Design Documentation

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.
$cr	7	Read only*	Accumulator - stores most recently computed value
$ra	8	Read/write	Stores the return address of a function
$a*n* (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
$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**

Assembly

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:

Arithmetic and Logical Instructions

Arithmetic and Logical Instructions are C-type instructions (see Machine Language for more information). These instructions take two registers as operands to their computations. Their results are always stored in the specialized $cr register.

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 compare the value in $cr to the value in r1. The branch instruction will then succeed or fail based on equivalence or inequality.

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:

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 often require a value to be computed and stored in $cr prior to execution. Specific requirements are denoted for each instruction.

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 that do not actually exist, but that our processor will handle with smaller instructions. All pseudo instructions are I-type. Translations are provided. It is important to note that data in the $cr register is overwritten any time a C-type instruction is executed. Therefore, users should back up any data in $cr they wish to use later.

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

Example 1 - relPrime and Euclid's algorithm:

```
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
```

Example 2 - Common operations:

Table of Common Operations

	SAPA 1.0	Description
Load Address	*la $s2, 0x4EF6*	Loading an address is a pseudoinstruction. Refer to *Pseudoinstruction* 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:*	This example of code does recursive addition of adding two until the value of 20

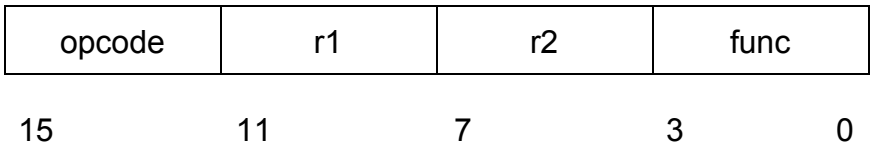
	`` ltr $t0, 2 add $t0, $a0 ctr $t1 sbc 20 bneq $t1, add2 sjr end jr $ra … end: … ```	is reached. However, it needs to move to end instead of moving to the next set of recursive code. Unfortunately, it is too far for the regular jump code to reach. The user can instead; however, set the value of the desired address to the $ra register and can now jump to it.

Machine Language

Instruction Types:

Basic Instruction Formats:

C-type, Computation types (register to register)

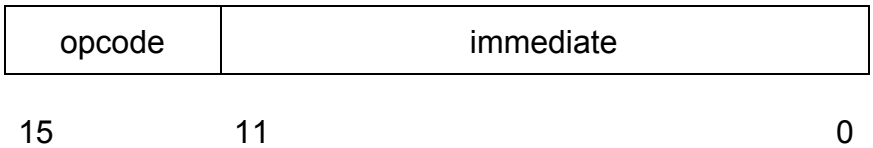


C-type instructions are used for register to register computations. They handle arithmetic and logical computations such as add, sub, and, or, etc. These instructions share a single opcode, and are distinguishable by their func code.

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

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

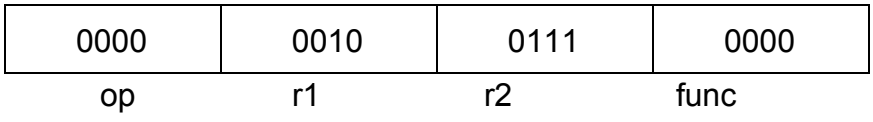


L-type instructions are used for jal and j instructions.

Rules for Translating Assembly to Machine Language:

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



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

Branch instructions are PC-relative, meaning they use an 8-bit offset that allows a user to jump to 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

Jump instructions use a 12-bit immediate with the top 4 bits of the PC concatenated to create a 16-bit address. Jump instructions use this address for the new PC address to jump to, giving a 2^{11} size block of memory to work with.

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

lw $t0

1000	0101	unused
op	r1	Imm

Table of the machine language translations for instructions

Key: 8 = $cr, 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 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)	*Pseudoinstruction*		
Set Jump Register	-	R[8] = JumpAddr	*Pseudoinstruction*		
ORi	-	R[8] = r1 | ZeroExtImm	*Pseudoinstruction*		
Load Address	-	R[r1] = SignExtImm	*Pseudoinstruction*		
Register to Register	-	R[r1] = R[r2]	*Pseudoinstruction*		
Addi	-	R[r1] = SignExtImm	*Pseudoinstruction*		

Machine Language Translations:

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]
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	Control Descriptions
0	FETCH INSTR:	PC = PC + 2 IR = Mem[PC]	- Set ALUsrcA to PC - Set ALUsrcB to 2 - Set ALU op to addition - Set mem to read - Set mem to instruction address
1	DECODE:	A = R1 B = R2 ALUout = PC + SE(IMM8 << 2)	- Set ALUsrcA to R1/PC - Set ALUsrcB to R2/SE(IMM<<2) - Set ALU op to addition
Last	DONE:	Goto FETCH INSTR	- Set PC based on PC source (jump, branch, or routine PC increment)

Unique RTL:

Instr. / Type C-type	**Type** C	**Op** 0000	**func** 0000 - 0111
Cycle	**Label**	**RTL**	**Control Descriptions**
2	OPERATION:	ALUout = A op B	- Set ALU op based on ALU control code (add, sub, or look at func)

3		CR = ALUout	- Set write dest to $cr - Set write data to ALUout result - Set CR to write
Instr. / Type Branch if equal / not equal	**Type** I	**Op** 0001 / 0010	**func** -
Cycle	**Label**	**RTL**	**Control Descriptions**
2	SETCR:	B = CR	- Set r2 to $cr
3	BIEQ: BNEQ:	if (A == B) then PC = ALUout if (A != B) then PC = ALUout	- Turn branch conditions on - Set PC source to be ALUout result
Instr. / Type Load upper / lower Imm.	**Type** I	**Op** 0110 / 0111	**func** -
Cycle	**Label**	**RTL**	**Control Descriptions**
2	LUI: LLI:	A = IMM16 B = 16'b1111111100000000 R1 = ZE(IMM16[7-0])	- Set ALUsrcA/ALUsrcB as appropriate - Set Reg Write - Set write destination to R1 - Set write data to ZE(IMM16[7-0])
3	LUI:	ALUout = A & B	- Set ALUop
4	LUI:	R1 = ALUout	- Set write destination to R1 Set register Write - Set write data to ALUout result

Instr. / Type Load to register	**Type** I	**Op** 1000	**func** -
Cycle	**Label**	**RTL**	**Control Descriptions**
2	LTR16: LTR8:	A = IMM16 B = 16'b1111111100000000 if IMM8 then R1 = SE(IMM8)	- Set ALUsrcA/ALUSrcB to appropriate sources - Set ALUop appropriately - Set write destinations appropriately - Set register write
3	LTR16:	if IMM16 then ALUout = A & B I1 = ZE(IMM[7-0])	
4	LTR16:	I0 = ALUout	
5	LTR16:	ALUout = I0 | I1	
6	LTR16:	R1 = ALUout	
Instr. / Type Copy to register	**Type** I	**Op** 1001	**func** -
Cycle	**Label**	**RTL**	**Control Descriptions**
2	CTR:	R1 = CR	- Choose r1 as write destination - $cr as write data - Set register write
Instr. / Type Store word	**Type** I	**Op** 1010	**func** -
Cycle	**Label**	**RTL**	**Control Descriptions**
2	DECODE2:	R2 = CR	- Set r2 to $cr
3	LDFROMEM:	MDR = Mem[R2]	- Set memory address to data

			- Set memory to read
4	LOADTOREG:	R1 = MDR	- Choose r1 as write destination - Set write data as MDR - Set register write
Instr. / Type Store word	**Type** I	**Op** 1011	**func** -
Cycle	**Label**	**RTL**	**Control Descriptions**
2	SW:	Mem[CR] = R1	- Set write data to r1 - Set r2 to $cr - Set memory to data address - Set memory write
Instr. / Type syscall	**Type** I	**Op** 1110	**func** -
Cycle	**Label**	**RTL**	**Control Descriptions**
2	INCR PC:	PC = PC + 2	- Set PC source to accept syscall address - Set other PC controls
?	READ: WRITE: EXIT:	if FINISHED then goto RDONE if FINISHED then goto WDONE Close program	- Control will set input values in preparation for a system call or accept output values in response to a system call
?	RLOOP: WLOOP:	if FINISHED then goto RDONE else goto RLOOP if FINISHED then goto WDONE else goto WLOOP	Description N/A
?	RDONE: / WDONE:		- Returns control to the program

Instr. / Type Jump Register	**Type** I	**Op** 0101	**func** -
Cycle	**Label**	**RTL**	**Control Descriptions**
2	JR:	PC = R1	- Set PC source to accept jump register address
Instr. / Type L-Type	**Type** L	**Op** 0011 / 0100	**func** -
Cycle	**Label**	**RTL**	**Control Descriptions**
2	J: JAL:	PC = IMM12 | PC[15-12] RA = PC PC = IMM12 | PC[15-12]	- Set PC source to accept j/jal address respectively

RTL Components:

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

Components Signals	Inputs	Outputs	Description of Component & Controls
PC	- newPC	- PC	- Program counter - Control the source of PC and whether PC should be written to
Memory	- Address - Write Data	- MemData	- Access to memory - Control whether an instruction or data is being read - Control whether memory should be read from or written to
IR	- Instruction Data	- r1 - r2 - imm	- Stores instruction data
MDR	- Memory Data	- Memory Data	- Store memory data
Register File	- r1 - r2 - Write - Write Data	- r1 data - r2 data	- Holds registers for reading/writing data - Control decides whether registers should be written to
A/B	- ALUsrcA - ALUsrcB	- A or B respectively	- These components hold operands for the ALU - Control will choose source of A/B
ALU	- A - B	- isZero - Overflow - Result	- Performs operation on two operands - Control chooses the result operation
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

RTL Test Descriptions:

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

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/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:* For A and B, they need to send the correct value to the ALU based on source.
- *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

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).