Processor Design

Group V

Ethan Liu
Wenzi Qian
Agnay Srivastava
Emma Letscher

Table of Contents

Design Principles

The architecture of our instruction set is load-store. Its design is drawn with priority given to the following principles:

1. *Make the common case fast*
2. *Good design demands good compromises*
3. *Smaller is faster*

General Hardware Information

In this document, we define a word to mean 16 bits or two bytes. All instructions are 1 word long.

The instructions are structured as follows:

a. The most significant bit is considered the start of an instruction.
b. The opcode and func codes are placed in the most significant bits.
c. The registers used are specified in the least significant bits in a continuous manner. An instruction may use 0, 3, 6 or 9 bits for this purpose.
d. Bits between the opcode and the register is used for a single immediate. If an instruction uses no general-purpose registers, all bits after the opcode is used for the immediate.

All registers in our processor hold exactly 1 word. We opted to provide the processor with **8 general purpose registers**. In addition to the general purpose registers, the processor also has **3 special registers**, namely:

1. PC, program counter
2. SP, stack pointer
3. RA, return address

The process also has a 16 bit input line and a 16 bit output line, used to receive data from the outside world. The inputs are accessed via instructions `READ` and `WRITE`.

Note that **special registers cannot be accessed directly** in place of regular registers

For memory access, we implemented two access modes:

1. `SET` and `RTV` uses the value in a register as the address in **bytes** to retrieve the value from
2. `SETSP` and `RTVSP` adds the stack pointer in **bytes** and the sign-extended immediate in **words** as the address.

The capability of reading odd-byte aligned word depends on the capability of the memory.

General Purpose Registers

Register	Alias	Usage
x0	a0	Function Argument, Temporary, Return Value
x1	a1	Function Argument, Temporary, Return Value
x2	a2	Function Argument, Temporary, Return Value
x3	a3	Function Argument, Temporary, Return Value
x4	a4	Function Argument, Temporary, Return Value
x5	s0	Saved Value
x6	s1	Saved Value
x7	s2	Saved Value

Special Registers

Stack Pointer and Program Counter are accessible via their dedicated instructions

- Program Counter can be used via `GETPC`, `SETPC`, `CHGPC`, and `CHGPCI`
- Stack Pointer can be used via `GETSP`, `SETSP`, `CHGSP`, and `CHGSPI`
- Return addressed is not accessible with single instruction, and should not be accessed.
    - If access is required, a `JUMP`-`RETURN` pair will place the return address at position 0 on the stack.

These special register values are not accessed or modified via other register instructions.

Instruction Set

Instruction	Desription
ADD	Performs integer addition on the inputs
SUB	Performs integer subtraction on the inputs
AND	Performs bitwise AND on the inputs
OR	Performs bitwise OR on the inputs
XOR	Performs bitwise XOR on the inputs
NAND	Performs bitwise NAND on the inputs
NOR	Performs bitwise NOR on the inputs
XNOR	Performs bitwise XNOR on the inputs
SHIFT	Perform logical shift on the input
ADDI	Adds the input register value with a sign-extended immediate
EQ	Performs a branch if the inputs equal
LT	Performs a branch if the first input is less than the second
NEQ	Performs a branch if the first input is not equal to the second
GEQ	Performs a branch if the first input is greater or equal to the second
GETSP	Stores SP value in destination register
SETSP	Sets SP value with value in input register
CHGSP	Offsets SP value by value in input register
CHGSPI	Offsets SP value by value of the sign-extended immediate
GETPC	Stores PC value to the destination register
SETPC	Sets PC value with value in input register
CHGPC	Offsets PC value by value in input register
CHGPCI	Offsets PC value by value of the sign-extended immediate
JUMP	Store current RA to position 0 of the stack (It is a calling convention to reserve the word at SP as the return address). Set RA to PC + 1 **word** (the line after jump in caller). Offset PC by the value of sign-extended immediate in **words**.

RETURN	Sets PC to RA. Release from the stack a number of **words** equal to the sign-extended immediate value. Restore old RA from the 0 position of the parent's SP.
LU	Sets the upper half of the destination register to be the immediate value, while copying the lower half.
LL	Sets the destination register to be the sign-extended immediate value.
STR	Stores the value of input 2 to the memory address in input 1.
RTV	Gets the value in the memory address stored in input 1 and stores it in the destination register.
STRSP	Stores the value of input 2 to the memory address of SP adding sign-extended immediate value.
RTVSP	Gets the value in the memory address of SP adding sign-extended immediate value and stores it in the destination register.
READ	Read the value from the input line to the destination register.
WRITE	Write the value in input 1 to the output line.

Instruction Formats

Type	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0RR	Opcode			Immediate							Input 2			Input 1		
1R	Opcode			F1	Immediate									Input 1		
1W	Opcode			F1	Immediate									Output		
1WR	Opcode			F1	Immediate						Input 1			Output		
2WR	Opcode			F1	F2	Immediate					Input 1			Output		
2W	Opcode			F1	F2	Immediate								Output		
3	Opcode			F1	F2	F3	Immediate									
4WRR	Opcode			F1	F2	F3	F4	Input 2			Input 1			Output		
7RR	Opcode			F1	F2	F3	F4	F7			Input 2			Input 1		
7WR	Opcode			F1	F2	F3	F4	F7			Input 1			Output		
10R	Opcode			F1	F2	F3	F4	F7			F10			Input 1		
10W	Opcode			F1	F2	F3	F4	F7			F10			Output		

Opcode & Function Code

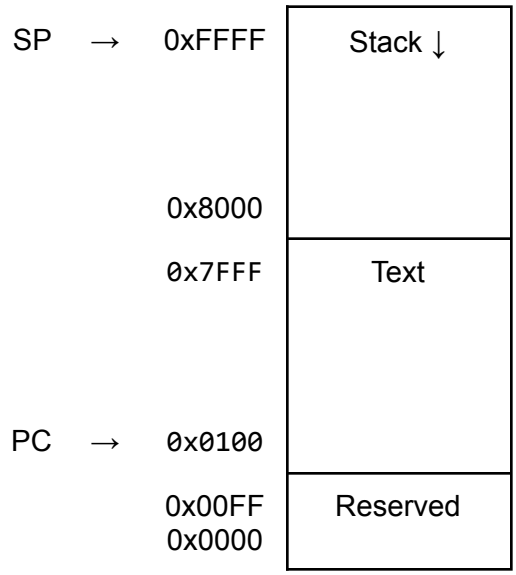
<SE> in the description indicates that the immediate is sign-extended.

Mnemonic	Type	Opcode	F1	F2	F3	F4	F7	F10	Full Name
LT	0RR	101							BRANCH LESS THAN
EQ	0RR	100							BRANCH EQUAL
NEQ	0RR	110							BRANCH NOT EQUAL
GEQ	0RR	111							BRANCH GREATER OR EQUAL
STRSP	1R	010	0						STORE REGISTER AT SP OFFSET
RTVSP	1W	010	1						GET VALUE AT SP OFFSET
ADDI	1WR	001	0						ADD WITH IMMEDIATE
LU	2W	000	1	0					LOAD IMMEDIATE UPPER HALF
LL	2W	000	1	1					LOAD IMMEDIATE LOWER HALF
SHIFT	2WR	001	1	0					LOGICAL LEFT SHIFT

RETURN	3	001	1	1	0				RETURN TO PARENT
JUMP	3	001	1	1	1				JUMP TO CHILD
CHGSPI	3	011	1	1	0				CHANGE SP BY IMMEDIATE
CHGPCI	3	011	1	1	1				CHANGE PC BY IMMEDIATE
ADD	4WRR	000	0	0	0	0			ADDITION
AND	4WRR	000	0	0	0	1			BITWISE AND
OR	4WRR	000	0	0	1	0			BITWISE OR
XOR	4WRR	000	0	0	1	1			BITWISE XOR
SUB	4WRR	000	0	1	0	0			SUBTRACTION
NAND	4WRR	000	0	1	0	1			BITWISE NAND
NOR	4WRR	000	0	1	1	0			BITWISE NOR
XNOR	4WRR	000	0	1	1	1			BITWISE XNOR
STR	7RR	011	1	0	0	0	010		STORE REGISTER AT ABSOLUTE ADDRESS
RTV	7WR	011	1	0	0	0	011		GET VALUE AT ABSOLUTE ADDRESS
WRITE	10R	011	1	0	0	0	100	001	WRITE OUTPUT LINE
GETSP	10R	011	1	0	1	0	000	000	GET SP IN REGISTER
GETPC	10R	011	1	0	1	1	000	000	GET PC IN REGISTER
READ	10W	011	1	0	0	0	100	000	READ INPUT LINE
CHGSP	10W	011	1	0	1	0	100	000	CHANGE SP BY REGISTER
SETSP	10W	011	1	0	1	0	100	001	SET SP BY REGISTER
CHGPC	10W	011	1	0	1	1	100	000	CHANGE PC BY REGISTER
SETPC	10W	011	1	0	1	1	100	001	SET PC BY REGISTER

* Note: RA uses the SP after addition as address to retrieve old RA.

Memory Map



The “Reserved” section is for initializations necessary, such as instruction that stores the initialization of sp and pc and initializing bus registers to 0. Not all space may be used.

Psuedo-Instructions

```
LI: Load Immediate
      Converts to LL, or LI and LU if the immediate is bigger than 8 bits.
STOP: Permanently halt the program until reset
      Can be implemented a number of ways: "EQ x0 x0 0" is one.
```

Performance

We measure the performance of our processor based on the execution speed of programs, including relprime, factoring in how fast can it clock.

Procedure Calling Convention

Procedure calling involves a parent and a child, where the parent prepares the arguments and calls the child, and the child executes code and returns the result. From here on, we will refer to the parent as the caller, and the child as the callee. Note that the calling convention responsibilities are specific to each caller-callee pair, and a function can have responsibilities both as a caller and a callee.

Before a procedure call, the caller does the following:

1. Back up all values that should be retained after the call in a0-a4.
2. Load function arguments in a0 through a4. Additional arguments are passed on the stack.
3. Allocate and reserve the word at location 0 of its stack for storing its parent's return address.

After the jump to callee, it performs the following:

1. If it plans to use any saved registers, it must allocate space on the stack to preserve those values and restore them before it returns.
2. Once computation is complete, put return values in a0 through a4.
3. Call RETURN and release stack space claimed

The processor makes the assumption that when a program returns, it must return to its parent. We applied an optimization which combines multiple actions into single instructions: `JUMP` and `RETURN`. These instructions do the following:

`JUMP`:

1. Store current value in return address register at stack position 0
2. Store in return address register the new return address (program counter plus 1)
3. Change program counter by number of words specified by the sign-extended immediate

`RETURN`:

1. Set program counter to value currently in return address register
2. Set return address with value at stack position 0 *after* stack pointer increment.
   Note: The hardware implementation is that address used comes from the adder calculating the increment. The actual stack pointer value will not change until next half cycle.
3. Change stack pointer by number of words specified by the sign-extended immediate

Note that the new values being set into the registers will not be reflected until the next half cycle. As far as this instruction is concerned, the register outputs are the old values.

The `JUMP` and `RETURN` instructions expects that **if a program is a caller, it must reserve position 0 on its stack** for its parent's return address.

It is currently assumed that the processor will only run a single program and put out results a single time at the top level, since it has no parent to return to, and we cannot allow the program counter to continue emitting the value, we have implemented a psuedo-instruction `STOP` to halt the processor and thus have it keep displaying the outputs.

RTL

RTL Categories

Category	Instructions
1	All type 0RR instructions
2	All type 1R instructions
3	All type 1W instructions
4	All type 1WR instructions
5	Type 2W: LL
6	Type 2W: LU
7	All type 2WR instructions
8	Type 3: CHGPCI, CHGSPI
9	Type 3: JUMP
10	Type 3: RETURN
11	All type 4WRR instructions
12	All type 7RR instructions
13	All type 7WR instructions
14	Type 10R: READ
15	Type 10R: CHGSP, SETSP, CHGPC, SETPC
16	Type 10W: WRITE
17	Type 10W: GETSP, GETPC

Single-Cycle Instruction RTL

Note that for actual implementation, PC and SP would have to be fetched into a register and then that register used for operations. They aren't directly accessible, but we omit that fact in our RTL.

Type	RTL	Type	RTL
1 GEQ NEQ LT EQ	inst = M[PC] newPC = PC + 2 PC = newPC a = R[inst[2:0]] b = R[inst[5:3]] branch = a (cond) b **if (branch) then:** PC = PC + (SE(inst[12:6])<<1)	**10** RETURN	inst = M[PC] PC = RA imm = SE(inst[9:0]) << 1 address = SP + imm SP = address newRA = M[address] RA = newRA
2 STRSP	inst = M[PC] newPC = PC + 2 PC = newPC spv = SP a = R[inst[2:0]] imm = SE(inst[11:3]) << 1 address = spv + imm M[address] = a	**11** AND ADD XOR OR NOR SUB NAND XNOR	inst = M[PC] newPC = PC + 2 PC = newPC a = R[inst[5:3]] b = R[inst[8:6]] funcCode = R[inst[13:9]] result = a (op) b R[inst[2:0]] = result
3 RTVSP	inst = M[PC] newPC = PC + 2 PC = newPC spv = SP imm = SE(inst[11:3]) << 1 address = spv + imm a = M[address] R[inst[2:0]] = a	**12** STR	inst = M[PC] newPC = PC + 2 PC = newPC A = inst[2:0] B = inst[5:3] a = R[B] M[A] = a
4 ADDI	inst = M[PC] newPC = PC + 2 PC = newPC a = R[inst[5:3]] imm = SE(inst[11:6]) result = a + imm R[inst[2:0]] = result	**13** RTV	inst = M[PC] newPC = PC + 2 PC = newPC A = inst[2:0] B = inst[5:3] a = M[A] R[B] = a
5	inst = M[PC] newPC = PC + 2	**14**	inst = M[PC] newPC = PC + 2

LU	PC = newPC imm = inst[10:3] a = R[inst[3:0]] R[inst[3:0]] = {imm[7:0], a[7:0]}	READ	PC = newPC R[inst[2:0]] = ProcessorIn
6 LL	inst = M[PC] newPC = PC + 2 PC = newPC imm = SE(inst[10:3]) R[inst[3:0]] = imm	**15** CHGSP SETSP CHGPC SETPC	inst = M[PC] newPC = PC + 2 PC = newPC imm = SE(inst[9:0]) << 1 CHG: oldValue = (SP/PC) CHG: (SP/PC) = oldValue + imm SET: (SP/PC) = R[inst[2:0]]
7 SHIFT	inst = M[PC] newPC = PC + 2 PC = newPC a = R[inst[5:3]] imm = SE(inst[10:6]) shift = a << imm R[inst[2:0]] = shift	**16** WRITE	inst = M[PC] newPC = PC + 2 PC = newPC ProcessorOut= R[inst[2:0]]
8 (3) CHGSPI CHGPCI	inst = M[PC] oldPC = PC newPC = PC + 2 PC = newPC imm = SE(inst[9:0]) << 1 newVal = SP/PC + imm SP/PC = newVal	**17** GETSP GETPC	inst = M[PC] newPC = PC + 2 PC = newPC R[inst[2:0]]= newSP/newPC
9 JUMP	inst = M[PC] M[SP] = RA newRA = PC + 2 RA = newRA newPC = PC + SE(inst[9:0]) << 1 PC = newPC		

Multi-Cycle Instruction RTL

SE is not an Array, use ( ) not [ ]

Cat.	Cycle 1/2	Cycle 3	Cycle 4	Cycle 5
1		imm = SE(inst[9:0]) branch = a (cond) b	if(branch){PC = target}	
2		imm = SE[inst[11:3]] address = SP + imm	M[address] = a	
3		imm = SE[inst[11:3]] address = SP + imm	a = M[address]	
4		imm = SE[inst[11:6]] value = b + imm	R[a] = value	
5	Cycle 1	R[inst[2:0]]={imm[7:0],a[7:0]}		
6	inst = M[PC]	R[inst[2:0]] = imm		
7	newPC = PC + 2	shift = a << imm	R[inst[2:0]] = shift	
8	---	imm = inst[9:0] SP = SP + imm		
9	Cycle 2 PC = newPC	M[SP] = RA RA = PC + 2 PC = PC + SE(inst[9:0])<<1		
10	a = R[inst[2:0]] b = R[inst[5:3]] c = R[inst[8:6]]	imm = SE(inst[9:0]) << 1 address = + imm	SP = address L = M[address]	RA = L
11		res = b (op) c	R[inst[2:0]] = res	
12		M[a] = b		
13		value = M[a]	R[b] = value	
14		R[inst[2:0]] = ProcessorIn		
15		R[inst[2:0]] = newSP/newPC		
16		ProcessorOut = R[inst[2:0]]		
17		SP / PC = R[inst[2:0]]		

?

division?

Write Flags By Instruction

control bits?

Category	Write To GP Register	Write To Stack Pointer	Write Alternate Value To Program Counter	Write To Return Address	Write To Memory
1			Cond.		
2					X
3	X				
4	X				
5	X				
6	X				
7	X				
8		Cond.	Cond.		
9			X	X	X
10		X	X	X	
11	X				
12					X
13	X				
14					
15	X				
16	X				
17		Cond.	Cond.		

RTL Error-Checking Process

General Syntax Checking:

1. Ensure that only one register retrieval (any reference to register bus R[x] or dedicated registers SP, RA, and PC) occurs on a given line
2. 3-bit sequences of instruction bits (e.g. inst[2:0]) should only be accessed when loading from a register. Longer sequences are immediates and can be accessed directly
3. Make sure sign extension <SE> and bit-shifting are noted as needed (see instruction format) when handling immediates

RTL Error-Checking:

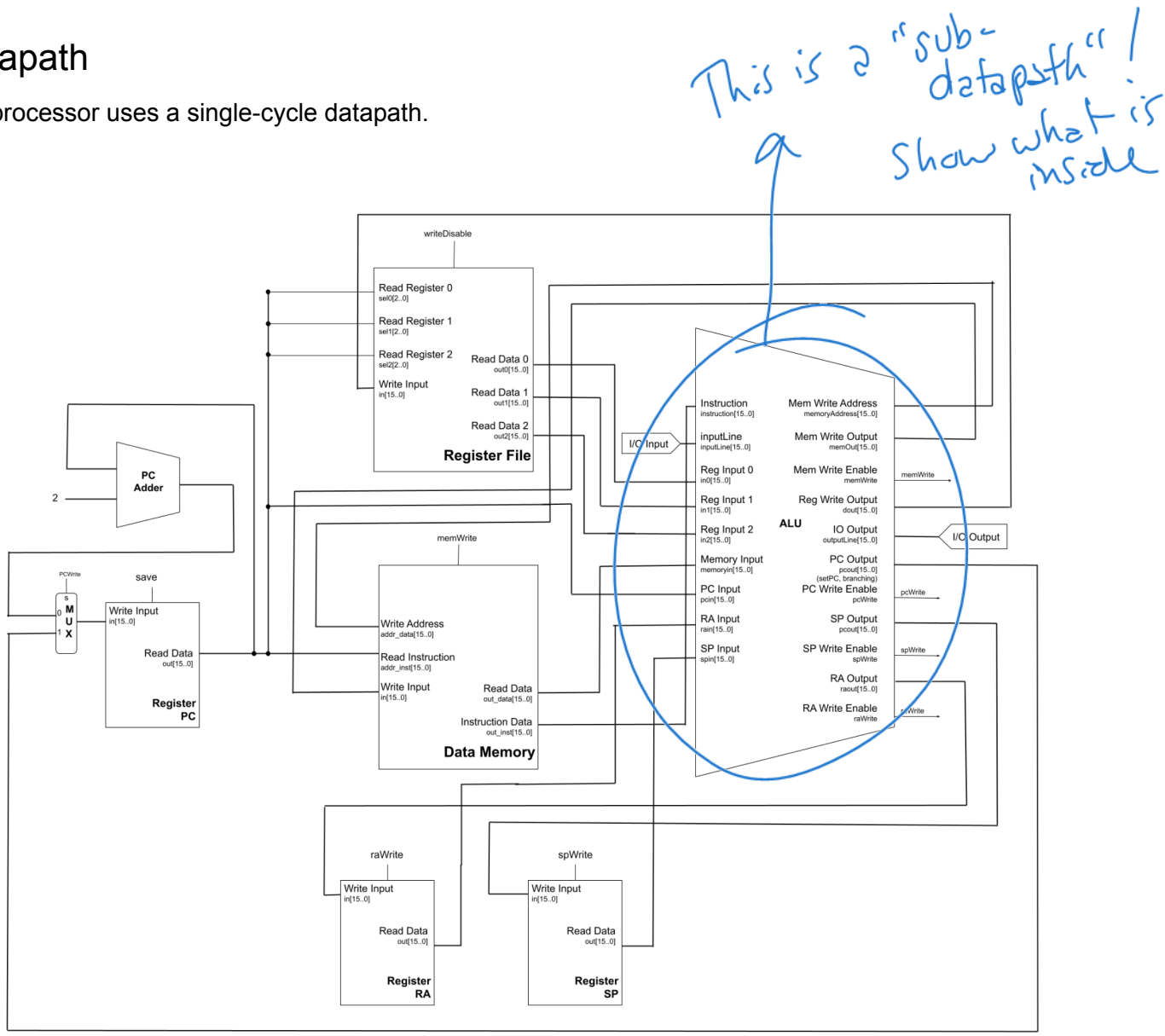
1. Write a complete instruction in machine code. Separate it bitwise by different components of the instruction (e.g. clearly label opcode, rs1, etc.)
2. Then, use it to step by the instruction line-by-line. For every RTL instruction, write the instruction itself along with the following:
    a. If the RTL references some portion of inst[], write those bits of inst out where they are referenced, or draw an arrow to the portion of the instruction they are referenced.
    b. Draw a stack frame indicating the current position of the SP and RA and any values that were pushed onto/retrieved from the stack with that instruction.
    c. Draw a basic diagram of the register file and update it with the relevant values being loaded/stored.
    d. If the instruction uses a component such as the ALU, make note of the inputs and outputs to this component.
3. After doing this for all RTL instructions, look at the final state of the stack and registers to verify that your instruction does what you intended it to do.

RTL Naming Convention

1. Output wires of components are denoted in the “RTL Symbols” column of the above table.
2. Access to the bus registers (x0-x11) is denoted by R[address], access to the dedicated registers RA, SP, and PC are denoted directly by mnemonic
3. Access to data memory is denoted by M[address]

Datapath

Our processor uses a single-cycle datapath.



Control Signals

Name	Description
RAWrite	Writes `in[15:0]` to RA Register when high
SPWrite	Writes `in[15:0]` to SP Register when high
MemWrite	Writes `in[15:0]` to `writeAddress[15:0]` when high
PCWrite	Determines source of PC register input PCWrite = 1 - When a branch instruction or CHGPC instruction that directly modifies the PC is executed, setting PC input to this instruction-specified value PCWrite = 0 - Default value; the next PC instruction will default to the old instruction run through an adder value fixed to 2 (PC Adder)
WriteDisable	Writes `in[15:0]` to register specified by `inst[2:0]` when high

Integration

Datapath Implementation Plan

We are overall taking a bottom-up approach to implementing the datapath, starting with the smallest subcomponents possible. We will perform integration tests on any components that use more than one subcomponent.

draw diagram for this!

ALU:

- In our implementation, the instruction decoder is built into the ALU. The main ALU_Control unit inside the ALU decodes the instruction and sets the appropriate read/write flags. The ALU itself is subdivided into the 4 ALUs outlined in the component list, and then combined into a fully operational L_ALU_Complex, where instruction[15:0] will be appropriately parsed and the appropriate function code passed to the appropriate sub-ALU (along with the appropriate read/write flags being set as the ALU output travels elsewhere in the processor).

Registers:

- The main bus registers (x0-x11) will be stored in a Register File that has an array of register subcomponents.
- PC, RA, and SP, as shown, are dedicated registers accessible at any time via their own dedicated commands (e.g. CHGPC, CHGRA, CHGSP, and Branching commands for the RA).

Component Descriptions

Component	Input	Outputs	RTL Symbols
Register	din[15:0] clk rst load save	dout[15:0]	SP PC RA
Register File	readSel1[2:0] readSel2[2:0] readSel3[2:0] writeSel[2:0] readEnable writeEnable	regOut1[15:0],regOut2[15:0],regOut3[15:0]	R[x]
L_ALU_Complex	instruction[15:0] inputLine[15:0] in0[15:0] in1[15:0] in2[15:0] memoryin[15:0] rain[15:0] spin[15:0] pcin[15:0]	memoryAddress[15:0] memOut[15:0] memWrite out[15:0] outputLine[15:0] PCOut[15:0] spWrite raOut[15:0] raWrite	
L_ALU_11	inputA[15:0] inputB[15:0] control[2:0]	output[15:0]	result
L_ALU_3	immediate[8:0] spIn[15:0] memoryIn[15:0]	memoryAddress[15:0] dout[15:0]	
0RR_ALU	inputA[15:0] inputB[15:0] EQ LT NEQ GEQ	output	branch
10_ALU	inputA[15:0] inputB[15:0] GETSP CHGSP SETSP GETPC SETPC CHGPC SETPCI	output[15:0]	newSP, newPC, address

	CHGPCI		
Misc_ALU	inputA[15:0] inputB[15:0] LU LL ADDI SHIFT RETURN JUMP STRSP RTVSP STR RTV READ WRITE	output[15:0]	<<
Memory Unit	i_addr[15:0] d_addr[15:0] in_data[15:0] read write	i_data[15:0] d_data[15:0]	MEM[x]

Appendix A: Assembled Instructions

Type 0RR

EQ a0 a3 5

Bits	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
Usage	Opcode			Immediate							Input 2			Input 1		
Value	1	0	0	0	0	0	0	1	0	1	0	1	1	0	0	0
Meaning	-			<SE> Immediate: 5							Input 2: a3			Input 1: a0		

Type 1WR

ADDI x3 x5 -11

Bits	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
Usage	Opcode			F1	Immediate						Input 1			Output		
Value	0	0	1	0	1	1	0	1	0	1	1	0	1	0	1	1
Meaning	-			<SE> Immediate: -11							Input 1: x5			Output: x3		

Type 1R

STRSP s0 3

Bits	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
Usage	Opcode			F1	Immediate									Input 1		
Value	0	1	0	0	0	0	0	0	0	0	0	1	1	1	0	1
Meaning	-			<SE> Immediate: 3										Input 1: s0		

Type 1W

RTVSP s1 2

Bits	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
Usage	Opcode			F1	Immediate									Output		
Value	0	1	0	1	0	0	0	0	0	0	0	1	0	1	1	0
Meaning	-			<SE> Immediate: 2										Output: s1		

Type 2W

LL a3 17

Bits	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
Usage	Opcode			F1	F2	Immediate								Output		
Value	0	0	0	1	1	0	0	0	1	0	0	0	1	0	1	1
Meaning	-					<SE> Immediate: 17								Output: a3		

Type 2WR

SHIFT a1 s2 16

Bits	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
Usage	Opcode			F1	F2	Immediate					Input 1			Output		
Value	0	0	1	1	0	0	1	0	0	0	1	1	1	0	0	1
Meaning	-					<SE> Immediate: 16					Input 1: s2			Output: a1		

Type 3

JUMP 177

Bits	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
Usage	Opcode			F1	F2	F3	Immediate									
Value	0	0	1	1	1	1	0	0	1	0	1	1	0	0	0	1
Meaning	-						<SE> Immediate: 177									

Type 4WRR

ADD x1 x3 x5

Bits	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
Usage	Opcode			F1	F2	F3	F4	Input 2			Input 1			Output		
Value	0	0	0	0	0	0	0	1	0	1	0	1	1	0	0	1
Meaning	-							Input 2: x5			Input 1: x3			Output: x1		

Type 7RR

STR s1 s0

Bits	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
Usage	Opcode			F1	F2	F3	F4	F7			Input 2			Input 1		
Value	0	1	1	1	0	0	0	0	1	0	1	0	1	1	1	0
Meaning	-										Input 2: s0			Input 1: s1		

Type 7WR

RTV s0 s1

Bits	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
Usage	Opcode			F1	F2	F3	F4	F7			Input 2			Input 1		
Value	0	1	1	1	0	0	0	0	1	1	1	1	0	1	0	1
Meaning	-										Input 2: s1			Input 1: s0		

Type 10R

SETSP a0

Bits	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
Usage	Opcode			F1	F2	F3	F4	F7			F10			Input 1		
Value	0	1	1	1	0	1	0	1	0	0	0	0	1	0	0	0
Meaning	-													Input 1: a0		

Type 10W

GETSP a2

Bits	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
Usage	Opcode			F1	F2	F3	F4	F7			F10			Output		
Value	0	1	1	1	0	1	0	0	0	0	0	0	0	0	1	0
Meaning	-													Output: a2		

Appendix B: Single Input RelPrime

We demonstrate the workings of our processor with the following instruction:

```
void main() {
      int result = relPrime(30);
}

int relPrime(int n) {
   int 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;
}
```

Address	Assembly	Machine Code	Comments
0x0100	li a0 6	0001 1000 1111 0000	Input 6
0x0102	jump 2	0011 1100 0000 0010	
0x0104	stop	1000 0000 0000 0000	Assembled as `eq x0 x0 0`; halts the program indefinitely.
			Start of RelPrime
0x0106	chgspi -3	0111 1011 1111 1101	Allocate 3 words of stack space for the program

			Saves on stack:
			(0) Relprime parent's return address. Saving this is automated by instruction `JUMP`.
0x0108	strsp s0 1	0100 0000 0000 1101	(1) Original Value in S0
0x010A	strsp s1 2	0100 0000 0001 0110	(2) Original Value in S1
0x010C	addi s0 a0 0	0010 0000 0000 0101	s0 stores input
0x010E	li s1 2	0001 1000 0001 0110	s1 stores current tested number
0x0110	li a3 1	0001 1000 0000 1011	# Load in the constant 1
0x0112	addi a0 s0 0	0010 0000 0010 1000	Prior to procedure call, store input a0 in s0
0x0114	addi a1 s1 0	0010 0000 0011 0001	Prior to procedure call, store input a1 in s1
0x0116	jump 8	0011 1100 0000 1000	Go forward 8 words to call GCD
0x0118	eq a0 a3 3	1000 0001 0100 0011	After return, the GCD is in a0. If the GCD equals 1, the number currently tested (s1) is relatively prime to our target. In this case, we jump forward 3 words to return
0x011A	addi s1 s1 1	0010 0000 0111 0110	Otherwise, we need to keep searching. Increment the current tested number.
0x011C	chgpci -5	0111 1111 1111 1011	Go back 5 words to the start of loop
0x011E	addi a0 s1 0	0010 0000 0011 0000	Put return value in a0
0x0120	rtvsp s0 1	0101 0000 0000 1101	Restore original value in s0
0x0122	rtvsp s1 2	0101 0000 0001 0110	Restore original value in s1
0x0124	return 3	0011 1000 0000 0110	Return to parent (main) and moves stack pointer by 3 words
			Start of GCD
0x0126	li a2 0	0001 1000 0000	Load constant 0 for comparison

		0110	
0x0128	neq a2 a0 3	1100 0000 1101 0000	If a0 is already zero, simply return a1.
0x012A	addi a0 a1 0	0010 0000 0000 1000	Load a1 into a0 for return
0x012C	return 0	0011 1000 0000 0000	Return to parent (RelPrime). No stack space was allocated, so none needs returned.
0x012E	eq a1 a2 6	1000 0001 1000 1100	If b equals 0, the loop is done. Jump to the end.
0x0130	geq a1 a0 3	1110 0001 1000 1001	Go to the else clause if a is less than or equal to b. Otherwise, proceed to then clause.
0x0132	sub a0 a0 a1	0000 1000 0100 0000	Subtract b from a
0x0134	chgpci -3	0111 1111 1111 1101	Repeat loop
0x0136	sub a1 a1 a0	0000 1000 0000 1001	Subtract a from b
0x0138	chgpci -5	0111 1111 1111 1011	Repeat loop
0x013A	return 0	0011 1000 0000 0000	Return to parent (RelPrime). No stack space was allocated, so none needs returned.

Appendix C: Sample Operations

Iteration

The following code segment is roughly equivalent to this Java code:

```
int x = 7;
for(int i = 0; i<10; i++){
   x = x + 2;
}
```

Address	Assembly	Machine Code	Comments
0x0100	li a0 7	0001 1000 0011 1000	# # a0 = x, initialized to 7
0x0102	li a1 10	0001 1000 0011 1000	# a1 = 10
0x0104	li a2 0	0001 1000 0000 0010	# # a2 = i, initialized to 0
0x0106	geq a2 a1 4	1110 0001 0000 1010	# # if i>= 10, we are done, jump forward 4 instructions
0x0108	addi a0 a0 2	0010 0000 1000 0000	# # a += 2
0x010A	chgpci -3	0111 1111 1111 1101	# # go to start of loop

Loading Large Immediates

```
int number = 12007;

12007 = 0b0010111011100111
```

Address	Assembly	Machine Code	Comments
0x0100	ll a0 -25	0001 1111 0011 1000	# Loads lower 8 bits into a0 (0b11100111 = 231)
0x0102	lu a0 46	0001 0001 0111 0000	# Loads upper 8 bits into a0 (0b00101110 = 46)

Conditionals (if/else/then)

The following assembly is roughly equivalent to this Java code:

```
if(x >= y){
      x++;
}
else {
      x–;
}
```

Address	Assembly	Machine Code	Comments
0x0100	lt a0 a1 3	1010 0000 1100 1000	# # x++
0x0102	addi a0 a0 1	0010 0000 0100 0000	# # skip over else block
0x0104	chgpci 2	0111 1100 0000 0010	# # x--
0x0106	addi a0 a0 -1	0010 1111 1100 0000	

Procedure Calling

```
void main() {
	int product = mult(9, 15);
}

int mult(int a, int b) {
	int result = 0;
	for(int i = 0; i < b; i++){
		result+=a;
	}
	return result;
}
```

Address	Assembly	Machine Code	Comments
0x0100	li a0 9	0001 1000 0100 1000	Main
0x0102	li a1 15	0001 1000 0111 1001	
0x0104	jump 2	0011 1100 0000 0010	
0x0106	stop	1000 0000 0000 0000	
0x0108	li a2 0	0001 1000 0000 0010	# # multiplication # # arguments in a0 and a1 # # result in a2 # # i in a3 # # nothing on stack
0x010A	li a3 0	0001 1000 0000 0011	
0x010C	# geq a3 a1 4	1110 0001 0000 1011	# # if i >= b, leave loop and return
0x010E	add a2 a2 a0	0000 0000 0001 0010	# # result += a
0x0110	addi a3 a3 1	0010 0000 0101 1011	# i++
0x0112	chgpci -3	0111 1111 1111 1101	# loop back
0x0114	addi a0 a2 0	0010 0000 0001 0000	# # store return value in a0
0x0116	return 0	0011 1000 0000 0000	

Backup & Restore of Saved Registers

Note that this is a flawed program that does not work when the input is negative. This is reflected in the assembled program as well.

```
void main() {
	int seriesSum = getSeriesSum(7);
}

int getSeriesSum(int in) {
	if (in == 0) {
		return in;
	}
	int myNum = in;
	in = in - 1;
	return (myNum + getSeriesSum(in));
}
```

Address	Assembly	Machine Code	Comments
0x0100	li a0 7	0001 1000 0011 1000	
0x0102	jump 2	0011 1100 0000 0010	
0x0104	stop	1000 0000 0000 0000	
0x0106	li a1 0	0001 1000 0000 0001	
0x0108	neq a0 a1 2	1100 0000 1000 1000	
0x010A	return 0	0011 1000 0000 0000	
0x010C	chgspi -2	0111 1011 1111 1110	# # allocate space on stack
0x010E	strsp s0 1	0100 0000 0000 1101	# # back up mynum
0x0110	addi s0 a0 0	0010 0000 0000 0101	# in = in - 1
0x0112	addi a0 a0 -1	0010 1111 1100 0000	
0x0114	jump -6	0011 1111 1111 1010	# call itself
0x0116	add a0 a0 s0	0000 0001 0100 0000	# add mynum to return value
0x0118	rtvsp s0 1	0101 0000 0000 1101	# put back old s0
0x011A	return 2	0011 1000 0000 0010	# go to parent

Appendix D: Instruction Description

Mnemonic	Type	Opcode	F1	F2	F3	F4	F7	F10	Description
ADD	4WRR	000	0	0	0	0			Dest = Input 1 + Input 2
AND	4WRR	000	0	0	0	1			Dest = Input 1 AND Input 2
OR	4WRR	000	0	0	1	0			Dest = Input 1 OR Input 2
XOR	4WRR	000	0	0	1	1			Dest = Input 1 XOR Input 2
SUB	4WRR	000	0	1	0	0			Dest = Input 1 - Input 2
NAND	4WRR	000	0	1	0	1			Dest = Input 1 NAND Input 2
NOR	4WRR	000	0	1	1	0			Dest = Input 1 NOR Input 2
XNOR	4WRR	000	0	1	1	1			Dest = Input 1 XNOR Input 2
LU	2W	000	1	0					Dest[0:7] = IMMEDIATE[0:7] Dest[8:15] = Dest[8:15]
LL	2W	000	1	1					Dest = <SE>IMM
ADDI	1WR	001	0						Dest = Input 1 + <SE>IMM
SHIFT	2WR	001	1	0					Shift input 1 <SE> IMMEDIATE bits left and store in output. Perform a right shift if the immediate is negative. Fill with zeroes.
RETURN	3	001	1	1	0				PC = RA SP = SP + <SE>IMM << 1 RA = M[SP]*
JUMP	3	001	1	1	1				M[SP] = RA RA = PC+2 PC = PC + <SE>IMM << 1
STRSP	1R	010	0						M[SP + <SE>IMM << 1] = Input1
RTVSP	1W	010	1						Dest = M[SP + <SE>IMM]
STR	7RR	011	1	0	0	0	010		M[Input 1] = Input 2
RTV	7WR	011	1	0	0	0	011		Dest = M[Input 1]

READ	10W	011	1	0	0	0	100	000	Output = IN
WRITE	10R	011	1	0	0	0	100	001	OUT = Input 1
GETSP	10R	011	1	0	1	0	000	000	Dest = SP
CHGSP	10W	011	1	0	1	0	100	000	SP = SP + Input 1
SETSP	10W	011	1	0	1	0	100	001	SP = Input 1
GETPC	10R	011	1	0	1	1	000	000	Dest = PC
CHGPC	10W	011	1	0	1	1	100	000	PC = PC + Input 1
SETPC	10W	011	1	0	1	1	100	001	PC = Input 1
CHGSPI	3	011	1	1	0				SP = SP + <SE>IMM
CHGPCI	3	011	1	1	1				PC = PC + <SE> IMM
EQ	0RR	100							If Input 1 = Input 2 PC = PC + <SE>IMM << 1
LT	0RR	101							If Input 1 < Input 2 PC = PC + <SE>IMM << 1
NEQ	0RR	110							If Input 1 != Input 2 PC = PC + <SE>IMM << 1
GEQ	0RR	111							If Input 1 >= Input 2 PC = PC + <SE>IMM << 1