Design Documentation

Super Awesome Processor A (SAPA) v1.0

Team 1A - Khaled Alfayez, Shaun Davis,
Trinity Merrell, and Logan Smith

CONTENTS:

Register Descriptions

Table of Register Descriptions

Register	Number	Availability	Description
$s*n* *(0-2)* <(^.^)>	0-2	Read/write	General purpose registers. The intent is to store long term computation results and save values over functions calls
$t*n* *(0-3)* <(-_-,)>	3-6	Read/write	General purpose registers, to be used like $s*n* registers. Will NOT be saved over function calls.
$v <(-_-,)>	7	Read/write	This register is for storing the return value from a function
$cr <(-_-,)>	8	Read only*	Accumulator - stores most recently computed value
$ra <(^.^)>	9	Read/write	Stores the return address of a function
$a*n* (0-1) <(-_-,)>	10-11	Read/write	These registers are used to store arguments for use in a called function
$st <(^.^)>	12	Read/write	Reference “top” or lowest memory address of stack
$i*n* (0-1) :~/	13-14	Not available	Used by assembler for pseudo instructions.
$k :~/	15	Read/write (while handling exceptions)	Exception registers; only accessible with permissions*

Unsafe between procedure calls = <(-_-,)> **Safe between procedure calls = <(^_^)>**

*** Enforced by exception handler**

Instructions - Assembly and Machine Language

Procedure Call Conventions:

1. Registers $i*n*, $k*n* (where n is 0-1) are reserved for the assembler and operating system and should not be used by user programs or compilers.
2. Registers $a*n* (where n is 0-1) are used to pass arguments to procedures, any other arguments should go on the stack. Register $v is used to return a value from functions.
3. Registers $a*n*, $t*n*, $d, $cr, and $v are temporary and volatile. Expect them to contain different data after a procedure call.
4. $s*n* registers must be backed up on the stack at the beginning of a procedure and restored before returning from the procedure. This preserves values in these registers over procedure calls.
5. $st is the stack register. It points to the top memory location in the stack. If the stack is grown at any time in a procedure, it must be reduced before returning from that procedure.
    a. Memory is allocated to the stack by subtracting from the value in $st. Memory is deallocated from the stack by adding to the value in $st.
6. $ra is the return address of a procedure. Jal will overwrite $ra to be the next instruction, so $ra must ALWAYS be backed up on the stack before a procedure call and restored after returning from the procedure.
7. The instruction jr $ra will return the program to the address.

Syntax and Semantics:

Basic Instruction Formats

C-type, Computation types (register to register)

C-type instructions are used for register to register computations. They handle arithmetic and logical computations such as add, sub, and, or, etc. These instructions share a single opcode, and are distinguishable by their func code.

I-type, Immediate types (register to data, register to memory)

opcode	r1	immediate
15	11	7 0

I-type instructions are used for register to data and register to memory computations. These instructions handle storing data from registers into memory and loading data into registers from memory, immediate values, and the $cr register. They also handle control flow such as branches and jumps that are necessary for loops and procedure calls.

L-type, Leap type

opcode	immediate
15	11 0

L-type instructions are used for jal and j instructions.

Arithmetic and Logical Instructions

Arithmetic and logical instructions are **C-type** instructions. These instructions take two registers as operands to their computations. Their results are always stored in the specialized $cr register.

Arithmetic and logical instructions are directly translated from their assembly to the machine language. For example, the following assembly would translate accordingly into binary:

add $s2, $t3

0000	0010	0111	0000
op	r1	r2	func

Registers are directly translated from their respective numbers in the registry (see chart in Registers).

List of Arithmetic and Logical Instructions:

Addition:

add r1, r2

0000	r1	r2	0000
4	4	4	4

Stores the sum of r1 and r2 into register $cr

Subtraction:

sub r1, r2

0000	r1	r2	0001
4	4	4	4

Stores the difference between r1 and r2 into register $cr

AND:

and r1, r2

0000	r1	r2	0010
4	4	4	4

Stores the logical AND of r1 and r2 into register $cr

OR:

or r1, r2

0000	r1	r2	0011
4	4	4	4

Stores the logical OR of r1 and r2 into register $cr

NOR:

nor r1 ,r2

0000	r1	r2	0100
4	4	4	4

Stores the logical NOR of r1 and r2 into register $cr

NAND:

nand r1, r2

0000	r1	r2	0101
4	4	4	4

Stores the logical NAND of r1 and r2 into register $cr

Exclusive OR:

xor r1, r2

0000	r1	r2	0110
4	4	4	4

Stores the logical Exclusive OR of r1 and r2 into register $cr

Set Less Than:

slt r1, r2

0000	r1	r2	0111
4	4	4	4

Stores either a 1(True) or a 0(False) in $cr depending on if r1 is less than r2

Branch Instructions

Branch instructions are **I-type** instructions, and compare the value in $cr to the value in r1. The branch instruction will then succeed or fail based on equivalence or inequality.

Branch instructions are PC-relative, meaning they use an 8-bit offset that allows a user to jump 2^7-1 instructions forward or 2^7 backward. A translation may appear as below (where “loop” is 3 instructions above bieq):

bieq $s0, loop

0001	0000	1101
op	r1	BranchAddr

List of Branch Instructions:

Branch if equal:

bieq r1, location

0001	r1	BranchAddr
4	4	8

Conditional branch to address in immediate if r1 is equal to register $cr

Branch not equal:

bneq r1, location

0010	r1	BranchAddr
4	4	8

Conditional branch to address in immediate if r1 does not equal register $cr

Jump Instructions

Jump instructions utilize two instructions formats: **L-type** and **I-type.**

Jump instructions use a 12-bit immediate with the top 4 bits of the PC concatenated to create a 16-bit address and allowing for a 2^{11} size block of “jump space”. PC is set to this new address.

List of Jump Instructions:

Jump:

j location

0011	JumpAddr
4	12

Unconditional jump to the address in immediate

Jump and link:

jal location

0100	JumpAddr
4	12

Unconditional jump to the address in immediate, storing the address of subsequent instruction into register $ra

Jump register:

jr r1

0101	r1	unused
4	4	8

Unconditional jump to the address in r1

Load/Store Instructions

Load/Store instructions are **I-type** and often require a value to be computed and stored in $cr prior to execution. Specific requirements are denoted for each instruction.

Load/store instructions are directly translated similar to arithmetic and logical instructions. The following assembly would translate accordingly:

lw $t0

1010	0101	unused
op	r1	Imm

List of Load/Store Instructions:

Load upper immediate:

lui r1, upper(big)

0110	r1	Immediate
4	4	8

Load top half of 16-bit immediate into r1. Bottom bits are set to zero.

Load lower immediate:

lli r1, lower(big)

0111	r1	Immediate
4	4	8

Load lower half of 16-bit immediate into r1. Top bits are set to sign of immediate.

Load to register:

ltr r1, immediate

1000	r1	SignExtImm
4	4	8

Take a sign-extended 8-bit immediate value and store in r1. If immediate is greater than 8-bits, utilizes load upper immediate and load lower immediate.

ltr r1, big

Translates to:

lui $i0, upper(big)
lli $i1, lower(big)
or $i0, $i1
ctr r1

Copy to register:

ctr r1

1001	r1	unused

4	4	8

Take a previously computed value from $cr and store in r1

Load word:

lw r1

1010	r1	unused
4	4	8

Access memory at the address stored in $cr and store the value in r1

Store word:

sw r1

1011	r1	unused
4	4	8

Store value in r1 at the previously computed memory address in register $cr.

Special Procedures

Interact with I/O

syscall

1110	unused	unused
4	4	8

How to use syscall:
Step 1. Load appropriate code into $v.
Step 2. Load argument values, if any, into $a0 or $a1 as specified.
Step 3. Issue the SYSCALL instruction.
Step 4. Retrieve return values, if any, from result registers as specified.

Example:

```
ltr     $v0, 1      # service 1 is print integer
ltr     $a0, 0      # load desired value into argument register $a0
syscall
```

Table of Syscall Codes

Function	Integer in $v	Argument or Return Value
PRINT_INT	1	$a0 = value
PRINT_STRING	2	$a0 = address of string
READ_INT	3	Result placed in $v
READ_STRING	4	$a0 = address, $a1 = maximum length
EXIT	5	None
PRINT_CHAR	6	$a0 low byte = character
READ_CHAR	7	Character returned in low byte of $v

Pseudo Instructions

These are instructions are not official instructions but sets of instructions that our processor will handle with smaller instructions. All pseudo instructions are **I-type**. It is important to note that the expansion of pseudo instructions often require **C-type** instructions. Therefore, users should generally assume that $cr will be overwritten when executing pseudo instructions and backup $cr if they wish to use it afterward.

Set branch comparison:

sbc imm

Sets register $cr to a sign-extended 8-bit immediate value in preparation for a branch instruction.
Translation:

ltr i0, 0
ltr i1, imm
add i0, i1

Set jump register:

sjr JumpAddr

Sets register $ra to a jump address specified by the user.
Translation:

ltr i1, JumpAddr
ltr i0, 0

```
add    i0, i1
ctr    $ra
```

OR immediate:

```
ori    r1, imm
```

Stores the logical OR of r1 and immediate into register $cr. 8-bit numbers are zero-extended. For 16-bit numbers, the following translation occurs:
Translation:

```
lui    $i0, upper(big)      # Loads upper 8 bits
lli    $i0, lower(big)      # Loads upper 8 bits
or     r1, $i0
```

ADD immediate:

```
addi   r1, imm
```

Stores the arithmetic result of r1 and immediate into register $cr. 8-bit numbers are zero-extended. For 16-bit numbers, the following translation occurs:
Translation:

```
lui    $i0, upper(big)
lli    $i0, lower(big)
add    r1, $i0
```

Load address:

```
la     r1, address
```

Stores the sign-extended address in immediate into r1.
Translation:

```
lui    $i0, upper(address)
lli    $i1, lower(address)
or     $i0, $i1
ctr    r1
```

Register to Register:

```
rtr    r1, r2
```

Moves the value in r2 to r1.
Translation:

```
ltr    $i0, 0
```

add $r2, $i0
ctr r1

Table of Machine Language Translations for Instructions

Key: R[8] = $cr, R[9] = $ra

Instruction	Type	Code	Description of bits and rules			
Add	C	R[8] = R[r1] + R[r2]	0000	r1	r2	0000
Sub	C	R[8] = R[r1] - R[r2]	0000	r1	r2	0001
AND	C	R[8] = R[r1] & R[r2]	0000	r1	r2	0010
OR	C	R[8] = R[r1] | R[r2]	0000	r1	r2	0011
NOR	C	R[8] = ~(R[r1] | R[r2])	0000	r1	r2	0100
NAND	C	R[8] = ~(R[r1] & R[r2])	0000	r1	r2	0101
XOR	C	R[8] = R[r1] ^ R[r2]	0000	r1	r2	0110
Set Less Than	C	R[8] = (R[r1] < R[r2]) ? 1 : 0	0000	r1	r2	0111
Branch Equal	I	If (R[8] = R[r1]) PC = PC + 1 + BranchAddr	0001	r1	BranchAddr	
Branch Not Equal	I	If (R[8] != R[r1]) PC = PC + 1 + BranchAddr	0010	r1	BranchAddr	
Jump	L	PC = JumpAddr	0011	JumpAddr		
Jump and Link	L	R[9] = PC + 1 PC = JumpAddr	0100	JumpAddr		
Jump Register	I	PC = R[r1]	0101	r1	8’b0	
Load Upper Immediate	I	R[r1] = {imm, 8’b0}	0110	r1	Immediate	

Load Lower Immediate	I	R[r1] = {8'b0, imm}	0111	r1	Immediate
Load to Register	I	R[r1] = SignExtImm	1000	r1	Immediate
Copy to Register	I	R[r1] = R[8]	1001	r1	8'b0
Load Word	I	R[r1] = M[R[8]]	1010	r1	8'b0
Store Word	I	M[R[8]] = R[r1]	1011	r1	8'b0
syscall	I	I/O	1110	4'b0	8'b0
Set Branch Comparison	-	R[8] = SE(BranchComparison)	*Pseudo instruction*		
Set Jump Register	-	R[9] = JumpAddr	*Pseudo instruction*		
ORi	-	R[8] = r1 | ZeroExtImm	*Pseudo instruction*		
Load Address	-	R[r1] = SignExtImm	*Pseudo instruction*		
Register to Register	-	R[r1] = R[r2]	*Pseudo instruction*		
Addi	-	R[r1] = SignExtImm	*Pseudo instruction*		

Table of Common Operations:

	SAPA 1.0	Description
Load Address	*la $s2, 0x4EF6*	Loading an address is a pseudo instruction. Refer to *Pseudo instruction* for full details on how *la* works.
Arithmetic / Logical	*ltr $t1, 1* *ltr $t2, 1* *add $t1, $t2*	Load values into two registers and call add to store their sum in $cr. Similar for other arithmetic and logical operations.

Iterations	… ltr $s1, 15 ltr $t0, 1 ltr $t1, 0 ltr $t2, 3 loop: slt $s1, $t1 bieq $t0, exit add $t1, $t2 ctr $t1 j loop exit: …	This *for-loop* keeps adding 3 to register $s3 (0) until $t1 becomes greater than $s1 (15), resulting with a 18 stored in $t1 after the loop has exited.
Branches	*See above*	The example above utilizes *slt* and *bieq* to create a branch on greater than, which isn't an instruction itself but can be created using multiple instructions.
Jumps	… add2: ltr $t0, 2 add $t0, $a0 ctr $t1 sbc 20 bneq $t1, add2 sjr end jr $ra … end: …	This example of code does recursive addition of adding two until the value of 20 is reached. However, it needs to move to end instead of moving to the next set of recursive code. Unfortunately, it is too far for the regular jump code to reach. The user can instead; however, set the value of the desired address to the $ra register and can now jump to it.

relPrime and Euclid's Algorithm (Assembly):

```
relPrimeSetup:
        ltr     $t0 -4
        add     $st $t0         # current value of $st - 4
        ctr     $st             # grow stack by 4
        sw      $ra             # store $ra on stack
        add     $st $t0         # current value of $st - 4
        ctr     $st             # grow stack by 4
        sw      $s0             # store $s0 on stack
        ltr     $t1 2           # m = 2
```

```
        rtr     $a1 $t1         # m in $a1
        rtr     $s0 $a1         # m in $s0

relPrimeLoop:
        jal     gcd
        sbc     1               # store 1 in $cr for branch
        bieq    $v cleanup      # if gcd(n, m) == 1 jump to cleanup

        ltr     $t0 1
        add     $s0 $t0         # m + 1
        ctr     $s0             # m = m + 1
        rtr     $a1 $s0         # $a1 = m
        j       relPrimeLoop

gcd:
        sbc     0               # store 0 in $cr for branch
        bneq    $a0 subOne      # if a != 0, go to subOne
        rtr     $v $a1          # $v = m
        jr      $ra             # return to caller

subOne:
        slt     $a1 $a0         # b < a
        ltr     $t0 1
        bieq    $t0 subTwo      # if (a > b) go to subTwo
        sub     $a1 $a0         # b - a
        ctr     $a1             # b = b - a
        rtr     $v $a0          # return a
        jr      $ra

subTwo:
        sub     $a0 $a1         # a - b
        ctr     $a0             # a = a- b
        rtr     $v  $a0         # return a
        jr      $ra

cleanup:
        rtr     $v $a1          # return m
        ltr     $t0 0
        add     $st $t0         # moves stack pointer address to $cr
```

```
lw      $s0         # restore #s0
ltr     $t0 4
add     $st $t0     # current value of $st + 4
ctr     $st         # reduce stack
lw      $ra         # restore $ra
add     $st $t0     # current value of $st + 4
ctr     $st         # reduce stack
jr      $ra
```

relPrime and Euclid's Algorithm (Machine Language):

relPrimeSetup:

ltr $t0 -4

1000	0011	11111100

add $st $t0

0000	1100	0011	0000

ctr $st

1001	1100	00000000

sw $ra

1011	1001	00000000

add $st $t0

0000	1100	0011	0000

ctr $st

1001	1100	00000000

sw $s0

1011	0000	00000000

ltr $t1 2 # m = 2

1000	0100	00000010

rtr $a1 $t1 # m in $a1

1000	1101	00000000

0000	0100	1101	0000

1001	1011	00000000

rtr $s0 $a1 # m in $s0

1000	1101	00000000

0000	1010	1101	0000

1001	0000	00000000

relPrimeLoop:

jal gcd

0100	000000010010

sbc

1000	1101	00000000

1000	1110	00000001

0000	1101	1110	0000

bieq $v cleanup # if gcd(n, m) == 1 jump to cleanup

0001	1011	00100001

 # body of while loop
ltr $t0 1

1000	0011	00000001

add $s0 $t0 *# m + 1*

0000	0000	0011	0000

ctr $s0 *# m = m + 1*

0111	0000	00000000

rtr $a1 $s0

1000	1101	00000000

0000	0000	1101	0000

0111	1010	00000000

j relPrimeLoop

0011	000000001010

gcd:

sbc 0

1000	1101	00000000

1000	1110	00000000

0000	1101	1110	0000

bneq $a0 subOne # if a != 0, go to subOne

0010	1001	00010110

rtr $v $a1

1000	1101	00000000

0000	1010	1101	0000

0111	1011	00000000

jr *$ra* *# return to loop*

1010	1000	00000000

subOne:

slt *$a1 $a0*

0000	1010	1001	0111

ltr *$t0 1*

1000	0011	00000001

bieq *$t0 subTwo* *# if (a > b) go to subTwo*

0001	0011	00011101

sub *$a1 $a0* *# b - a*

0000	1010	1001	0001

ctr *$a1* *# b = b - a*

0111	1010	00000000

rtr *$v $a0* *# return a*

1000	1101	00000000

0000	1001	1101	0000

0111	1011	00000000

jr *$ra*

0101	1000	00000000

subTwo:

sub *$a0 $a1* *# a - b*

0000	1001	1010	0001

ctr $a0 # a = a- b

0111	1001	00000000

rtr $v $a0 # return a

1000	1101	00000000

0000	1001	1101	0000

0111	1011	00000000

jr $ra

0101	1000	0000000

cleanup:

 # restore $s0 from stack
rtr $v $a1 # return m

1000	1101	00000000

0000	1010	1101	0000

0111	1011	00000000

ltr $t0 0

1000	0011	00000000

add $st $t0

0000	1100	0011	0000

lw $s0

1010	0000	00000000

ltr $t0 4

1000	0011	00000100

add $st $t0

0000	1100	0011	0000

ctr $st

1001	1100	00000000

lw $ra

1010	1000	00000000

add $st $t0

0000	1100	0011	0000

ctr $st

1001	1100	00000000

jr $ra

0101	1000	00000000

Register Transfer Language (RTL)

Definitions:

CR = Register[8]
RA = Register[9]
I0 = Register[13]
I1 = Register[14]
R1 = Register[IR[11-8]]
R2 = Register[IR[7-4]]

OP = IR[15-12]
IMM8 = IR[7-0]
IMM12 = IR[11-0]
IMM16 = 16-bit immediate
IR = Instruction Register
MDR = Memory Data Register

Common RTL:

Cycle	Label	RTL
0	FETCH INSTR:	PC = PC + 1 IR = Mem[PC]
1	DECODE:	A = R1 B = R2 C = CR ALUout = PC + SE(IMM8)
Last	DONE:	Goto FETCH INSTR

Unique RTL:

Instruction **-**	**Type** **C**	**Op** **0000**	**func** **0000 - 0111**
Cycle	**Label**	**RTL**	
2	OPERATION:	ALUout = A op B	
3		CR = ALUout	

Instruction *Branch if equal / not equal*	**Type** **I**	**Op** **0001 / 0010**	**func** **-**
Cycle	**Label**	**RTL**	

2	BIEQ:
	
	BNEQ:
		if (A == C) then
		PC = ALUout
		
		if (A != C) then
		PC = ALUout	

3	STALL		

Instruction *Load upper / lower Imm.*	**Type** **I**	**Op** **0110 / 0111**	**func** **-**
Cycle	**Label**	**RTL**	
2	LUI: LLI:	R1 = IMM8[8-0] << 8 R1 = ZE(IMM8[8-0])	

Instruction *Load to register*	**Type** **I**	**Op** **1000**	**func** **-**
Cycle	**Label**	**RTL**	

2	LTR16:
	
	LTR8:
		if IMM16 then
		I0 = IMM16[15-8] << 8
		
		I1 = ZE(IMM16[7-0])
		
		if IMM8 then
		R1 = SE(IMM8)	

3	LTR16:	ALUout = I0 | I1	

4	LTR16:	R1 = ALUout

Instruction *Copy to register*	**Type** **I**	**Op** **1001**	**func** **-**
Cycle	**Label**	**RTL**	
2	CTR:	R1 = C	

Instruction *Load word*	**Type** **I**	**Op** **1010**	**func** **-**
Cycle	**Label**	**RTL**	
2	LD FROM MEM:	MDR = Mem[C]	
3	LOAD TO REG:	R1 = MDR	
4	STALL		

Instruction *Store word*	**Type** **I**	**Op** **1011**	**func** **-**
Cycle	**Label**	**RTL**	
2	SW:	Mem[C] = A	
3	STALL		

Instruction *syscall*	**Type** **I**	**Op** **1110**	**func** **-**
Cycle	**Label**	**RTL**	
2	INCR PC:	PC = PC + 1	
?	READ:	if FINISHED then goto RDONE	

	WRITE: EXIT:	if FINISHED then goto WDONE Close program
?	RLOOP: WLOOP:	if FINISHED then goto RDONE else goto RLOOP if FINISHED then goto WDONE else goto WLOOP
?	RDONE: / WDONE:	

Instruction *Jump Register*	**Type** **I**	**Op** **0101**	**func** **-**
Cycle	**Label**	**RTL**	
2	JR:	PC = A	
3	STALL		

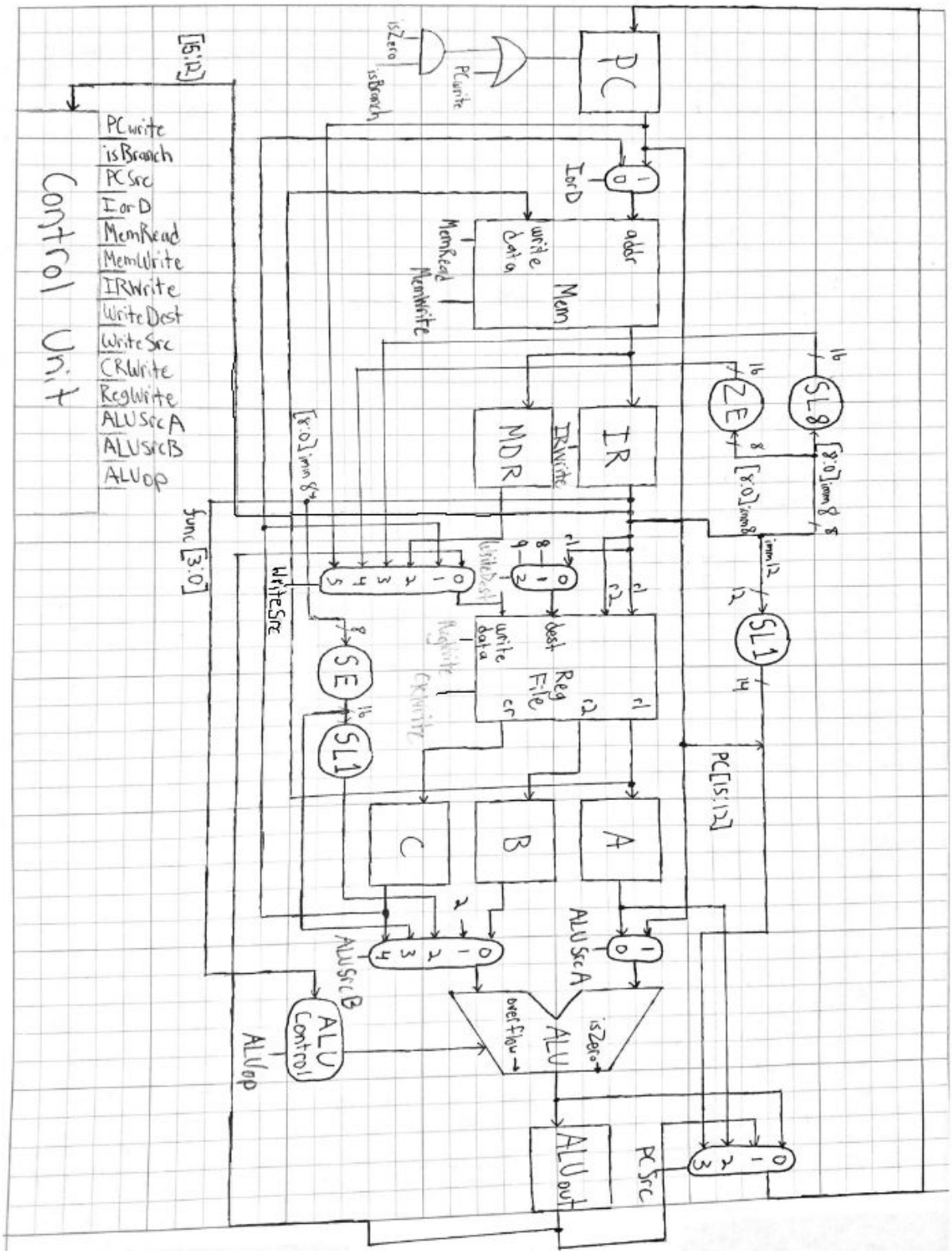
Instruction -	**Type** **L**	**Op** **0011 / 0100**	**func** **-**
Cycle	**Label**	**RTL**	
2	J: JAL:	PC = (IMM12) | PC[15-12] RA = PC PC = (IMM12) | PC[15-12]	
3	STALL		

Datapath Components

Table of Components and Signals

Signals Components	Inputs (size)	Outputs (size)	Description of Component
PC	- newPC 16'b	- PC 16'b	- PC => Program counter
Memory	- Address 16'b - Write Data 16'b	- MemData 16'b	- Access to memory
IR	- Instruction Data 16'b	- r1 4'b - r2 4'b - imm 8'b or 12'b	- IR => instruction register - Stores instruction data
MDR	- Memory Data 16'b	- Memory Data 16'b	- MDR => Memory data register - Stores memory data
Register File	- r1 4'b - r2 4'b - WriteDest 4'b - Write Data 16'b	- r1 data 16'b - r2 data 16'b	- Registers that store data for reading/writing
A/B Registers	- A or B respectively 16'b	- A or B respectively 16'b	- These registers hold operands for the ALU
ALU	- A 16'b - B 16'b	- isZero 1'b - Overflow 1'b - Operation Result 16'b	- Performs operation on two operands
Shift Left by 1	- Imm 12'b	- Shifted imm size + 1	- Shifts immediate by 1
Shift Left by 8	- Imm 8'b	- Shifted imm size + 8	- Shifts immediate by 8
Sign Extender	- Imm 8'b	- Sign Extended imm 16'b	- Fill in top bits of immediate with sign of immediate
Zero Extender	- Imm 8'b	- Zero extended imm 16'b	- Fill in top bits of immediate with 0s

Block Diagram of Datapath



Control Unit

Table of Control Signals

Control Signal	Size	Component	Description
PCSrc	2'b	PC Source (*Mux*)	Controls the source of PC: incremented PC, jump address, or branch address.
PCWrite	1'b	PC	Controls whether PC should be written to
isZero	1'b	PC	ALU isZero result **Branch condition*
isBranch	1'b	PC	Controls whether an instruction is a branch **Branch condition*
isBIEQ	1'b	ALU	Determines whether an branch instructions is a bieq
IorD	1'b	MemAddr (*Mux*)	Decides whether a memory address is coming from PC (instruction) or the C register (data)
MemRead	1'b	Memory	Controls whether memory is being read
MemWrite	1'b	Memory	Controls whether memory is being written to
IRWrite	1'b	IR	Controls whether IR is being written to
WriteDest	2'b	RegDest (*Mux*)	Controls whether destination of a register write is $cr or r1
WriteSrc	3'b	WriteData (*Mux*)	Controls whether data source of a register write is MDR, ALUout, C, or one of two immediates
CRWrite	1'b	Register File	Controls whether the $cr register can be written to (ie disable $cr being set as r1 in **I-type**

			instructions)
RegWrite	1’b	Register File	Controls whether a register is being written to
ALUSrcA	1’b	ALUSrcA (*Mux*)	Controls whether the source of ALU input A is PC or the A register
ALUSrcB	2’b	ALUSrcB (*Mux*)	Controls whether the source of ALU input B is one of three immediates or the B register
ALUop	2’b	ALUControl	Controls the operation sent to the ALUControl component

Specification for Controls:

PCSrc

00 - PC + 1
01 - ALUout
10 - Register A
11 - (PC[15-12] | IMM12)

PCWrite

0 - turns off; 1 - turns on

isBranch

0 - not branch
1 - is branch

isBIEQ

0 - is bieq instruction
1 - is not bieq instruction

IorD

0 - C register (data)
1 - PC (instruction)

MemRead

0 - turns off; 1 - turns on

MemWrite

0 - turns off; 1 - turns on

IRWrite

0 - turns off; 1 - turns on

WriteDest

00 - r1
01 - $cr
10 - $ra

WriteSrc

000 - ALUout
001 - Register C
010 - MDR
011 - (IMM8 << 8)
100 - ZE(IMM8)
101 - PC
110 - SE(IMM8)

CRWrite

0 - turns off; 1 - turns on

RegWrite

0 - turns off; 1 - turns on

ALUSrcA

0 - Register A
1 - PC

ALUSrcB

00 - Register B
01 - 1
10 - SE(IMM8)
11 - Register C

ALUop

00 - Add (PC increment)
01 - Sub (Branch)
10 - function code (C-type)
11 - OR

func

4’b function determined by C-type

opcode

Determined by instruction

current_state/next_state

Keeps track of state in multi-cycle

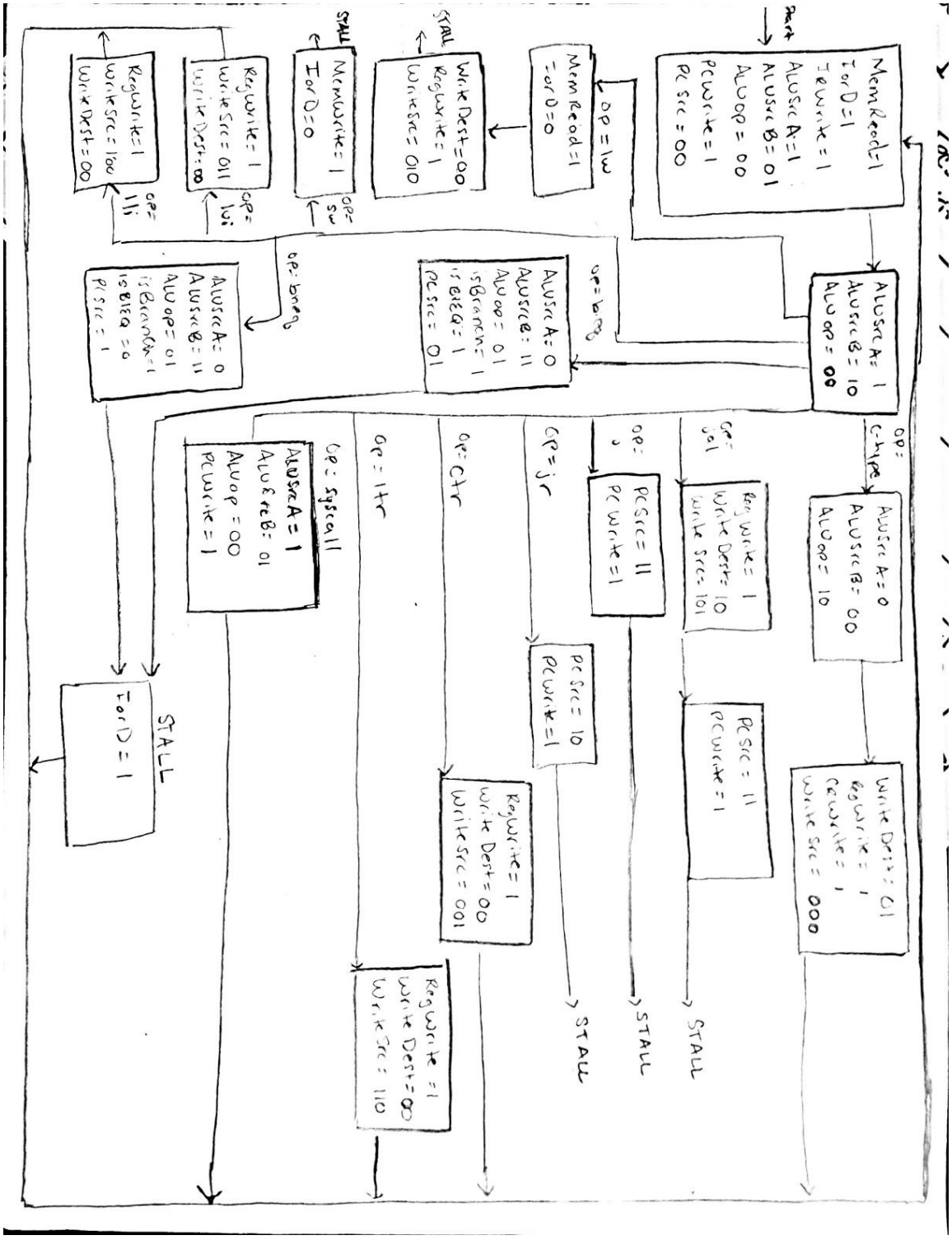
CLK

Clock speed

Reset

0 - Don’t Reset; 1 - Reset

Finite State Machine Transition Diagram



Implementation and Testing

Hardware Implementation Plan:

Special Registers (PC, IR, MDR, A, B, C):

- Each register will be able to hold 16-bits and have one output with as much.
- They will have multiple inputs such as the clock signal, a reset signal, and an input for the data.
- These will be made with a Verilog module in Xilinx.

Memory:

- For each bit, use a flip flop.
- Use buses and muxes to combine the data.
- Built as a schematic using these parts in Xilinx.

Register File:

- The register file will be made of registers and parts for the logic required to choose the correct outputs.
- 4'b address input and value output
- The cr register will always be outputted to a special C register.
- The register file will be modified from the resource provided to us to handle 16 registers.

ALU:

- For a 16-bit ALU, we will add a unit for Arithmetic (adds and subs), a unit for Logic (OR, NOR, XOR, AND, NAND), and a unit for less than (slt).
- Each unit will have a 16-bit input and output with their outputs connected to a mux.
- The mux chooses which data to output of the ALU, based on the ALU control signal.
- The ALU can be built in Xilinx using schematics, a modified design from the 4-bit ALU built in Lab 6.

Sign Extender:

- A simple verilog module that takes the most significant bit of the input and extends it to the left, extending the size of the data to 16 bits.

Zero Extender:

- Another verilog module that fills top bits with zeros, extending the size of the data to 16 bits.

Shifters:

- Verilog module that will take input and shift the bits to certain amount.

Muxes (1'b, 2'b, 3'b, 4'b)

- Modify resources given to us

Unit Testing:

General Testing:

- Consider instruction description including inputs and outputs
- Draw out the path of data flow from one component to another
- Confirm desired data source input and output for each component
    - Upon error return to last step and debug
    - Repeat until problem is found
- Confirm desired result/action at the end of calculation

Regular and Special Registers (A, B, C, IR, MDR):

Overall, all of these registers have one thing that they need to accomplish. They need to be able to take in values, hold them and then output that same value on the output for that register.

Memory:

The memory needs to able to hold the instructions that we will be using for all operations and other values that the user might want to store into it after the processor starts doing operations. The memory should be able to output any instruction we have stored in it as well as any data when called upon.

Register File:

The register file needs to be able to select any registers located within it (Note: The only situation that it should not be able to select the proper register is if the user tries selecting a register that they are not allowed to manipulate themselves). These registers, just like the previous ones, must be able to hold any and all data values presented to them and output them on their respective output. It must be able to write any data to the proper register when called upon (same restrictions as note).

ALU:

The ALU is a combination of different logic gates that can perform different tasks, arithmetic and logical. At the end, it needs to be able to select the correct output among the multiple outputs it will calculate. Below are all the possible operations we will need to test. There are also two flags that we will need to make sure they are set at proper times, they will be tested with the functions that will need to be able to manipulate these flags.

- Addition
    - Needs to be able to handle big additions that cause overflow like numbers out of the range of a 16 bit data type.
    - Needs to be able to handle adding positive and negative numbers, those that will and will not cause overflow. Basically, detect overflow.
- Subtraction

    - Be able to set an IsZero flag upon subtracting two values to get zero.
    - Needs to be able to handle subtracting positive and negative numbers from each other, those that will and will not cause overflow.
    - Needs to be able to set a Negative flag to represent the confirmation that first number in the argument is less than the second number in the argument.
- Bit AND any two numbers of similar bit size
- Bit OR any two numbers of similar bit size
- Bit NOR any two numbers of similar bit size
- Bit XOR any two numbers of similar bit size
- Bit NAND any two numbers of similar bit size

Control Unit:

The control unit must be able to take in an instructions opcode and proceed to handle the flow of cycles. This means that for each instruction, the transition of the datapath's state must be checked at every stage to ensure correct control bits for the respective cycle.
This will likely need to be tested in Verilog with state transitions and simulations for each instruction.

ALUControl:

This will be tested to ensure that in can take in the function bits decoded from the IR register and hold them until the control finishes decides what it needs to do. It should then be able to either send the function code to the ALU if it needs to use it or send a different function code than what was originally sent into the control.

Zero/Sign Extenders:

- Must be able to extend any immediate we give correctly.

Shifters:

- Shift Left 1:
    - Must be able to shift in a zero to the right of any immediate we give it, even a 16 bit immediate (the bit on the far left would be lost after the shift in this situation).
- Shift Left 8:
    - Must be able to shift in a zero to the right, like the Shift Left 1, just seven more times.

Integration plan and testing:

General Note: Assume all steps are independent of each other and only require listed components.

Step 0:

Goal: Increment PC

Components: PC, ALU (No Mux)

Testing:

General:

- Input A: PC
- Input B: 1
- Op: XXXX (Add)
- Output: PC + 1

Edge Cases:

- N/A

Step 1:

Goal: Read Memory

Components: PC, MemAddr Mux, Memory, MDR (Read-only)

Testing:

General:

NOTE: PCWrite is set to 1

- MemAddr Mux properly selects 1
- Correct address is returned from MemAddrMux
- With MemRead set to 1, value at PC's memory address is outputted
    - MDR value is now value at PC's memory address
- With MemRead set to 0, no value is outputted
    - MDR value is unchanged

Edge Cases:

- Memory address doesn't exist
- No data in PC
- No data at that memory address
- MemWrite set to 1, behavior is expected

Step 2:

Goal: Write Memory

Components: MemAddr Mux, Memory (Write only)

Testing:

General:

- MemAddr Mux properly selects 0
- Correct address is returned from MemAddrMux

- With MemWrite set to 1, value at address is set to write data
- With MemWrite set to 0, value at address is unchanged

Edge Cases:

- With MemRead set to 1, behavior is expected
- Memory address doesn't exist

Step 3:

Goal: Read Register File motivated by IR

Components: IR, Register File (Read-only)

Testing:

General:

- When IRWrite is set to 1, IR is overwritten
- When IRWrite is set to 0, IR is unchanged
- Bits 11-8 from the Instruction Register go to r1 in the register file
- Bits 7-4 from the Instruction Register go to r2 in the register file
- The value in r1 is correctly outputted from the register file
- The value in r2 is correctly outputted from the register file
- Value in $cr is outputted

Edge Cases:

- Bits 11-8 do not refer to an existing register.
- Bits 7-4 do not refer to an existing register.
- The Instruction Register has no data.
- IRWrite is set to 0, and no data is in IR

Step 4:

Goal: Write to Register File

Components: WriteDest Mux, WriteData Mux, Register File (Write only)

Testing:

General:

- WriteDest Mux can select between 2 - $ra, 1 - $cr and 0 - another register and output correct value
- WriteData Mux can select between 5 sources and output correct value
- When RegWrite is set to 1 and WriteDest is set to 0, correct register stores correct value after write
- When RegWrite is set to 0 and WriteDest is set to 0, register destination is unchanged
- When CRWrite is set to 1, RegWrite is set to 1, and WriteDest is set to 1, $cr contains
- When CRWrite or RegWrite is set to 0, and WriteDest is set to 1, $cr is unchanged

- No other register should be written to

Edge Cases:

- WriteDest does not exist (input of >= 3)
- WriteData does not exist (input of >= 5)

Step 5:

Goal: ALU ops and outputs

Components: ALUSrcA Mux, ALUSrcB Mux, ALU, ALUControl

Testing:

General:

- ALUSrcA Mux can select between 2 sources and output correct value
- ALUSrcB Mux can select between 4 sources and output correct value
- ALUControl outputs proper value based on ALUop and func
- When computation results in zero, isZero output is 1, otherwise output is 0
- When computation results in overflow, overflow output is 1, otherwise 0

Edge Cases:

- Func code does not exist
- Input for logical operations are different in bitsize
- Overflow occurs
- ALUop requires function code as output

Step 6:

Goal: PCSrc and control signals

Components: PCSrc Mux, PCWrite, isBranch, isZero, PC

Testing:

General:

- PCSrc properly selects between 4 sources and outputs correct value
- PCWrite set to 1, PC is changed to PCSrc output
- PCWrite set to 0, isBranch and isZero set to 1, PC is changed to PCSrc output
- PCWrite set to 0, isBranch or isZero set to 0, PC is unchanged

Edge Cases:

-

Step 7:

Goal: Shifters and extenders

Components: Shifters, Extenders

Testing:

General:

- Shift Left 1:
  - Ensure any sized input is shifted left one and outputted with original size + 1
- Shift Left 8:
  - Ensure any size input is shifted left eight and outputted with original size + 8
- Zero Extend:
  - 8-bit input is outputted with zero extension
- Sign Extend:
  - 8-bit input is outputted with sign extension

Edge Cases:

- Data of incorrect size is inputted

Final Step:

Combine pieces of processor together and proceed to system testing.

System test plan:

a. Individual instructions
b. Small programs
   i. Code Fragments
      1. Small loop
      2. Small programs
         a. Simple arithmetic and logic
         b. Simple jumps
c. Large programs
   i. relPrime (see pg. 13-20)