $ are used in conjunction, jumps and branches will still be calculated correctly

```
% 0
$ start
Inst
J notstart
% 500
$ notstart
J start
```

In this example if pc is ever set to 0X0000 an infinite loop of performing Inst, jumping to not start and jumping to start will occur.

10.3. Instruction Syntax

Instruction arguments are split using spaces, not commas. Having a space before an instruction WILL result in an error, having multiple spaces between instruction arguments WILL result in an error, having commas anywhere **WILL** result in an error. Anything after the last expected argument will be ignored for the purpose of assembly but will be printed out as a comment. Comment by adding text after a space after an instruction is the safest way to comment. Text after % and $ may or may not be displayed in outputs.

Instructions are not capitalized but the O in blastOn/blastOff is

Spaces in branches, j and jl and parentheses in lw/sw are required

See Table 10.1 for table of instructions syntax.

Table 10.1.: Instruction Syntax

Instruction	Arg1 (and Arg2 for branches)
add	<reg>
sub	<reg>
blastOn	<reg>
blastOff	<reg>
addi	<immediate>
slli	<immediate>
lui	<immediate>
lw	<immediate>(<reg>)
sw	<immediate>(<reg>)
addsp	<immediate>
jar	<label>
j	<immediate> <label> //always uses 0
jl	<immediate> <label> //always uses 0
jarl	<label>
bge	<reg> <label>
beq	<reg> <label>
blt	<reg> <label>
or	<reg>
and	<reg>
xor	<reg>
slt	<reg>
ori	<immediate>
andi	<immediate>
xori	<immediate>
slti	<immediate>
nop	<none>

10.3.1. Labels and Limits

A label can use any character represent able with 33-126 on an ASCII table with the sole exception of “‘” (96) which is reserved for the singleLevelLoop pesudo instruction

10.3.2. Pseudo Instructions

Push

Syntax: “`push <reg>`”
Description: pushes current accumulator to sp-2 and decrements sp by 2

Pop

Syntax: "`pop <reg>`"
Description: pops sp to current accumulator and increments sp by 2

Li

Syntax: "`li <immediate>`"
Description: loads up to a 16 bit immediate into CA

singleLevelLoop

Syntax: “`singleLevelLoop <immediate>`”
Description: prepares to loop immediate times
Restriction: x12, x13, x14 are overwritten and reserved until endLoop is called, cannot call singleLevelLoop again until endLoop is called

endLoop

Syntax: “`endLoop`”
Description: describes the end to singleLevelLoop, all code between singleLevelLoop and endLoop will be run every loop
Requirements: must be called after singleLevelLoop

10.3.3. User Error Catching

The assembler will catch the following errors

1. Immediates being too large for their available size
2. Multiple instructions sharing the same address in memory
3. Instructions being written to outside the available memory space

Bad assembly Example:

```
% 0              // write the next instruction at 0X0000
$ hello          // set a label at memory address 0X0000
j 0 hi           // jump to an address way outside of the
                 //range of a jump (using a positive immediate)
addi 1000000     //uses an immediate that is larger
                 //than 11 bits (positive) (addi uses 11 bits)
addi -1000000    //uses an immediate that is larger
                 //than 11 bits (negative) (addi uses 11 bits)
ori 1000         //uses an immediate that is larger than 7
                 //bits (positive) (ori uses 7 bits)
% 20000000       // write the next instruction well outside of memory's bounds
$ hi             // set a label in the middle of nowhere
j 0 hello        // jump to an address way outside of
                 //the range of a jump (using a negative immediate)
% 0
ori -1000        //uses an immediate that is larger than 7
                 //bits (positive) (ori uses 7 bits)
                 //(also writes to the same address as "j 0 hi")
```

Will Throw the Following Errors:

1. JUMP TOO BIG (A J-TYPE INSTRUCTION TRIED TO JUMP FORWARD MORE THAN 1023 INSTRUCTIONS) Error at line: 4 in input.txt
2. JUMP TOO BIG (A J-TYPE INSTRUCTION TRIED TO JUMP FORWARD MORE THAN 1023 INSTRUCTIONS) Error at line: 4 in input.txt
3. IMMEDIATE TOO LARGE! IMMEDIATE LARGER THAN 1023 PASSED INTO AN INSTRUCTION THAT ACCEPTS UP TO 11 BIT IMMEDIATES Error at line: 5 in input.txt
4. IMMEDIATE TOO SMALL! IMMEDIATE SMALLER THAN -1024 PASSED INTO AN INSTRUCTION THAT ACCEPTS UP TO 11 BIT IMMEDIATES Error at line: 6 in input.txt
5. IMMEDIATE TOO LARGE! IMMEDIATE LARGER THAN 63 PASSED INTO AN INSTRUCTION THAT ACCEPTS UP TO 7 BIT IMMEDIATES Error at line: 7 in input.txt
6. JUMP TOO BIG (A J-TYPE INSTRUCTION TRIED TO JUMP BACKWARD MORE THAN 1024 INSTRUCTIONS) Error at line: 10 in input.txt

7. ERROR: PROGRAM EXCEEDS MEMORY BOUNDS CHANGE % TO BE FURTHER FROM MEMORY EDGE OR SHRINK PROGRAM Error at line: 11 in input.txt
8. IMMEDIATE TOO SMALL! IMMEDIATE SMALLER THAN -64 PASSED INTO AN INSTRUCTION THAT ACCEPTS UP TO 7 BIT IMMEDIATES Error at line: 12 in input.txt
9. ERROR: MULTIPLE INSTRUCTIONS WITH THE SAME ADDRESS, MULTIPLE INSTANTIATIONS OF % TOO CLOSE TO EACH OTHER
10. Error 9 again

Note that errors other than "MULTIPLE INSTRUCTIONS WITH THE SAME ADDRESS" provide a line to go to see where the error originated. To provide additional clarity, this line is in input.txt, not baseinput.txt. If, however, a user wishes to see the line their issue was from in their baseinput.txt file, they may simply go to the line in input.txt where the error occurred and added to the comments of that line, will be a bit of text saying "from line<number>" which corresponds to that instruction's line in baseinput.txt.

11. Conclusion

Designing and creating this processor was a very informative and challenging task. Our team learned a lot in both computer architecture and team work. Something that made this project especially challenging was you were learning in class during the project portion rather than applying already known knowledge. By doing it this way we were able to learn from our previous mistakes and learn more about best practices.

While working on this processor one of the biggest hurdles we needed to overcome was the integration of the subsystems. The part of this that we struggled the most with was timing of the various sub systems. Though each of the systems worked independently of each other the way they were programmed wasn't always to the specifications of what our control was doing. For example creating temporary registers to store values in them that should have been assign statements or making logic that was combinational clock based. To address these issues we created and ran better system tests along with using wave form debugging to address these issues. Doing this allowed us to track down bugs much faster.

This was an extremely challenging but rewarding project but overall our team is extremely proud of the final processor we created and wished we had more time to implement more and more features into this processor.

Appendix

A. Appendix

A.1. Single-Cycle RTL

The initial design was a single-cycle RTL which can be seen below. This is provided to demonstrate the progression of our CPU design and does not necessarily, directly impact our final CPU design.

Table A.1.: Single-Cycle RTL Part 1

Instruction Format	Instruction Name	Instruction Name	Instruction Name	Instruction Name
A	add	sub	pop	pop
	newPC = PC + 2 PC = newPC inst = Mem[PC] a = Reg[CA] b = Reg[inst[8:5]] result = a + b Reg[CA] = result	newPC = PC + 2 PC = newPC inst = Mem[PC] a = Reg[CA] b = Reg[inst[8:5]] result = a - b Reg[CA] = result	newPC = PC + 2 PC = newPC inst = Mem[PC] a = Reg[CA] Reg[inst[8:5]] = a	newPC = PC + 2 PC = newPC inst = Mem[PC] b = Reg[inst[8:5]] Reg[CA] = b
I	addi	lui	slli	
	newPC = PC + 2 PC = newPC inst = Mem[PC] a = Reg[CA] b = SE(inst[15:5]) result = a + b Reg[CA] = result	newPC = PC + 2 PC = newPC inst = Mem[PC] b = SE(inst[15:5]) Reg[CA] = b	newPC = PC + 2 PC = newPC inst = Mem[PC] a = Reg[CA] b = SE(inst[15:5]) result = a « b Reg[CA] = result	
M	addsp	lw	sw	
	newPC = PC + 2 PC = newPC inst = Mem[PC] a = Reg[sp] b = SE(inst[15:9]) result = a + b Reg[sp] = result	newPC = PC + 2 PC = newPC inst = Mem[PC] a = Reg[inst[8:5]] b = SE(inst[15:9]) offset = a + b val = Mem[offset] Reg[CA] = val	newPC = PC + 2 PC = newPC inst = Mem[PC] a = Reg[inst[8:5]] b = SE(inst[15:9]) offset = a + b val = Reg[CA] Reg[offset] = val	

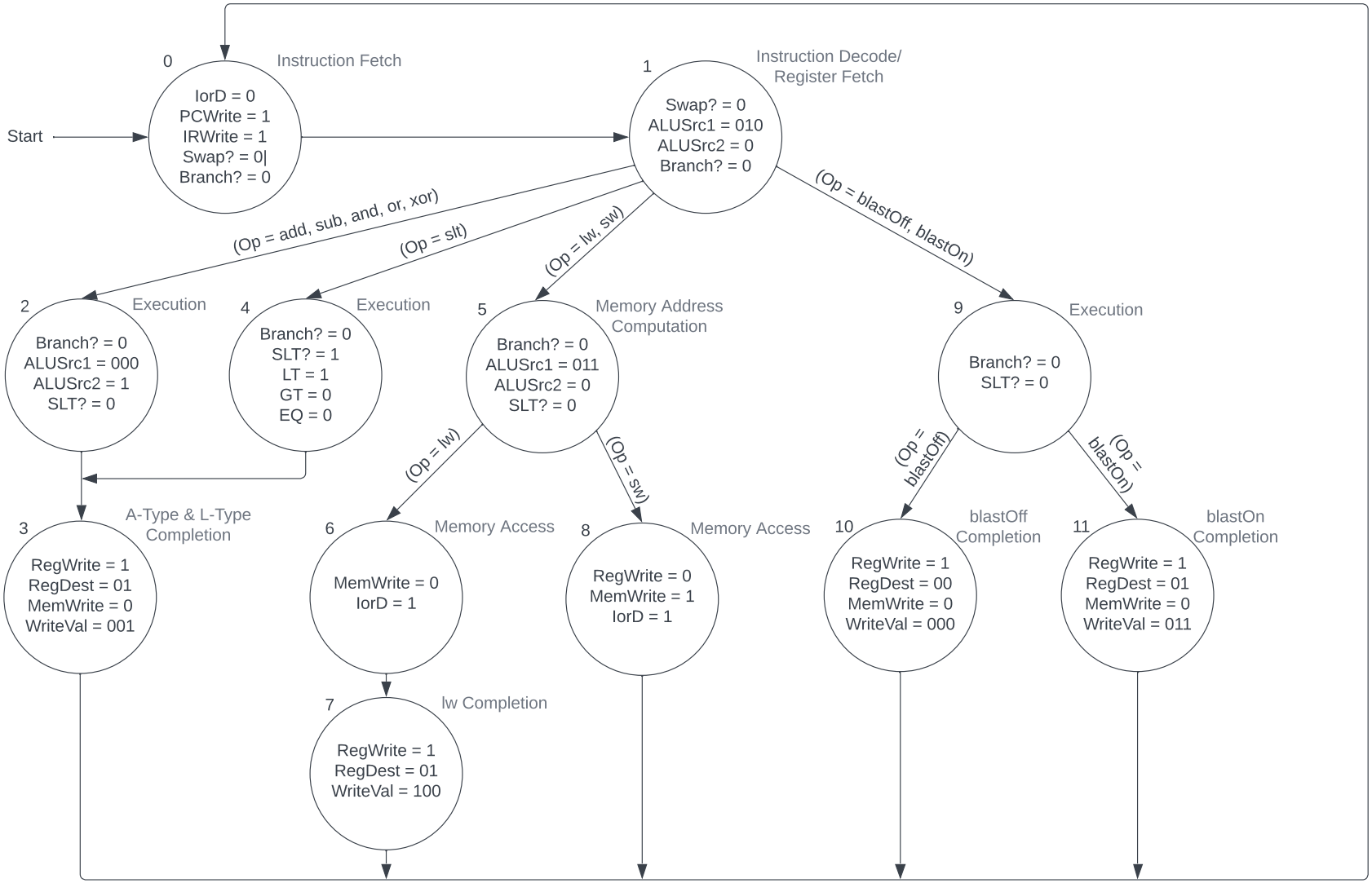
Table A.2.: Single-Cycle RTL Part 2

Instruction Format	Instruction Name	Instruction Name	Instruction Name	Instruction Name
B	beq	bge	blt	
	newPC = PC + 2 PC = newPC inst = Mem[PC] a = Reg[CA] b = SE(inst[8:5]) imm = SE(inst[15:9]) target = PC + imm if (a == b) ? PC = target	newPC = PC + 2 PC = newPC inst = Mem[PC] a = Reg[CA] b = SE(inst[8:5]) imm = SE(inst[15:9]) target = PC + imm if (a >= b) ? PC = target	newPC = PC + 2 PC = newPC inst = Mem[PC] a = Reg[CA] b = SE(inst[8:5]) imm = SE(inst[15:9]) target = PC + imm if (a < b) ? PC = target	
L	and	or	xor	slt
	newPC = PC + 2 PC = newPC inst = Mem[PC] a = Reg[CA] b = Reg[inst[8:5]] result = a & b Reg[CA] = result	newPC = PC + 2 PC = newPC inst = Mem[PC] a = Reg[CA] b = Reg[inst[8:5]] result = a | b Reg[CA] = result	newPC = PC + 2 PC = newPC inst = Mem[PC] a = Reg[CA] b = Reg[inst[8:5]] result = $a \oplus b$ Reg[CA] = result	newPC = PC + 2 PC = newPC inst = Mem[PC] a = Reg[CA] b = Reg[inst[8:5]] result = a < b ? 1 : 0 Reg[CA] = result
LI	andi	ori	xori	slti
	newPC = PC + 2 PC = newPC inst = Mem[PC] a = Reg[CA] b = SE([inst[12:5]) result = a & b Reg[CA] = result	newPC = PC + 2 PC = newPC inst = Mem[PC] a = Reg[CA] b = SE([inst[12:5]) result = a | b Reg[CA] = result	newPC = PC + 2 PC = newPC inst = Mem[PC] a = Reg[CA] b = SE([inst[12:5]) result = $a \oplus b$ Reg[CA] = result	newPC = PC + 2 PC = newPC inst = Mem[PC] a = Reg[CA] b = SE([inst[12:5]) result = a < b ? 1 : 0 Reg[CA] = result

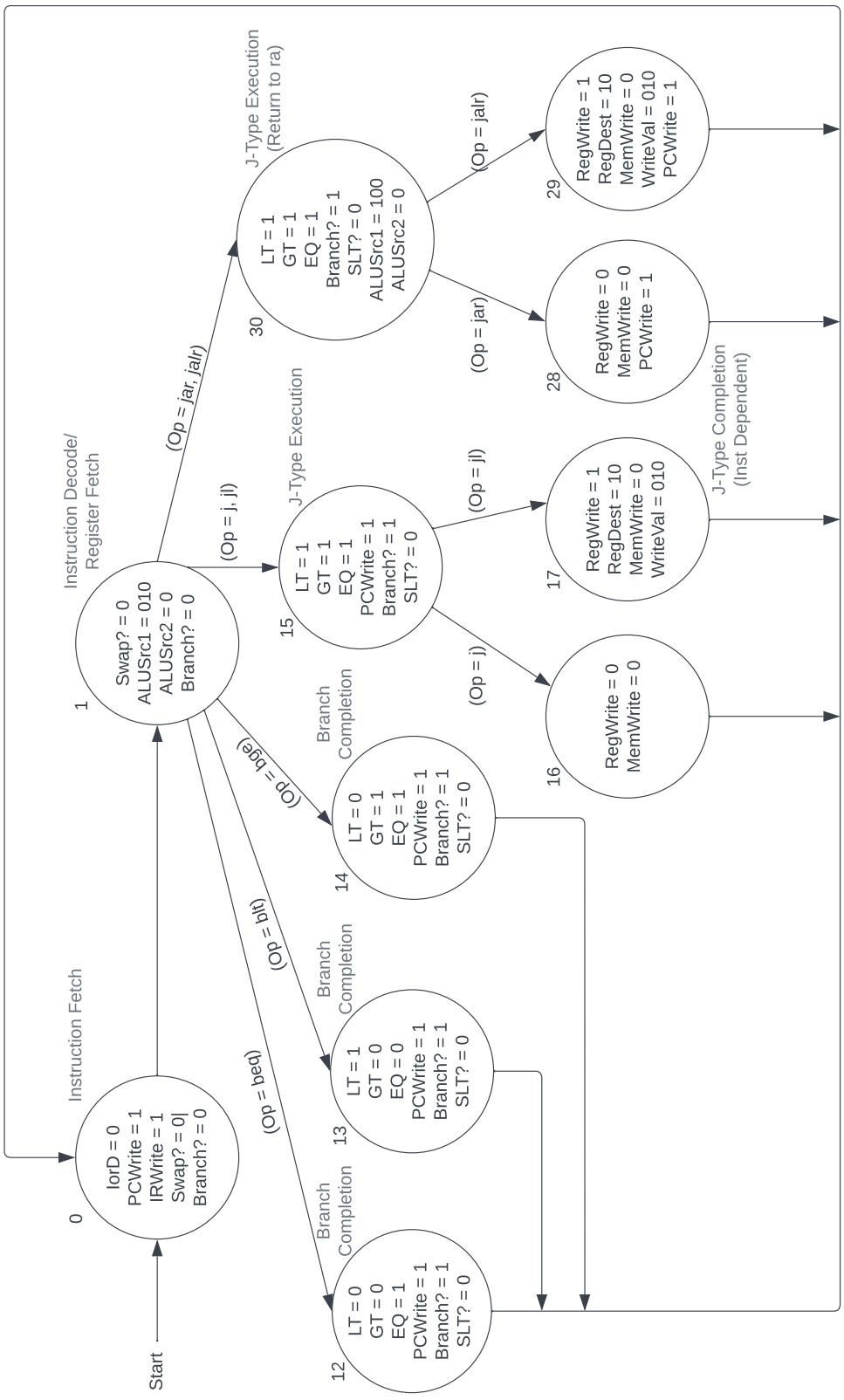
Instruction Format	Instruction Name	Instruction Name	Instruction Name	Instruction Name
C	swap	nop		
	newPC = PC + 2 PC = newPC inst = Mem[PC] b = SE([inst[12:5]) CA = b	newPC = PC + 2 PC = newPC inst = Mem[PC]		
J	jar	j		
	newPC = PC + 2 PC = newPC inst = Mem[PC] b = SE(inst[15:5]) result = PC + b Mem[ra] = PC PC = result	newPC = PC + 2 PC = newPC inst = Mem[PC] b = SE(inst[15:5]) result = PC + b PC = result		

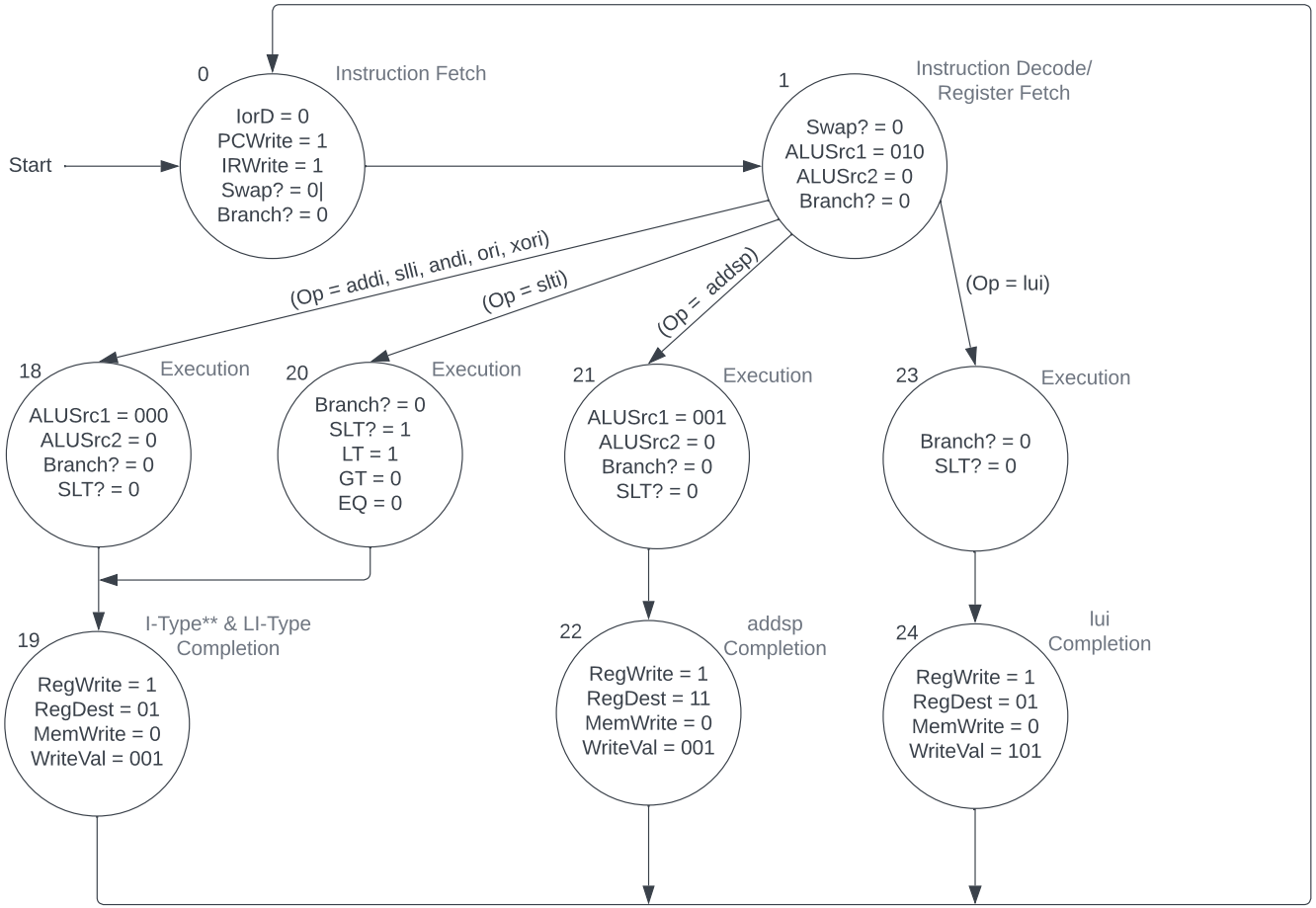
Instruction	Format Type	Opcode	F2	IorD	MemRead	MemWrite	RegWrite	RegDest
add	A	001	00	0	0	0	1	01
sub	A	001	01	0	0	0	1	01
blastOff	A	001	10	0	0	0	1	00
blastOn	A	001	11	0	0	0	1	01
addi	I	010	00	0	0	0	1	01
slli	I	010	01	0	0	0	1	01
lui	I	010	10	0	0	0	1	01
addsp	I	010	11	0	0	0	1	11
lw	M	011	00	0/1	1	0	1	01
sw	M	011	01	0	0	1	0	X
jar	J	100	00	0	0	0	1	10
j	J	100	01	0	0	0	0	00
jl	J	100	10	0	0	0	0	00
jarl	J	100	11	0	0	0	1	10
beq	B	101	00	0	0	0	0	X
blt	B	101	01	0	0	0	0	X
bge	B	101	10	0	0	0	0	X
or	L	110	00	0	0	0	1	01
and	L	110	01	0	0	0	1	01
xor	L	110	10	0	0	0	1	01
slt	L	110	11	0	0	0	1	01
ori	LI	111	00	0	0	0	1	01
andi	LI	111	01	0	0	0	1	01
xori	LI	111	10	0	0	0	1	01
slti	LI	111	11	0	0	0	1	01
swap	C	000	00	0	0	0	0	X
nop	C	000	01	0	0	0	0	X

Instruction	GT	LT	EQ	Swap?	Branch?	SLT?	ALUOp	(op)	ALUSrc1	ALUSrc2	WriteVal
add	X	X	X	0	0	0	0000	add	00	1	001
sub	X	X	X	0	0	0	0001	sub	00	1	001
blastOff	X	X	X	0	0	0	X	X	X	X	000
blastOn	X	X	X	0	0	0	X	X	X	X	011
addi	X	X	X	0	0	0	0000	add	00	0	001
slli	X	X	X	0	0	0	0010	sll	00	0	001
lui	X	X	X	0	0	0	X	X	X	X	101
addsp	X	X	X	0	0	0	0000	add	01	0	001
lw	X	X	X	0	0	0	0000	add	11	0	100
sw	X	X	X	0	0	0	0000	add	11	0	X
jar	1	1	1	0	1	0	0000	add	10	0	010
j	1	1	1	0	1	0	0000	add	10	0	X
jl	1	1	1	0	1	0	0000	add	10	0	X
jarl	1	1	1	0	1	0	0000	add	10	0	010
beq	0	0	1	0	1	0	0000	add	10	0	X
blt	0	1	0	0	1	0	0000	add	10	0	X
bge	1	0	1	0	1	0	0000	add	10	0	X
or	X	X	X	0	0	0	0100	or	00	1	001
and	X	X	X	0	0	0	0011	and	00	1	001
xor	X	X	X	0	0	0	0101	xor	00	1	001
slt	0	1	0	0	0	1	X	X	X	X	001
ori	X	X	X	0	0	0	0100	or	00	0	001
andi	X	X	X	0	0	0	0011	and	00	0	001
xori	X	X	X	0	0	0	0101	xor	00	0	001
slti	0	1	0	0	0	1	X	X	X	X	001
swap	X	X	X	1	0	0	X	X	X	X	101
nop	X	X	X	0	0	0	X	X	X	X	X

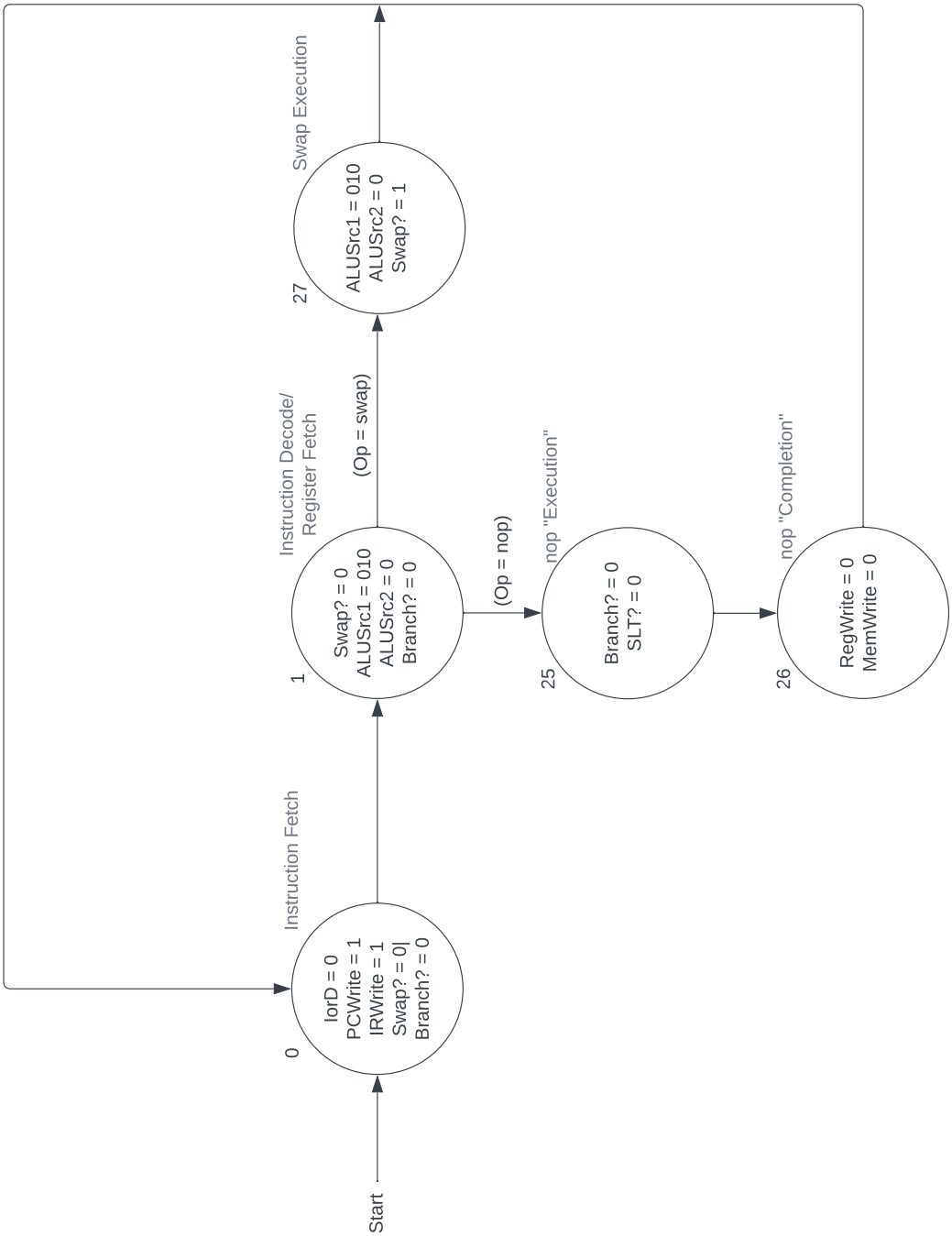


Note: Particular control signals will default to 0 after every state change. These signals are: PCWrite, IRWrite, RegWrite, MemWrite


Start
0
Instruction Fetch
IorD = 0
PCWrite = 1
IRWrite = 1
Swap? = 0|
Branch? = 0
1
Instruction Decode/
Register Fetch
Swap? = 0
ALUSrc1 = 010
ALUSrc2 = 0
Branch? = 0
(Op = jar, jalr)
(Op = j, jl)
(Op = bge)
(Op = blt)
(Op = beq)
12
Branch
Completion
LT = 0
GT = 0
EQ = 1
PCWrite = 1
Branch? = 1
SLT? = 0
13
Branch
Completion
LT = 1
GT = 0
EQ = 0
PCWrite = 1
Branch? = 1
SLT? = 0
14
Branch
Completion
LT = 0
GT = 1
EQ = 1
PCWrite = 1
Branch? = 1
SLT? = 0
15
J-Type Execution
LT = 1
GT = 1
EQ = 1
PCWrite = 1
Branch? = 1
SLT? = 0
16
RegWrite = 0
MemWrite = 0
17
J-Type Completion
(Inst Dependent)
RegWrite = 1
RegDest = 10
MemWrite = 0
WriteVal = 010
30
J-Type Execution
(Return to ra)
LT = 1
GT = 1
EQ = 1
Branch? = 1
SLT? = 0
ALUSrc1 = 100
ALUSrc2 = 0
(Op = jalr)
(Op = jar)
28
RegWrite = 0
MemWrite = 0
PCWrite = 1
29
RegWrite = 1
RegDest = 10
MemWrite = 0
WriteVal = 010
PCWrite = 1
(Op = jl)
(Op = j)



* - ALUOp is determined by the specific instruction in ALU Control. Simplified here for clarify.
** - lui, slti, and addsp are exempt from this state.


Start
0
Instruction Fetch
IorD = 0
PCWrite = 1
IRWrite = 1
Swap? = 0|
Branch? = 0
1
Instruction Decode/
Register Fetch
Swap? = 0
ALUSrc1 = 010
ALUSrc2 = 0
Branch? = 0
(Op = swap)
27
Swap Execution
ALUSrc1 = 010
ALUSrc2 = 0
Swap? = 1
(Op = nop)
25
nop "Execution"
Branch? = 0
SLT? = 0
26
nop "Completion"
RegWrite = 0
MemWrite = 0

Accumulator ISA Reference Data

Michael Donaghy, Braedyn Edwards,
Emily Hart, Liam Hill, Logan Manthey
November 15, 2022

1 Base Instructions in alphabetical order

MNEMONIC	FMT	NAME	VERILOG DESCRIPTION
add	A	Add	R[ca] = R[rs1] + R[ca]
addi	I	Add Immediate	R[ca] = imm + R[ca]
addsp	I	Add Stack Pointer	sp = R[sp] + imm
and	L	AND	R[ca] = R[rs1] & R[ca]
andi	LI	AND Immediate	R[ca] = imm & R[ca]
beq	B	Branch Equal	if(R[ca] == R[rs1]) PC=PC+imm
bge	B	Branch Greater Than Equal	if(R[ca] R[rs1]) PC=PC+imm
blt	B	Branch Less Than	if(R[ca] < R[rs1]) PC=PC+imm
jar	J	Jump and Return	PC = Reg[ra] + imm
j	J	Jump	PC = PC + imm + label
jarl	J	Jump and Return Immediate	PC = Reg[ra] + imm Reg[ra] = oldPc
jl	J	Jump Immediate	Reg[ra] = PC PC = PC + imm + label
lui	I	Load Upper Immediate	ca = imm[15:08]
lw	M	Load Word	ca = MEM[rs1] + imm(6:0)
nop	C	No Operation	N/A
or	L	Or	ca = ca | R[s1]
ori	LI	OR with Immediate	ca = ca | imm
blastOff	A	Blast Off	R[rs1] = ca
blastOn	A	Blast On	ca = R[rs1]
slli	I	Shift Left Immediate	ca = ca « imm
slt	L	Set Less Than	ca = (R[ca] < R[rs1]) ? 1 : 0
slti	LI	Set Less Than Immediate	if(imm< R[rs1]) ca = 1
sub	A	Subtract	ca = ca - R[rs1]
sw	M	Store Word	MEM[rs1] + imm(6:0) = ca
swap	C	Swap Current Accumulator	ca = *R[rs1]
xor	L	XOR	ca = R[rs1] ĉa
xori	LI	XOR Immediate	ca = R[rs1] îmm

2 Register Name, Use, Calling Convention

REGISTER	NAME	USE	SAVER
x0	zero	Zero	N.A
x1	sp	Stack Pointer	Callee
x2	ra	Return Address	Caller
x3-x6	a0-a3	Accumulators	Caller
x7-x8	p0-p1	Function Args/ Return Type	Caller
x9-x14	p2-p7	Func Args	Caller
x15	ca	Current Accum.	—

3 Core Instruction Formats

	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00 [b]
A	—								rs1			f2			Opcode	
I	Immediate											f2			Opcode	
M	Immediate								rs1			f2			Opcode	
J	Immediate											f2			Opcode	
B	Immediate								rs1			f2			Opcode	
L	—								rs1			f2			Opcode	
LI	—			Immediate								f2			Opcode	
C	–								rs1			f2			Opcode	

4 Opcodes in Numerical Order By Opcode

MNEMONIC	FMT	OPCODE	FUNCT2
add	A	001	00
sub	A	001	01
blastOff	A	001	10
blastOn	A	001	11
addi	I	010	00
slli	I	010	01
lui	I	010	10
addsp	I	010	11
lw	M	011	00
sw	M	011	01
jar	J	100	00
j	J	100	01
jl	J	100	10
jarl	J	100	11
beq	B	101	00
bge	B	101	01
blt	B	101	10
or	L	110	00
and	L	110	01
xor	L	110	10
slt	L	110	11
ori	LI	111	00
andi	LI	111	01
oxri	LI	111	10
slti	LI	111	11
swap	C	000	00
nop	C	000	01