Design Documentation

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

CONTENTS:

Register Descriptions

Table of Register Descriptions

Register	Number	Availability	Description
$*sn* (0-2) **<(^.^)>**	0-2	Read/write	General purpose registers. The intent is to store long term computation results and save values over functions calls
$*tn* (0-3) **<(-_-,)>**	3-6	Read/write	General purpose registers, to be used like $*sn* registers. Will NOT be saved over function calls.
$cr **<(-_-,)>**	7	Read only*	Accumulator - stores most recently computed value
$ra **<(^.^)>**	8	Read/write	Stores the return address of a function
$*an* (0-1) **<(-_-,)>**	9-10	Read/write	These registers are used to store arguments for use in a called function
$v **<(-_-,)>**	11	Read/write	This register is for storing the return value from a function
$st **<(^.^)>**	12	Read/write	Reference “top” or lowest memory address of stack
$*in* (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)

opcode	r1	r2	func

15 11 7 3 0

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

<u>Branch Instructions</u>

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: **J-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	8’b0
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*

1000	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	8'b0

4	4	8

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

Load word:

lw r1

1010	r1	8’b0
4	4	8

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

Store word:

sw r1

1011	r1	8’b0
4	4	8

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

Special Procedures

Interact with I/O

syscall r1

1110	4’b0	8’b0
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 Verilog & Machine Language Translations for Instructions

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

Instruction	Type	Verilog	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
Set Less Than	C	R[8] = (R[r1] < R[r2]) ? 1 : 0	0000	r1	r2	0101
Branch Equal	I	If (R[8] = R[r1]) PC = PC + 2 + BranchAddr	0001	r1	BranchAddr	
Branch Not Equal	I	If (R[8] != R[r1]) PC = PC + 2 + BranchAddr	0010	r1	BranchAddr	
Jump	L	PC = JumpAddr	0011	JumpAddr		
Jump and Link	L	R[9] = PC + 2 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	I	R[r1] = SignExtImm	1000	r1	Immediate	

Register					
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[8] = 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*	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.

	```
	        ltr     $t2, 3
	loop:   slt     $s1, $t1
	        bieq    $t0, exit
	        add     $t1, $t2
	        ctr     $t1
	        j       loop
	exit:   ...
	```
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

0110	0100	11111100

add $st $t0

0000	1100	0011	0000

ctr $st

1001	1100	00000000

sw $ra

1011	1000	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

0111	1011	00000000

rtr $s0 $a1 # m in $s0

1000	1101	00000000

0000	1010	1101	0000

0111	0000	00000000

relPrimeLoop:

jal gcd

0100	000000100100

sbc

1000	1101	00000000

1000	1110	00000001

0000	1101	1110	0000

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

0001	1011	1010000

# 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	000000010100

gcd:

sbc 0

1000	1101	00000000

1000	1110	00000000

0000	1101	1110	0000

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

0010	1001	01101100

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	01000110

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]	OP = IR[15-12]
RA = Register[9]	IMM8 = IR[7-0]
I0 = Register[13]	IMM12 = IR[11-0]
I1 = Register[14]	IMM16 = 16-bit immediate
R1 = Register[IR[11-8]]	IR = Instruction Register
R2 = Register[IR[7-4]]	MDR = Memory Data Register

Common RTL:

Cycle	Label	RTL

0	FETCH INSTR:	PC = PC + 2
		IR = Mem[PC]

1	DECODE:	A = R1
		B = R2
		C = CR
		ALUout = PC + SE(IMM8 << 1)

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	

Instruction *Load upper / lower Imm.*	**Type** **I**	**Op** **0110 / 0111**	**func** **-**
Cycle	**Label**	**RTL**	

2	LUI:
	
	LLI:
		R1 = IMM16[15-8] << 8
		
		R1 = ZE(IMM16[7-0])	

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	

Instruction *Store word*	**Type** **I**	**Op** **1011**	**func** **-**
Cycle	**Label**	**RTL**	
2	SW:	Mem[C] = R1	

Instruction *syscall*	**Type** **I**	**Op** **1110**	**func** **-**
Cycle	**Label**	**RTL**	
2	INCR PC:	PC = PC + 2	

?	READ:
	
	WRITE:
	
	EXIT:
		if FINISHED then goto RDONE
		
		if FINISHED then goto WDONE
		
		Close program	

?	RLOOP:	if FINISHED then goto RDONE else goto RLOOP	

	WLOOP:	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	

Instruction **-**	**Type** **L**	**Op** **0011 / 0100**	**func** **-**
Cycle	**Label**	**RTL**	

2	J:
	
	JAL:
		PC = (IMM12 << 1) | PC[15-12]
		
		RA = PC
		PC = (IMM12 << 1) | PC[15-12]	

Processor Components:

- PC
- Memory
- IR (Instruction Register)
- MDR (Memory Data Register)
- Register File
- Shift Left 8
- A/B/C
- ALUout
- ALU
- ALUControl
- Shift Left 1
- Sign Extender
- Zero Extender

Table of Components and Signals

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

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*
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	1’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 ALUControl:

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

PCWrite
- 0 - turns off; 1 - turns on

isBranch
- 0 - not branch
- 1 - is branch

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
- 0 - r1
- 1 - $cr

WriteSrc
- 000 - ALUout
- 001 - Register C
- 010 - MDR
- 011 - (IMM16[15-8] << 8)
- 100 - ZE(IMM16[7-0])

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 - 2
- 10 - SE(IMM8 << 1)
- 11 - SE(IMM8)

ALUop
- 00 - Add (PC increment)
- 01 - Sub (Branch)
- 10 - Look at function code (C-type)
- 11 - Jump register and L-type

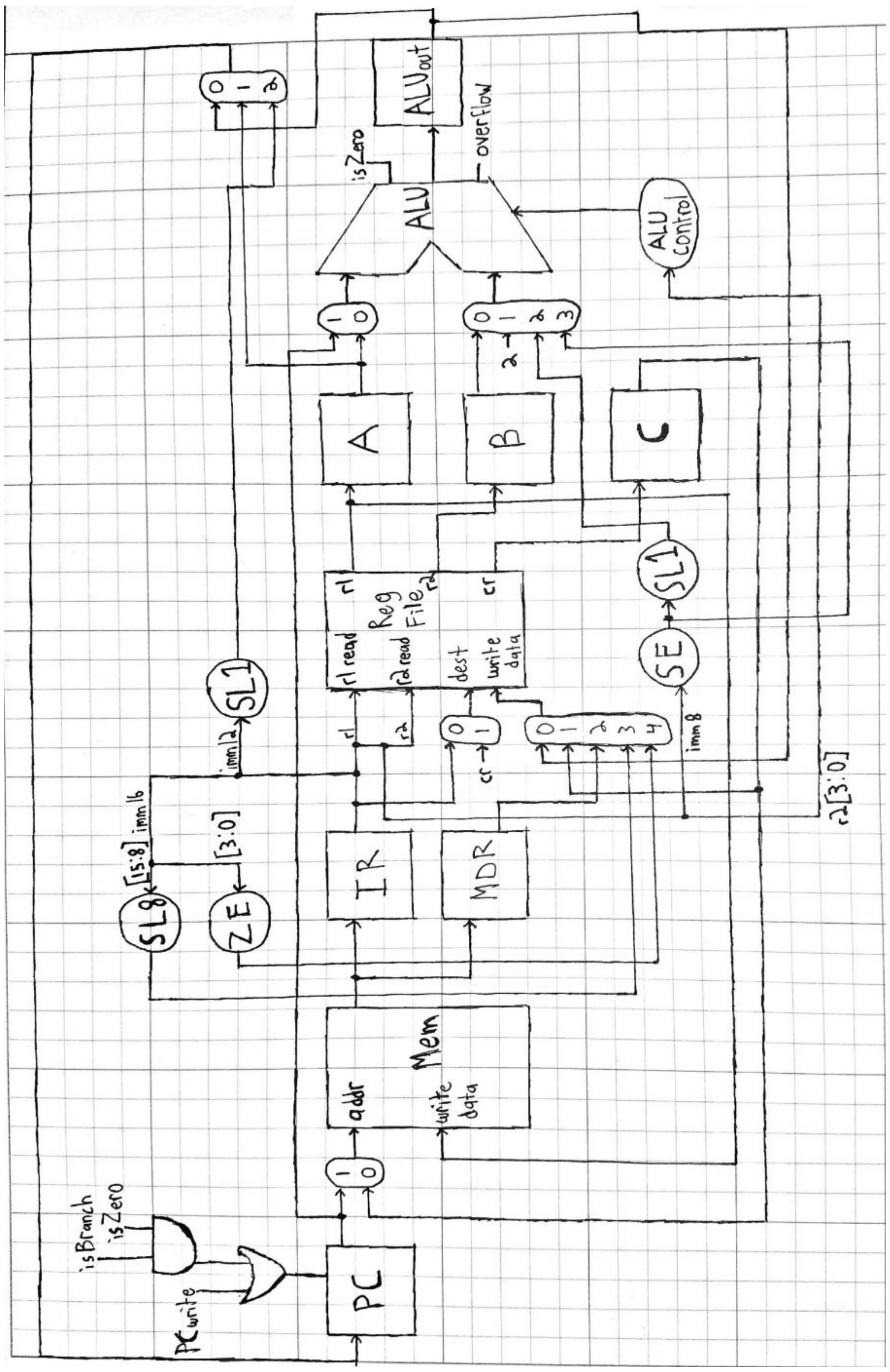
func
- 4’b function code determined by C-type

opcode
- Determined by instruction

current_state/next_state
- Keeps track of state in multi-cycle

CLK
- Clock speed

Reset
- 0 - Reset; 1 - Don't reset



Block Diagram of Datapath

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 VHDL 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
- There will be 1-bit for write enable for the register file which will be comprised of a flip flop with a specified register number
- The register file can be written in verilog

ALU:

- For a 16-bit ALU, we will a unit for Arithmetic (adds and subs), a unit for Logic (OR, NOR, XOR, AND, NAND), and a unit for shifting (slt).
- Each unit will have a 16-bit input and output
- The outputs channel into a mux that chooses which data to output of the ALU, based on ALU control
- 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

Zero Extender

- Another verilog module that fills top bits with zero.

Shifters:

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

Integration plan and 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

1. PC:
    - Needed Components: PC, PCSrc Mux, ALU, ALUSrcA/B Muxes, ALUOut
        - Test that the PC can select the correct PC Address from the PCSrc mux
        - Test that the ALU can send the proper address from branch testing and the have the PCSrc Mux select the correct address.
        - Test that the ALU can send the proper address from Jump/L-type testing and the PCSrc gets the right address.
2. C-type:
    - Needed Components: PC, MemAddr Mux (instr only), Mem, IR, CRWrite Mux, RegWrite Mux, WriteData Mux, RegFile, A/B/C Registers, ALU, ALUSrcA/B, ALUout
        - Tests that all operations write results to $cr
        - Test that the correct registers are read from based on the decoded instruction.
        - After the correct registers are read, they need to be sent to the correct holding register, A or B to be feed into their appropriate muxes.
        - Test that all the control signals sent during all C-type instructions selects the holding registers.
        - Test that the WriteData Mux can select the calculation and not the previous $cr data held in register C
        - Test that at the end of every C-type instruction, the C holding register gets updated.
        - Test that the control has the CRWriteMux select the write input so it can hold the calculated value while other instructions are happening that aren’t C-types.
        - Test that control signals will have the RegWrite Mux select the correct register to write to based on the decoded instruction from IR
        - Test that IR can hold the decoded instruction and send it to the Register File

3. Load/Store:
    - Additional Components: PC, MemAddr Mux (instr and data), Mem, IR, CRWrite Mux, RegWrite Mux, WriteData Mux, RegFile, A/B/C Registers, ALU, ALUSrcA/B muxes, ALUout
        - Tests that no operations store results to $cr
        - Correct address

    - Correctly sets MemAddr mux to data/instr based on cycle
    - Correct data in MDR
    - Correct input into ALU
    - Correct output from ALU

4. Jump register and L-Type:
    - Needed Components: PC, PCSrc, MemAddr Mux (instr only), Mem, IR, CRWrite Mux, RegWrite Mux, WriteData Mux, RegFile, A Register
        - Tests that no operations store results to $cr
        - Correctly sets PC
        - Jump And Link must correctly set $ra

5. Branch:
    - Needed Components: PC, PCSrc, MemAddr Mux (instr only), Mem, IR, RegFile, C, ALU, isZero, isBranch
        - Correctly sets branch conditions for writing to PC
        - Correct ALU output
        - Correctly sets PC

Unit Testing:

PC:

- Conditional branches
    - Needs to be able to go to the branch address upon the branch condition being fulfilled and needs to be able to continue to the next appropriate instruction on a unfulfilled branch condition
- Jumps
    - Needs to be able to be set to the correct jump address when a L-type instruction is called, as long as it is within the proper jump range.
- Regular(Increment)
    - Needs to be able to systematically go through a set of instructions without failing.

Memory:

- Read Data
    - Needs to be able to read from the correct address when needed.
    - Must always be set to 0 (turned off) when no reading is needed.
- Write Data
    - Needs to be able to write to the correct address when needed.
    - Must always be set to 0 (turned off) when no writing is needed.

- Instruction vs. Data
    - Will need to be able to decode instructions correctly and not try to decode data at appropriate times and vice versa .

Instruction Register/A/B/C/Memory Data Register:

- *IR:* Must be able to hold on to an instruction and send it out to the register file when needed.
- *A/B/C:* For A and B, they need to send the correct value to the ALU based on source. For C, it must always hold the value last stored in $cr.
- *MDR:* The Memory Data register must be able to hold the data acquired from the memory register and send it to the register file as write data.

Register File:

- Reading Registers
    - Needs to be able to read from the appropriate register based on what instruction type is presented.
- Writing to Registers
    - Need to be able to write to the appropriate register when called upon.
    - Needs to not be able to write to the $cr if the instruction type is not a C-type.
    - Must be set to 0 (turned off) when no writing is needed.
    - Needs to always write the result to the $cr register at the end of every C-type instruction.

ALU:

- Addition
    - Needs to be able to handle big additions that cause overflow.
    - Needs to be able to handle adding positive and negative numbers, those that will and will not cause overflow.
- Subtraction
    - Be able to set an IsZero output value 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.
- Bit AND
- Bit OR

ALUControl

- Confirm desired control bit outputs based on ALUop and func input

Zero/Sign Extender:

- Must be able to extend any immediate we give correctly, even if it already is 16 bits long (we assume it will do nothing).

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.