CSSE 232 Project Team 1B Design Document

Samuel Munro, Alyssa Russell, Jessica Russell

Table of Contents

JAHS Assembly Language

The overall architecture design will be implementing an accumulator style operating system, which utilizes an implicit accumulator register. This system will work by calling values to and from the stack with each instruction and manipulating the accumulator register, as well as any other registers used. This system utilizes a dedicated accumulator in order to keep instruction size low, as well as general purpose registers to aid the programmer and shorten programs. For instance, [A] type instructions primarily use the accumulator and [I] type instructions allow use of other registers. Instruction types are discussed in more detail below.

Register Name	Number	Usage	Stored across a call?
$0	0	constant 0	N/A
$r0	1	procedure return 0	No
$r1	2	procedure return 1	No
$a0	3	argument 0	No
$a1	4	argument 1	No
$a2	5	argument 2	No
$a3	6	argument 3	No
$acc	7	implicit accumulator	No
$t0	8	temporary register 0	No
$t1	9	temporary register 1	No
$t2	10	temporary register 2	No
$s0	11	Saved temporary register 0	Yes
$s1	12	Saved temporary register 1	Yes
$cr	13	comparison register. Stores the results of bnez and bez instructions. Details below.	No
$sp	14	stack pointer	Yes
$ra	15	return address (used by function call)	No

1. Register $cr (13): Stores the result of set less than operations for use in conditional branches. Not saved over procedure calls.
2. Registers $a0-$a3 (3-6) are used to pass the first four arguments to procedures. Any other arguments are passed on the stack. Registers $r0 and $r1 (1, 2) are used to return values from procedures. Neither $a or $r registers are preserved across a call.
3. Register $sp (14) points to the last location on the stack. Register $ra (15) is written to by the srj instruction.
4. Temporary Registers $t0-$t2 (8-10): Registers used by the program during execution. Not stored across a procedure call.
5. Saved Temporary Registers $s0-$s1 (11-12): Registers used by the program during execution. Stored across a procedure call.
6. Register $pc: Stores the address (from the code segment) of the current instruction being executed. Not part of the register file since it is never directly referred to by an instruction.
7. Memory is addressed at every 16-bit chunk. This means that in the programs, PC + 1 is where the next instruction is located..

Syntax and Semantics of Instruction Set

Addition (A)

```
add A
```

Adds value in register A to whatever is currently stored inside the accumulator, and stores the result into the accumulator.

Subtraction (A)

```
sub A
```

Subtracts value in register A from whatever is stored inside the accumulator, and stores the result into the accumulator.

Load from memory (I)

```
lm A, off
```

Loads value of A from the memory location A plus the signed immediate offset off into the accumulator register.

Store into memory (I)

```
sm A, off
```

Stores contents of the accumulator into memory location A plus the signed immediate offset off.

Set Less Than (A)

```
slt A
```

Determines if A is less than the value stored in the accumulator register and stores the result in the comparison register: 0 if A is greater than the accumulator value and 1 if A is less than the accumulator value.

Set Greater Than (A)

sgt A

Determines if A is greater than the value stored in the accumulator register and stores the result in the comparison register: 0 if A is less than the accumulator value and 1 if A is greater than the accumulator value.

Branch Equal to Zero (B)

bez L

Sets the program counter to label L if the comparison register is equal to zero.

Branch Not Equal to Zero (B)

benz L

Sets the program counter to label L if the comparison register is not equal to zero.

Set Immediate (I)

si rs, imm

Sets the value of a register to an immediate.

Set Register (A)

sr R

Sets the value of the accumulator to that of a register R.

Copy Register (A)

cr R

Sets the value of register R to that of the accumulator.

Add Immediate (I)

addi rs, imm

Performs the ADD operation on the sign-extended immediate and the value in register rs, and stores the result in register rs.

Set Return and Jump (B)

srj L

Sets the $ra register to $PC + 1 and jumps to label L by setting $pc to the label L.

Jump (B)

j L

Jump to label L

Jump Register (A)

jr A

Jump to address stored in register A

Check Equality (A)

eq A

Evaluates the equality of the value within register A to the value stored in the accumulator, and stores the result into the comparison register: 1 if they are equal, 0 if they are not.

Load Upper Immediate (I)

lui A, imm

Loads the passed immediate into the most significant 8 bits of register A, and clears the lower 8 bits.

Or Immediate (I)

ori rs, imm

Performs bitwise OR on the zero-extended immediate and the register rs, and stores the result in register rs.

And Immediate (I)

andi rs, imm

Performs bitwise AND on the zero-extended immediate and the value in register rs, and stores the result in register rs.

Or (A)

or A

Performs bitwise OR on register A and the accumulator, and stores the result in the accumulator.

And (A)

and A

Performs bitwise AND on register A and the accumulator, and stores the result in the accumulator.

Shift Left Immediate (I)

sli A, imm

Performs bitwise left shift by the zero-extended immediate imm on the value stored in the accumulator, and stores the result in register A.

Shift Right Immediate (I)

sri A, imm

Performs bitwise right shift by the zero-extended immediate imm on the value stored in the accumulator, and stores the result in register A.

Read Input (A)

ri A

Reads outside input into register A.

Program Ready (A)

pr

Reads program ready signal from control and uses it to start program execution. Stores into CR.

Instruction Formats

Accumulator Type (A):

opcode (4)	rs (4)	FUNCT (8)

Immediate Type (I):

opcode (4)	rs (4)	imm (8)

Branch Type (B)

opcode (4)	address (12)

i) SignExtImm = {8{immediate[7]}, immediate}
ii) JumpAddr = {PC+1[15:12], address}
iii) ZeroExtImm = {8'b0, immediate}
iv) BranchAddr = PC+1+ {4{address[11]}, address}

Machine Language Translation Instructions

1. Opcodes and function codes are translated into binary
2. Registers are translated by their numbers into binary
3. Immediates are directly translated (Use sign extended addressing, however. Refer to symbolic definitions)
4. Branch addresses are calculated by adding the address to PC + 1.

   Destination = PC + 1 + (address)

5. Jump addresses are calculated by appending the address to the first four bits of PC + 1.

Examples:

1. add $sp - 0000 1110 0000 0000
2. ori $t0, 34 - 1001 1000 0010 0010
3. bez LABEL - 0100 0000 0000 0010 (This assumes LABEL is 2 instructions away from PC+1)
4. jr $acc - 0000 0111 0000 1010
5. j LABEL - 0111 0000 0001 1000 (This assumes LABEL is at address 0000 0000 0001 1000)

Procedure Calling Conventions

When calling a procedure:

1. Pass arguments using $a0-$a3 registers. If there are more than 4 arguments, pass them to the stack.
2. Back-up registers that are not preserved across a call. These would be the $a registers, $ra register, temporary registers, and the current accumulator.
3. Perform an srj instruction, which jumps to a procedure's first instruction and saves the return address into $ra.
4. Once returned, restore $a0-$a3, $t0-$t1, and $ra and deallocate stack space.

Inside the callee:

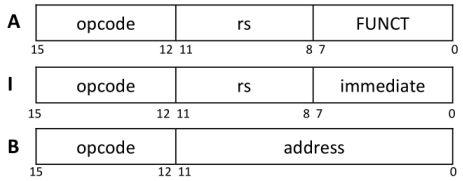
1. Perform operations and repeat caller steps if the procedure calls another.
2. Allocate stack space
3. Backup saved temporary values onto the stack
4. Place return values in $r0 and $r1
5. Restore saved temporaries
6. Deallocate any reserved stack space
7. Return by jumping to $ra

Reference Card

JAHS Reference Data

NAME, MNEMONIC		FORMAT	OPERATION [in Verilog]	OPCODE/FUNCT$_{(hex)}$
Addition	add	[A]	R[7] = R[7] + R[rs]	$0/0_{hex}$
Add Immediate	addi	[I]	R[rs] = R[rs] + SignExtImm	11_{hex}
Subtraction	sub	[A]	R[7] = R[7] - R[rs]	$0/1_{hex}$
Load from Memory	lm	[I]	R[7] = M[R[rs] + SignExtImm]	1_{hex}
Store into Memory	sm	[I]	M[R[rs] + SignExtImm] = R[7]	2_{hex}
Set Less Than	slt	[A]	R[13] = (R[rs] < R[7]) 1 : 0	$0/2_{hex}$
Set Greater Than	sgt	[A]	R[13] = (R[rs] > R[7]) 1 : 0	$0/3_{hex}$
Branch Equal to Zero	bez	[B]	if(R[13] == $0) PC = BranchAddr	3_{hex}
Branch Not Equal to Zero	benz	[B]	if(R[13] != $0) PC = BranchAddr	4_{hex}
Set Immediate	si	[I]	R[rs] = SignExtImm	5_{hex}
Set Register	sr	[A]	R[7] = R[rs]	$0/4_{hex}$
Copy Register	cr	[A]	R[rs] = R[7]	$0/5_{hex}$
Set Return and Jump	srj	[B]	R[15] = PC + 1; PC = JumpAddr	6_{hex}
Jump	j	[B]	PC = JumpAddr	7_{hex}
Check Equality	eq	[A]	R[13] = (R[rs] == R[7]) 1 : 0	$0/6_{hex}$
Load Upper Immediate	lui	[I]	R[rs] = {imm, 8'b0}	8_{hex}
Or Immediate	ori	[I]	R[rs] = R[rs] | ZeroExtImm	9_{hex}
And Immediate	andi	[I]	R[rs] = R[rs] & ZeroExtImm	10_{hex}
Or	or	[A]	R[7] = R[rs] | R[7]	$0/7_{hex}$
And	and	[A]	R[7] = R[rs] & R[7]	$0/8_{hex}$
Jump Register	jr	[A]	PC = R[rs]	$0/9_{hex}$
Shift Left Immediate	sli	[I]	R[rs] = R[7] << ZeroExtImm	12_{hex}
Shift Right Immediate	sri	[I]	R[rs] = R[7] >> ZeroExtImm	13_{hex}
Read Input	ri	[A]	R[rs] = Outside Input	$0/10_{hex}$
Program Ready	pr	[A]	R[13] = Program Ready	$0/11_{hex}$

BASIC INSTRUCTION FORMATS



i) SignExtImm = {8{immediate[7]}, immediate}
ii) ZeroExtImm = {8{1b'0}, immediate}
iii) BranchAddr = PC + 1 + {3{address[11]}, address}
iv) JumpAddr = {PC+1[15:12], address}

REGISTER NAME, NUMBER, USE, CALL CONVENTION

NAME	NUMBER	USE	STORED ACROSS A CALL?
$0	0	The Constant Value 0	N/A
$r0-$r1	1-2	Procedure Return	No
$a0-$a3	3-6	Arguments	No
$acc	7	Implicit Accumulator	No
$t0-$t2	8-10	Temporaries	No
$s0-$s1	11-12	Saved Temporaries	Yes
$cr	13	Comparison Register	No
$sp	14	Stack Pointer	Yes
$ra	15	Return Address	No

Assembly Language Usage

Assembly Fragments

Name	Pseudocode	Assembly Equivalent

Conditional Statement	`if (a < b){`
	`  c = a + b;`
	`}`
		`  sr a  #set value of a into acc`
		`  slt b  #compare acc value to b`
		`  bez SKIP  #if $cr = 0, break`
		`  add b  #if not, add b to acc`
		`  sm c, 0  #store result into c`
		`SKIP:`

Loop	`while (a <= b){`
	`  a += 1;`
	`}`
		`sr a  #set value of a into accumulator`
		`LOOP:`
		`  slt b  #check if b is less than a`
		`  bnez NEXT  #if cr != 0, skip`
		`  addi, $acc 1  #else, add 1 to acc`
		`  j LOOP  #loop`
		`NEXT:`

Load second element of array stored at A	`x = A[1];`	`lm A, 1`

Allocate Stack and store a value	`Stack[0] = 5;`	`lui $sp, 3  #set up stack pointer`
		`ori $sp, 255`
		`addi $sp, -1  #subtract 1 to accumulator`
		`si $acc, 5  #set accumulator to 5`
		`sm $sp, 0  #store accumulator value`
		`sm $sp, 1  #store duplicate value`

Call a sub-procedure	`x = foo(y);`	`lui $sp, 3  #set up stack pointer`
		`ori $sp, 255`
		`si $a0, y  #set accumulator to y`
		`addi $sp, -1  #add -1 to accumulator`
		`sr $ra  #set accumulator to $ra`
		`sm $sp, 0`
		`sr $a0`
		`sm $sp, 1`
		`srj foo  # call foo`
		`lm $sp, 0 #load memory into accumulator`

		cr $ra #copy value into $ra lm $sp, 1 #load memory into accumulator cr $a0 addi $sp, 1 #add 1 to the accumulator sr $r0

Wait For Input	while(pr != 1){
	
	}
	//program
		start:
		pr $cr #check if program is ready
		benz next #if so, branch to program
		j start #else, jump to loop

Relative Primes Program in Java

```
// Find m that is relatively prime to n.
int relPrime(int n)
{
   int m;
   m = 2;
   while (gcd(n, m) != 1) {  // n is the input from the outside world
     m = m + 1;
   }
   return m;
}

// The following method determines the Greatest Common Divisor of a
and b using Euclid's algorithm.

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;
}
```

Program in JAHS Assembly Language

```
     lui $sp, 3                    #set up stack pointer
     ori $sp, 255
start:
     pr $cr                        #check if program is ready
     benz relprime                 #if so, branch to program
     j start                       #else, jump to loop
relprime:
     ri $a0                        #read outside input
     addi $sp, -1                  #allocate stack space
     sr $a0
     sm $sp, 0                     #store N into Memory
     si $a1, 2                     #initialize argument M to 2
LOOP1:
     sr $a1
     sm $sp, 1                     #store M into Memory
     srj GCD                       #jump to GCD
     lm $sp, 0                     #once returned, load N
     cr $a0
     lm $sp, 1                     #load M
     cr $a1
     si $acc, 1
     eq $r0
     benz FINISHED                 #if not while condition, finish
     addi $a1, 1                   #if while condition, increment M
     j LOOP1                       #jump back to loop
FINISHED:
     sr $a1                        #copy M into accumulator
     cr $r0                        #copy accumulator into R0
     j start                       #wait for new input
GCD:
     sr $a0                        #check initial condition
     eq $0
     benz BReturn
LOOP2:
     sr $a1                        #enter while loop
     eq $0
```

```
      benz AReturn                    #if while loop fails, return A
      sgt $a0                         #else, check a > b
      bez ELSE                        #if a !> b, branch to else
      sr $a0
      sub $a1                         #subtract b from a
      cr $a0
      j LOOP2
ELSE:
      sr $a1
      sub $a0                         #subtract a from b
      cr $a1
      j LOOP2
BReturn:                              #return condition for initial
      sr $a1
      cr $r0
      jr $ra
AReturn:                              #return condition for post-loop
      sr $a0
      cr $r0
      jr $ra
```

Machine Code Translation

PC Address	Hex	Assembly
0000 0000 0000 0000	8e03,	lui $sp, 3
0000 0000 0000 0001	9eff,	ori $sp, 255
0000 0000 0000 0010 (start)	000b,	pr $cr
0000 0000 0000 0011	4001,	benz relprime
0000 0000 0000 0100	7002,	j start
0000 0000 0000 0101 (relprime)	030a,	ri $a0
0000 0000 0000 0110	beff,	addi $sp, -1
0000 0000 0000 0111	0304,	sr $a0
0000 0000 0000 1000	2e00,	sm $sp, 0
0000 0000 0000 1001	5402,	si $a1, 2
0000 0000 0000 1010 (LOOP1)	0404,	sr $a1
0000 0000 0000 1011	2e01,	sm $sp, 1
0000 0000 0000 1100	6019,	srj GCD
0000 0000 0000 1101	1e00,	lm $sp, 0
0000 0000 0000 1110	0305,	cr $a0
0000 0000 0000 1111	1e01,	lm $sp, 1
0000 0000 0001 0000	0405,	cr $a1
0000 0000 0001 0001	5701,	si $acc, 1
0000 0000 0001 0010	0106,	eq $r0
0000 0000 0001 0011	4002,	benz FINISHED
0000 0000 0001 0100	b401,	addi $a1, 1
0000 0000 0001 0101	700a,	j LOOP1
0000 0000 0001 0110 (FINISHED)	0404,	sr $a1
0000 0000 0001 0111	0105,	cr $r0
0000 0000 0001 1000	7002,	j start
0000 0000 0001 1001 (GCD)	0304,	sr $a0
0000 0000 0001 1010	0006,	eq $0
0000 0000 0001 1011	400d,	benz BReturn
0000 0000 0001 1100 (LOOP2)	0404,	sr $a1
0000 0000 0001 1101	0006,	eq $0
0000 0000 0001 1110	400d,	benz AReturn
0000 0000 0001 1111	0303,	sgt $a0
0000 0000 0010 0000	3004,	bez ELSE
0000 0000 0010 0001	0304,	sr $a0
0000 0000 0010 0010	0401,	sub $a1
0000 0000 0010 0011	0305,	cr $a0
0000 0000 0010 0100	701c,	j LOOP2
0000 0000 0010 0101 (ELSE)	0404,	sr $a1
0000 0000 0010 0110	0301,	sub $a0
0000 0000 0010 0111	0405,	cr $a1
0000 0000 0010 1000	701c,	j LOOP2
0000 0000 0010 1001 (BReturn)	0404,	sr $a1
0000 0000 0010 1010	0105,	cr $r0
0000 0000 0010 1011	0f09,	jr $ra
0000 0000 0010 1100 (AReturn)	0304,	sr $a0
0000 0000 0010 1101	0105,	cr $r0
0000 0000 0010 1110	0f09,	jr $ra

Implementation

RTL Specification

Instruction	RTL
add R	PC = PC + 1 IR = Mem[PC] A = Reg[7] B = Reg[IR[11:8]] ALUOut = A + B Reg[7] = ALUOut
sub R	PC = PC + 1 IR = Mem[PC] A = Reg[7] B = Reg[IR[11:8]] ALUOut = A - B Reg[7] = ALUOut
slt R	PC = PC + 1 IR = Mem[PC] A = Reg[7] B = Reg[IR[11:8]] ALUOut = B slt A Reg[13] = ALUOut
sgt R	PC = PC + 1 IR = Mem[PC] A = Reg[7] B = Reg[IR[11:8]] ALUOut = A slt B Reg[13] = ALUOut
bez R	PC = PC + 1 IR = Mem[PC] A = Reg[7] B = Reg[IR[11:8]] if (CR == 0): PC = PC + SE(IR[11:0])

benz R	PC = PC + 1 IR = Mem[PC] A = Reg[7] B = Reg[IR[11:8]] if (CR != 0): PC = PC + SE(IR[11:0])
sr R	PC = PC + 1 IR = Mem[PC] A = Reg[7] B = Reg[IR[11:8]] Reg[7] = A
cr R	PC = PC + 1 IR = Mem[PC] A = Reg[7] B = Reg[IR[11:8]] Reg[IR[11:8]] = A
and R	PC = PC + 1 IR = Mem[PC] A = Reg[7] B = Reg[IR[11:8]] ALUOut = A & B Reg[7] = ALUOut
or R	PC = PC + 1 IR = Mem[PC] A = Reg[7] B = Reg[IR[11:8]] ALUOut = A | B Reg[7] = ALUOut
jr R	PC = PC + 1 IR = Mem[PC] A = Reg[7] B = Reg[IR[11:8]] PC = B
eq R	PC = PC + 1 IR = Mem[PC] A = Reg[7] B = Reg[IR[11:8]] ALUOut = ZERO Reg[13] = ALUOut;
addi A, imm	PC = PC + 1

	IR = Mem[PC] A = Reg[7] B = Reg[IR[11:8]] ALUOut = A + ZE(imm) Reg[IR[11:8]] = ALUOut
andi A, imm	PC = PC + 1 IR = Mem[PC] A = Reg[7] B = Reg[IR[11:8]] ALUOut = A & ZE(imm) Reg[IR[11:8]] = ALUOut
ori A, imm	PC = PC + 1 IR = Mem[PC] A = Reg[7] B = Reg[IR[11:8]] ALUOut = A | ZE(imm) Reg[IR[11:8]] = ALUOut
lui A, imm	PC = PC + 1 IR = Mem[PC] A = Reg[7] B = Reg[IR[11:8]] Reg[IR[11:8]][15:8] = IR[7:0], 8b'0
j L	PC = PC + 1 IR = Mem[PC] A = Reg[7] B = Reg[IR[11:8]] PC = (newPC[15:12], IR[11:0])
srj L	PC = PC + 1 IR = Mem[PC] A = Reg[7] B = Reg[IR[11:8]] PC = (newPC[15:12], IR[11:0]) Reg[15] = PC
si A, imm	PC = PC + 1 IR = Mem[PC] A = Reg[7] B = Reg[IR[11:8]] ALUOut = SE(imm) + 0 Reg[IR[11:8]] = ALUOut

lm R, off	PC = PC + 1 IR = Mem[PC] A = Reg[7] B = Reg[IR[11:8]] ALUOut = SE(off)+B Reg[7] = Mem[ALUOut]
sm R, off	PC = PC + 1 IR = Mem[PC] A = Reg[7] B = Reg[IR[11:8]] ALUOut = SE(off)+B Mem[ALUOut] = Reg[7]
sli A, imm	PC = PC + 1 IR = Mem[PC] A = Reg[7] B = Reg[IR[11:8]] ALUOut = A << ZE(imm) Reg[IR[11:8]] = ALUOut
sri A, imm	PC = PC + 1 IR = Mem[PC] A = Reg[7] B = Reg[IR[11:8]] ALUOut = A >> ZE(imm) Reg[IR[11:8]] = ALUOut
ri A	PC = PC + 1 IR = Mem[PC] A = Reg[7] B = Reg[IR[11:8]] Reg[IR[11:8]] = OutsideInput
pr	PC = PC + 1 IR = Mem[PC] A = Reg[7] B = Reg[IR[11:8]] Reg[13] = ProgramReady

RTL Summary Table

A type	cr/sr	lm/sm/jr	bez/benz	ori/andi/si/lui/sli/sri/addi	j/srj
PC = PC + 1					
IR = Mem[PC] A = Reg[7] B = Reg[IR[11:8]]					
ALUOut = A OP B eq: ALUOut = ZERO slt: ALUOut = B OP A		ALUOut = SE(off)+B jr: PC = B	bez: if (CR == 0): PC = PC + SE(IR[11:0]) benz: if (CR != 0): PC = PC + SE(IR[11:0])	ALUOut = A OP ZE(imm) si: ALUOut = SE(imm) + 0 sli: ALUOut = A << ZE(imm) sri: ALUOut = A >> ZE(imm)	srj: Reg[15] = PC
Reg[7] = ALUOut eq, slt, sgt: Reg[13] = ALUOut ri: Reg[IR[11:8]] = OutsideInput pr: Reg[13] = ProgramReady	sr: Reg[7] = A cr: Reg[IR[11:8]] = A	lm: Reg[7] = Mem[ALUOut] sm: Mem[ALUOut] = Reg[7]		Reg[IR[11:8]] = ALUOut lui: Reg[IR[11:8]][15:8] = IR[7:0] C 8b'0	PC = (PC[15:12], IR[11:0])

Component Descriptions

Component	Input Signals	Output Signals	Control Signals	Implementation
ALU	ALUIn1 (16) ALUIn2 (16)	ALUOut(16) Zero (1)	ALUOp (2)	
Register	Write (16)	Read (16)	N/A	PC, ALUOut, A, B, Intermediate Stage Registers
Register File	WriteReg (16) WriteData (16) ReadReg1 (4) ReadReg2 (4)	ReadData1 (16) ReadData2 (16)	RegWrite (1)	Reg
Data Memory	DataAddress(16) WriteData (16)	OutputData (16)	DataWrite (1)	Mem
Instruction Memory	InAddress (16)	OutputData (16)	DataWrite (1)	Mem
Left Shifter	InData (12) InData (16)	OutData (16)	NumBits (8)	<<
Right Shifter	InData (12) InData (16)	OutData (16)	NumBits (8)	>>
Sign Extender	InData (12)	OutData (16)	N/A	SE
Zero Extender	InData (8)	OutData (16)	N/A	ZE
Adder	AddIn1 (16) AddIn2 (16)	OutData (16)		Used for PC incrementation
Combiner	PCin (3) ImmIn(13)	newPC (16)		Used to create jump addresses
Control Unit	OP (4) FUNCT (8) reset	See below		Controls processor

- ALU:
  The ALU controls data values and what kind of changes are made to them. Depending on the opcode, the ALU can add, multiply, subtract, or divide the given inputs from each other, giving us a resulting output based on these functions.

- Adder:
  The Adder is used to adjust the Program Counter [PC]. If the adder receives an offset from an instruction, then the output will add it to the PC to reflect the change. Otherwise, if the cycle runs normally it will increment the PC to the next instruction in the program.

- Register File:
  The register file keeps track of information held inside of the registers being used in the program. Any data input is written into the registers, and outputs are the data that was read from these registers.

- Data Memory:
  The Data Memory component stores inputs for retrieval later. This then outputs this stored memory when the control signal calls for it.

- Instruction Memory:
  The Instruction Memory takes in the specific instruction, splits the instruction information, and then sends the information to the specified component. Depending on whether or not the instruction has a single register and an immediate, a single register, or just an immediate, the instruction memory will output each section to either the Register File or another specified component needed for the data path.

- ShiftLeft (Barrel Shifter):
  The ShiftLeft component takes in an immediate and shifts the value to the left a specified amount of bits.

- ShiftRight (Barrel Shifter):
  The ShiftRight component takes in an immediate and shifts the value to the right a specified amount of bits.

- Sign-Extend:
  If Sign extend should be used, this component takes in the input value to be sign-extended and appends copies of the most significant bit to the front until the rest of the empty bits are filled. The output would then result in the sign-extended version of the input.

- Zero-Extend:
  If zero extend should be used, this component takes in the input value to be zero-extended and appends zeros to the front until the rest of the empty bits are filled. The output would then result in the zero-extended version of the input.

- Combiner: Combines the top 4 bits of PC with the Left shifter sign extended immediate for the jump and set return and jump instructions. These are combined to output a 16 bit value that is loaded into PC.

- Control Unit: Controls the processor by reading the OP code of the current instruction and sending the necessary control signals (see below) to the rest of the machine.

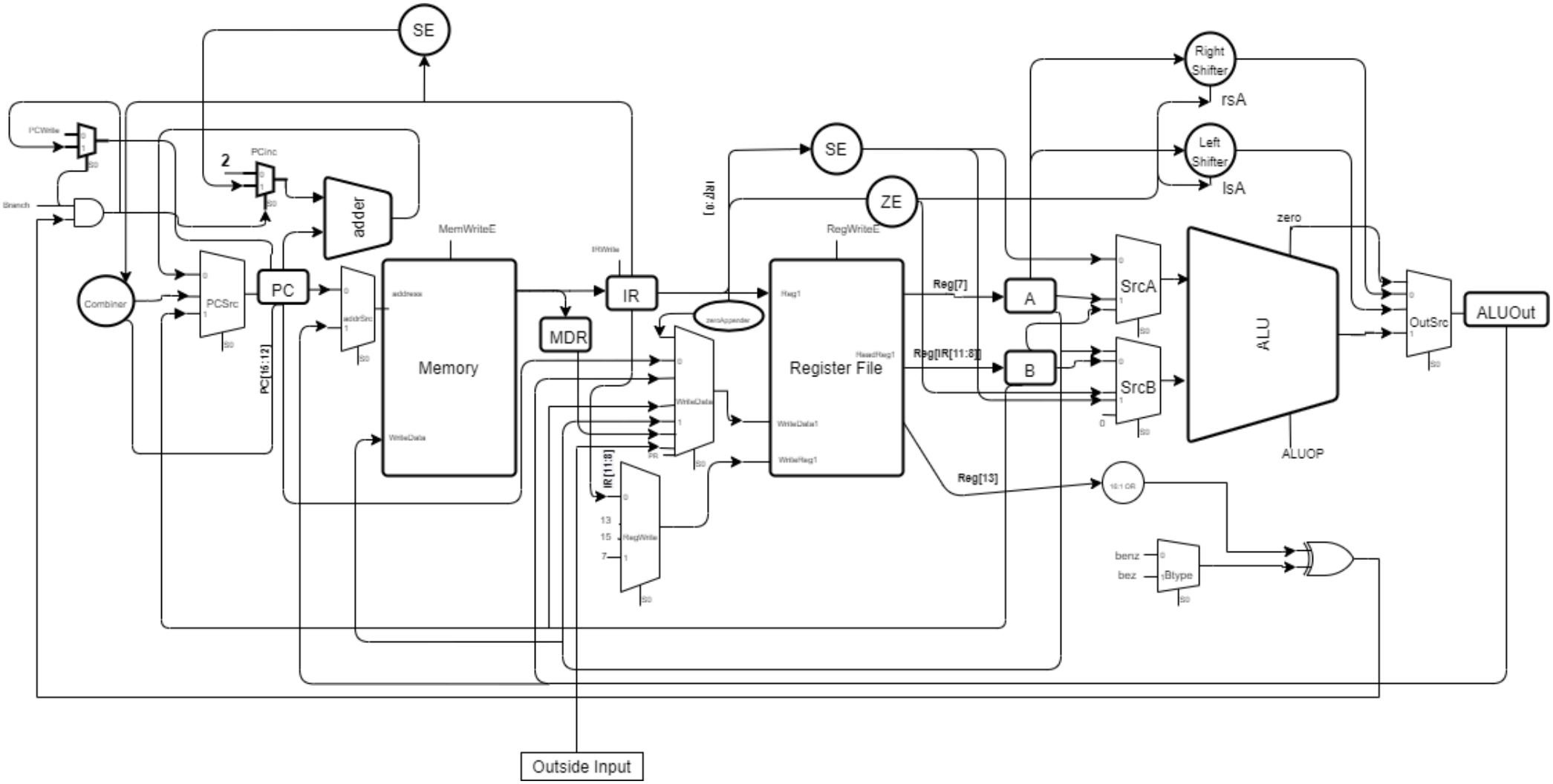
Overview of Our Error-Checking Process on the RTL

When checking our RTL for errors, we first made sure that all the bits were the correct numbers and the first three lines for each instruction in the RTL matched each other. We then double checked exactly what each instruction's function does and what that function affects, such as whether it takes anything from memory or it uses an immediate and the accumulator. We then matched up the bits with the rs, rt, accumulator, and other registers to make sure they were correctly completing the functions they were supposed to. This verified to us that the RTL is correct.

Note on Changes Made Since Milestone 1

We first fixed up some inconsistencies in the bits and instructions. We also added instructions for shifting to the right and to the left, updating the instruction sheet and adjusting the RTL, as well as the other related parts of this document to reflect this addition to our instruction list.

Datapath Block Diagram



Control Signal Descriptions

- PCSrc - chooses what is written into PC. The three options are: PC + increment, a jump address (defined above), or the value stored in register B. The last option is used for the jump register instruction.

- PCInc - chooses what PC is incremented by. The two options are: 2 and the left shifted sign extended immediate produced by the bez or benz instructions.

- addrSrc - chooses the read/write address for memory. This is PC or the address defined in register B, which implements sm and lm.

- MemWriteE - Write enable flag for memory.

- WriteData - Chooses what data is written into the register file. Chooses between: data read from memory, A register, B register, PC, zero-appended immediate, or ALUOut.

- RegWrite - Chooses which register data is written into. Chooses between: The accumulator, the comparison register, return address register, or the register defined in I-type instructions.

- RegWriteE - Write enable flag for the register file.

- SrcA - Chooses what goes into the first input of the ALU. The choices are: Register A, Register B (for sgt), and the sign extended immediate.

- SrcB - Chooses what goes into the second input of the ALU. The choices are: Register B, Register A (for sgt), zero, the sign-extended immediate, or the zero-extended immediate.

- OutSrc - Chooses what value ALUOut gets. The choices are: the ALU output, shift output (both left and right), and the result of the zero flag on the ALU.

- ALUOp - Decides what operation the ALU performs.

- PCWrite - Enables writing to the PC Register

- IRWrite - Enables writing to the IR Register

- MDRWrite - Enables MDR Register

- Btype - Indicates which type of branch is being used
- Branch - And's with the output of the comparison in order to set PC to either the branch target or not.
- regWriteExecute - Enables register writing for the A and B registers
- OutRegWrite - Enables writing of the ALUOut register

Component Implementation Plan

- ALU: We plan to use the standard ALU designs that are currently used for most applications. We will not be making a custom ALU.
- Register: We plan to use the standard Register designs that are currently used for most applications. We will not be making custom registers.
- Register File: We plan to use the standard Register File designs that are currently used for most applications. Our Register file will contain 16 registers that can be written to and read from. It will have dedicated outputs for the accumulator and the comparison register.
- Data Memory: We plan to use the standard designs that are currently used for most applications. Each address in memory will refer to 2 bytes, or 16 bits. Both Data and Instruction memory will be stored in the same block of DRAM.
- Instruction Memory: We plan to use the standard designs that are currently used for most applications. Each address in memory will refer to 2 bytes, or 16 bits. Both Data and Instruction memory will be stored in the same block of DRAM.
- Left Shifter: This component will shift the bits of the input signal to the right by an amount given by another input signal. Dynamic shifting will be done with a custom 16 bit barrel shifter.
- Right Shifter: This component will shift the bits of the input signal to the left by an amount given by another input signal. Dynamic shifting will be done with a custom 16 bit barrel shifter.

- Sign Extender: This component will be implemented by taking in the input signal, and setting the rest of the most significant bits to a copy of the most significant input bit.

- Zero Extender: This component will be implemented by taking in the input signal, and setting the rest of the most significant bits to 0.

- Adder: To implement this, we will create an adder circuit that adds two 16 bit input values, and outputs a 16 bit value. This was most easily done in Verilog.

- Combiner: This component will be implemented by taking two sets of input signals and combining them to output one 16 bit signal.

- Control Unit: To implement this, the logic will represent a table that we design, sending the correct control signals every time each instruction reaches the decode stage. In Verilog, it will represent a finite state machine.

Component Testing Plan

- ALU: Testing the ALU by itself requires that we are able to change at least the inputs and opcode which would allow us to see if the ALU properly functions. This includes testing operations such as adding and subtracting (or adding negative numbers) by checking if the ALU first can properly take in inputs and change them, and then checking if the opcodes correctly match the operations they are completing on the inputs. We also need to be aware if we need to be able to change registers or values at certain addresses.

- Register: To test a single register, we need to be sure that the component can read and write to the register. To test this specifically, we need to check if the input writes into memory, allowing it to be stored for later use, and then we need to try to read that input from the memory and make sure the value isn't changed or affected in any way during the transfer.

- Register File: A register file should be allowed to be accessed using register addresses. A good way to test the register file would be to put multiple inputs into the file and reading these values in different orders, as well as changing these values and reading them to make sure the addresses keep track of the input-to-register values and there is no form of mixing or value changes due to old values previously existing in the registers.

- Data Memory: This would be similar to testing the register and register file component. By allowing the data memory to either have information stored by address or register (or both), we can input values into the data memory to write the values into the memory addresses. We will then test if the data writing succeeded and read out the data to make sure it is the same value as what we originally wrote in. To thoroughly test this memory, we will need to repeat the data writing and reading process several times over several different addresses, checking the furthest addresses that data can be read or written to.

- Instruction Memory: To test instruction memory, it would be easier to see errors if certain instructions were already connected and set up with this component, like an ALU to check if the right operations were performed on inputs. However, we can test if the right instructions are being called by writing addresses into the instruction memory and simply reading out the same addresses to make sure they aren't changed when they are written or read out. We then just need to consistently check operations that are set up to certain instruction addresses to make sure each instruction performs the expected operation due to the correct address call.

- Left Shifter: Just like testing the ALU, we need to make sure that this component can take in inputs and change it/them accordingly. Since this has a very specific function, left shifting the bits, we just need to allow for one input that needs to be shifted, one input for how much the first input is shifted, and an output for the result. By checking to make sure that the results are simply the left shifted values of the inputs based on the amount they should be left-shifted, we can check to make sure this is working properly.

- Right Shifter: As we test the left shifter component, we do the same for the right shifter, except we just need to check that the input value-to-be-shifted is shifted to the right the correct amount of bits based on the second input.

- Sign Extender: To test the sign extender properly, we just need to check that not only is the value properly extended in the number of bits to the size it needs to be, but also that the value, in numbers only, isn't changed and is signed the same as the input.

- Zero Extender: To test the zero extender properly, we need to check that not only is the value properly extended in the number of bits to the size it needs to be, but also that the value isn't changed.

- Adder: We will test this by adding different positive and negative numbers, and verifying that the outputs are correct.

- Combiner: The combiner will need to test if the 2 input signals combine to an output signal that contains both of the inputs in the correct order.

- Control Unit: The control unit will be tested when connected with at least one or more other components that have already been checked to work.

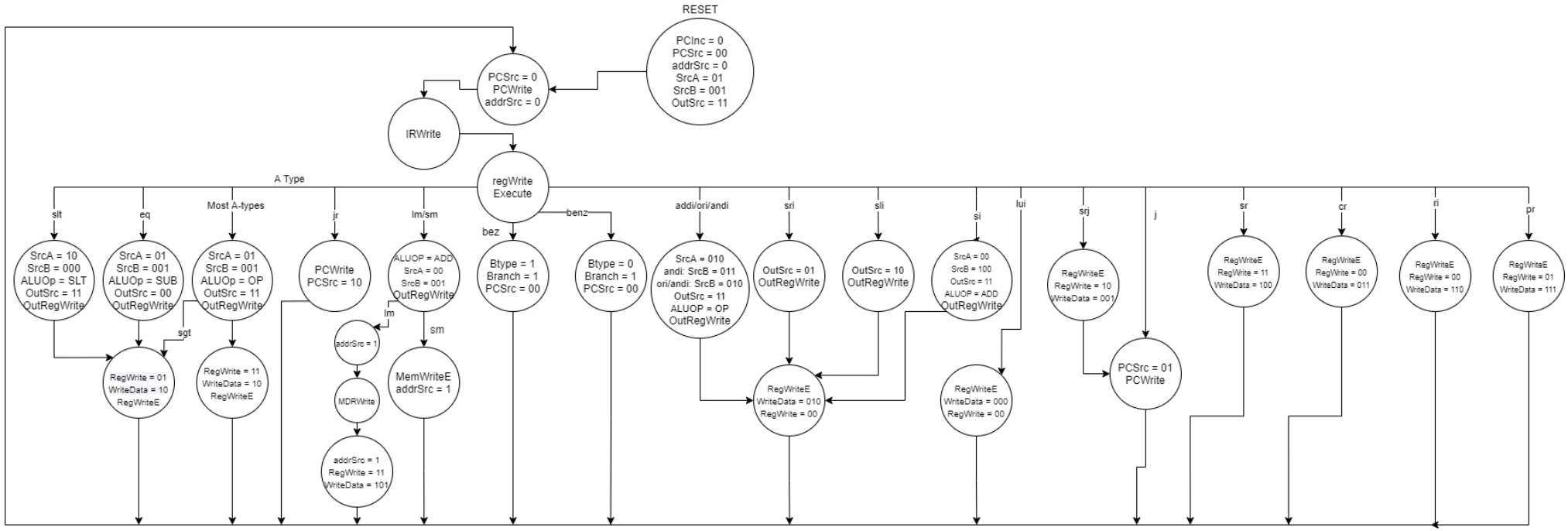
Integration Plan

For integrating the smaller pieces into the larger subsystems that will eventually come together to be the full microcomputer, we will start by connecting the most similar pieces together that accomplish similar functions. This includes connecting pieces such as the shifters and ALU together, as well as the instruction memory to allow for tests of these functions. Since our processor is multicycle, we will integrate each cycle's components sequentially, starting with PC increments, then register reads, then calculations, memory interactions, and finally register writes. We will just need to be aware of any control units used in specific portions of the subsystems so each subsystem can be tested and verified to work as they should, with and without the rest of the other subsystems.

Note on Changes Made Since Milestone 2

For this milestone, we adjusted all of the RTL for each instruction, after realizing that the best idea would be to make each instruction be completely consistent for the first two instructions. This ended up making some instructions longer, but made the whole set more consistent.

Control State Diagram



Control Implementation Plan

First we will identify the types of control signals we will need and what each will be used for so that we can properly place them when they are implemented. After each control is identified, we will go through each of the inputs and outputs that will be for the controls and start labeling them as we work out how they fit into the operations of the control units. We will draft up each unit individually from there on Verilog, and unit test each of them using other parts that have already been tested and have been verified to work.

Note on Changes Made Since Milestone 3

Adjustments were made to the datapath to implement lui correctly, as well as add a dedicated output for the comparison register, and branch logic using an XOR gate and an AND gate to decide whether to take a branch or not. It also seems we missed adding any RTL for addi, so that is now added. Added to the integration plan to keep in mind about control unit integration.

Performance Metrics

Memory Usage: (46 Instructions + 2 Stack locations) * 2 Bytes = **96 Bytes**

Total number of instructions executed for input 0x13B0: **91901**

Maximum frequency (defined by Xilinx) : **17.782 MHz**

Minimum period (defined by Xilinx) : **50.237 ns**

Instruction Breakdown:

Instruction Type	Cycles per Instruction	Number of Instructions	Cycles
A-type	5	30591	152955
I-type Arithmetic	5	22	110
Branch	4	20404	81616
Jump	4	10206	40824
Lui	4	2	8
Srj	4	10	40
Data Copy	4	30635	122540
Sm	5	11	55
Lm	7	20	140
Totals:		**91901**	**398288**

Average CPI of the machine running with input 0x13B0: 398288/91901 = **4.334 CPI**

Execution time:

$ET = \#instructions * Cycles/Instruction * 1\ second/Cycles$

$ET = 91901\ instructions * 4.334\ CPI * 1\ second/(17.782 * 10^6)\ Cycles$

$ET = 0.0224\ seconds$= **22.4 Milliseconds**

Device Utilization Summary

Logic Utilization	Used	Available	Utilization
Total Number Slice Registers	359	9,312	3%
Number used as Flip Flops	341		
Number used as Latches	18		
Number of 4 input LUTs	720	9,312	7%
Number of occupied Slices	511	4,656	10%
Number of Slices containing only related logic	511	511	100%
Number of Slices containing unrelated logic	0	511	0%
Total Number of 4 input LUTs	728	9,312	7%
Number used as logic	720		
Number used as a route-thru	8		
Number of bonded IOBs	56	232	24%
Number of RAMB16s	1	20	5%
Number of BUFGMUXs	1	24	4%
Number of RPM macros	3		
Average Fanout of Non-Clock Nets	3.55		