Design Document

Abe Gizaw, Liz Fogarty, Havalock Yin, Jay Smith
Instructor: Robert Williamson
CSSE 232

Table of Contents

High-Level Description

We are designing an accumulator architecture. We will be using four accumulators to reduce the amount of instructions that can access the stack/memory and keep track of more variables. Some of our instructions take a 2-bit ID value to specify which accumulator is used for an operation. We have 32 instructions and six types. All our procedures take advantage of the stack, by using it as a place to store the return address, accumulator registers, and arguments. We also use a heap pointer to save Accumulator values before going to another procedure.

The reason we chose to do 4 accumulators instead of a load store or 1 accumulator is that we wanted to find the best balance between the cost and speed of the processor. 4 accumulators make our processor a lot cheaper than load-store, but faster than a 1 accumulator architecture.

Performance Measures

We agreed that we would be measuring “performance” based on a combination of the speed and cost of the processor. Our design utilizes a 4-accumulator architecture, finding a balance between speed and cost. While not as fast as a load/store architecture, it surpasses the performance of a 1-accumulator setup. Conversely, it's more cost-effective compared to load/store systems, though slightly more expensive than a single accumulator design. Hence, we've determined that assessing "performance" based on cost-effectiveness is the most appropriate measure for our purposes.

(Put hard numbers for cost)

Description of Registers

We will use four accumulators which are just four registers. The names for the registers go as follows: reggie 0 represented as r0, reggie 1 represented as r1, reggie 2 represented as r2, and reggie 3 represented as r3. Two registers have specific assignments: r2 is allocated for lui values and r3 is allocated for return values. Both

these registers can be used for other operations, but whenever a lui/procedure is performed, whatever values were in these registers previously will be overwritten. The registers r0 and r1 will operate like normal accumulator registers.

Register	Name	Symbol	Assignment
R[0]	reggie 0	r0	Normal Accumulator Register
R[1]	reggie 1	r1	Normal Accumulator Register
R[2]	reggie 2	r2	Allocated for lui Values
R[3]	reggie 3	r3	Allocated for Return Values

Procedure Calling Conventions

- Calling Procedure

When a procedure is called, the user must open up space on the stack using movesp. movesp should only open up 2^9 bits, although its immediate can hold more. The return address will always be put on index 0 of the stack by jal, and the user does not have to do it themselves. If a user decides to open up more, it is up to them to make sure things are put in the correct order. Arguments must be placed in order of how a procedure expects it. So a procedure that takes (a, b, c) expects a at 0x00, b at 0x02, c at 0x4, etc. Another thing to keep in mind is that movesp takes in an 11-bit immediate, but loadsp and storesp only take a 7-bit immediate. This means you can open up more space than you can access. The user should be aware of this, although we don't ever expect the stack to be opened up that much in one instruction. The user is limited to

- At the start of a Procedure

Users shall load the arguments from the stack using loadsp. A 2-byte space needs to be opened if jal is used to jump to tags in this procedure.

- Returning From Procedure

The return value will be put in R[3]. Use instruction jb to jump back to the caller. Any extra return values will be put on the stack. Close the 2-byte space opened for jal to tag if there is before any jb.

- After Called Procedure Returned

Unpack any data or return value saved in the stack. Then, close the stack opened for the procedure and the extended stack for multiple return values.

- Preventing Data Loss

Whenever jumping to a procedure using jal, R[0], R[1], and R[2] will automatically be put on the stack. So space will be allocated, reggies saved, and whenever returning to the caller, the stack will be deallocated during the jb instruction. reggie3 is not saved since it holds the return value. Whenever a jb is called, reggies get automatically saved and restored. If other data needs to be preserved (for example the static data) open more space on the stack when calling a procedure and put the data after the arguments.

- Behind the scenes

Whenever a loadsp or storesp instruction is called, the user does not need to account for the registers that will be on the stack. Behind the scenes, 8 will always be added to the immediate that the user passes in. So loadsp r1 0, will move the sp 8 and load that value. This allows for the program to stay intuitive while reducing the amount of registers needed to save off values.

Instruction Set

Instruction	Opcode	Format Type	Bit-wise Description
add	00000	R	R[ID] = R[ID] + M[ME(imm)]

sub	00001	R	R[ID] = R[ID] - M[SE(imm)]
xor	00010	R	R[ID] = R[ID] ⊕ M[ME(imm)]
or	00011	R	R[ID] = R[ID] | M[ME(imm)]
and	00100	R	R[ID] = R[ID] & M[ME(imm)]
sll	00101	R	R[ID] = R[ID] << M[ME(imm)]
srl	00110	R	R[ID] = R[ID] >> M[ME(imm)]
stop	00111	J	PC = 0x00
ori	01000	I	R[ID] = R{ID] | SE(imm)
xori	01001	I	R[ID] = R{ID] ^ SE(imm)
andi	01010	I	R[ID] = R{ID] & SE(imm)
addi	01011	I	R[ID] = R[ID] + SE(imm)
slli	01100	I	R[ID] = R[ID] << imm[4:0]
srli	01101	I	R[ID] = R[ID] >> imm[4:0]
srai	01110	I	R[ID] = R[ID] >> imm[4:0]
slt	01111	R	R[ID] = (R[ID] < SE(imm))?1:0
beq	10000	B	if (R[ID1] == R[ID2]) PC += SE(imm2) << 1
bne	10001	B	if (R[ID1] != R[ID2]) PC += SE(imm2) << 1

blt	10010	B	if (R[ID1] < R[ID2]) PC += SE(imm2) << 1
bge	10011	B	if (R[ID1] >= R[ID2]) PC += SE(imm2) << 1
jal	10100	J	M[SP] = PC + 4 PC += SE(imm) << 1
load	10101	R	R[ID] = M[SE(imm)]
store	10110	R	M[SE(imm)] = R[ID]
storesp	10111	I	R[ID] = M[SP + SE(imm) + 6]
loadsp	11000	I	M[SP + SE(imm) + 6] = R[ID]
movesp	11001	J	SP += SE(imm)
input	11010	J	Mem[SP + SE(imm)] = input
lui	11011	J	R[2] = SE(imm) << 5
jb	11100	J	PC = M[SP]
swap	11101	C	hold = R[ID1] R[ID1] = R[ID2] R[ID2] = hold
alter	11110	A	R[returnID] = R[argID1] + R[argID2]
output	11111	J	Output = R[3]

Syntax and Sematix of Instructions:

add r1 0x188	This instruction will add the value stored at an address into a specified register

sub r2 0x188	This instruction will subtract what is in register ID by the passed-in address
xor r1 0x188	This instruction will do an XOR with the passed in address and the specified register
or r2 0x188	This instruction will do an OR with the passed in address and the specified register
and r2 0x188	This instruction will do an AND with the passed in address and the specified register
sll r1 0x188	Shitf the ID register to left by the value stored in the address
set r1 2	Set the ID register to immediate
stop 0	Stops the program and resets PC back to its initial state (0x00)
ori r1 1	This instruction will do an OR with the passed in immediate and the specified register
xori r1 0	This instruction will do an XOR with the passed in immediate and the specified register
andi r1 1	This instruction will do an AND with the passed in immediate and the specified register
addi r2 3	This instruction will add the imm to the ID register
slli r2 2	Shifts the ID register left by the value stored in the immediate
srli r2 2	Shifts the ID register right by the value stored in the immediate
srai r2 2	Shifts the ID register right by the value stored in the immediate and keeps the sign
slt r3 0x188	This instruction will evaluate the current value in the ID register to an address. If the current value in the register is less than the value at the address it returns 1 to the register otherwise it will return 0
beq r1 r2 0x1888	This instruction will evaluate if R[ID1] == R[ID2] , if so it will jump to some address
bne r1 r2 0x1888	This instruction will evaluate if R[ID1] != R[ID2] , if so it will jump to some address
blt r1 r2 0x1888	This instruction will evaluate if R[ID1] <R[ID2] , if so it will jump to some address
bge r1 r2 0x1888	This instruction will evaluate if R[ID1] >= R[ID2] , if so it will jump to some address
jal 0x1888	Jump to some address and save the address of the next line at index 0 of stack for return. Also saves of

	current values of registers onto the heap
load r1 0x188	This instruction will load the value at an address into the ID register.
store r1 0x188	This instruction will store the value in ID register into some address in memory
storesp r1 0x4	This instruction takes an offset and a register and stores the value at the register onto the stack + offset
loadsp r1 0x4	This instruction takes an offset and a register, and loads the value at stack + offset+ 8 onto the specified register
movesp 8	This instruction will take an immediate and move the stack pointer that much bits
input 0	Takes the immediate and puts some input at the stack pointer + the immediate. Effectively treating it like an offset
lui 0x7AA	loads an 11-bit immediate value into the upper 11 bits of R[2] and fills in the other 5 bits with 0's.
jb 0x0	Restore the three registers. Then, jump to the caller's return address, i.e. the ra stored at 0 index of the stack. The immediate DOES NOT do anything.
swap r1 2	Swaps the values in 2 registers
alter r2 2 r1 +	Takes in 3 IDs, return ID, arg1 ID, and arg2 ID. Also takes in an operation represented by some immediate. This instruction will calculate R[ID1] operation R[ID2] and store it in R[ReturnID]
output 0x0	Outputs what's in R[3]

Addressing Modes

- General Addressing mode

df	15 11	10 9	8 7	6 5	4 0
R	Addy			ID	Opcode
I	Immediate			ID	Opcode
B	Immediate		ID2	ID	Opcode

J	Immediate				Opcode
C	n/a		ID2	ID	Opcode
A	Operation	ReturnID	ID2	ID	Opcode

Our machine architecture supports several addressing modes to accommodate the diverse needs of our instruction set. Below we outline the addressing modes available and their application in our system, which is based on 16-bit instructions. Each instruction has a 5-bit opcode.

R Type:

- The R type instructions use a 2-bit ID field to specify one of the four registers in our register file as an operand.
- These instructions utilize an Addy field to access a specific location in our static memory, starting at address 0x1E00. Due to the limited size of our instructions, the upper 7 bits of the static memory address are hardcoded, and only the lower 9 bits are specified by the instruction, allowing access to a range within a 512-byte window. We call this Address extending.

I Type:

- I type instructions that feature an immediate value that is sign-extended to the full width of the data path.
- For loadsp and storesp instructions, the immediate is treated as stack pointer-relative offset. It is sign-extended to 16 bits, and when it increments the stack pointer, another 6 will be added. This is due to the registers being put on the stack at the first 3 spots.

B Type:

- Branch instructions use a PC-relative addressing mode where the immediate field specifies an offset from the current program counter. This offset is sign-extended to allow for both forward and backward branching within a limited range. The effective branch target is calculated by adding the sign-extended immediate to the current PC value.

- The immediate in the B-type is the exact amount of lines forward/backward it needs to jump. This number will be multiplied by 2 in the Immediate Generator

J Type:

- For J type instructions an immediate value is used to calculate one of the following:
    - The memory address offset where data will be stored.
    - Jump target address.
- Unlike B types, J types have an 11-bit Addy field, allowing for a broader range of target addresses. The jump target address is calculated by sign-extending the Addy field and adding it to the current PC value, facilitating longer jumps within the program. This is relevant to jb and jal
- The immediate in the J-type is the exact amount of lines forward/backward it needs to jump. This number will be multiplied by 2 in the Immediate Generator
- LUI Instruction: The lui instruction is a special case in our J types where the immediate is not sign-extended to 16 bits. This instruction sign extends to 11 bits which is then used to load a value into the upper bits of a register. It is left shifted by 5 bits in the Immediate Generator
- Jump Back has an immediate field that won't be used. It will read the top thing in the stack, which should be the address to jump back to. Users should always pass in 0 as the immediate field
- Output and stop are all J types that don't care about their immediate just like jb
- movesp takes its immediate and moves the stack pointer that much bits
- input has an immediate that acts as an offset for SP, but uses that offset to read from memory where to put some input at. It does not change SP

C and A Types

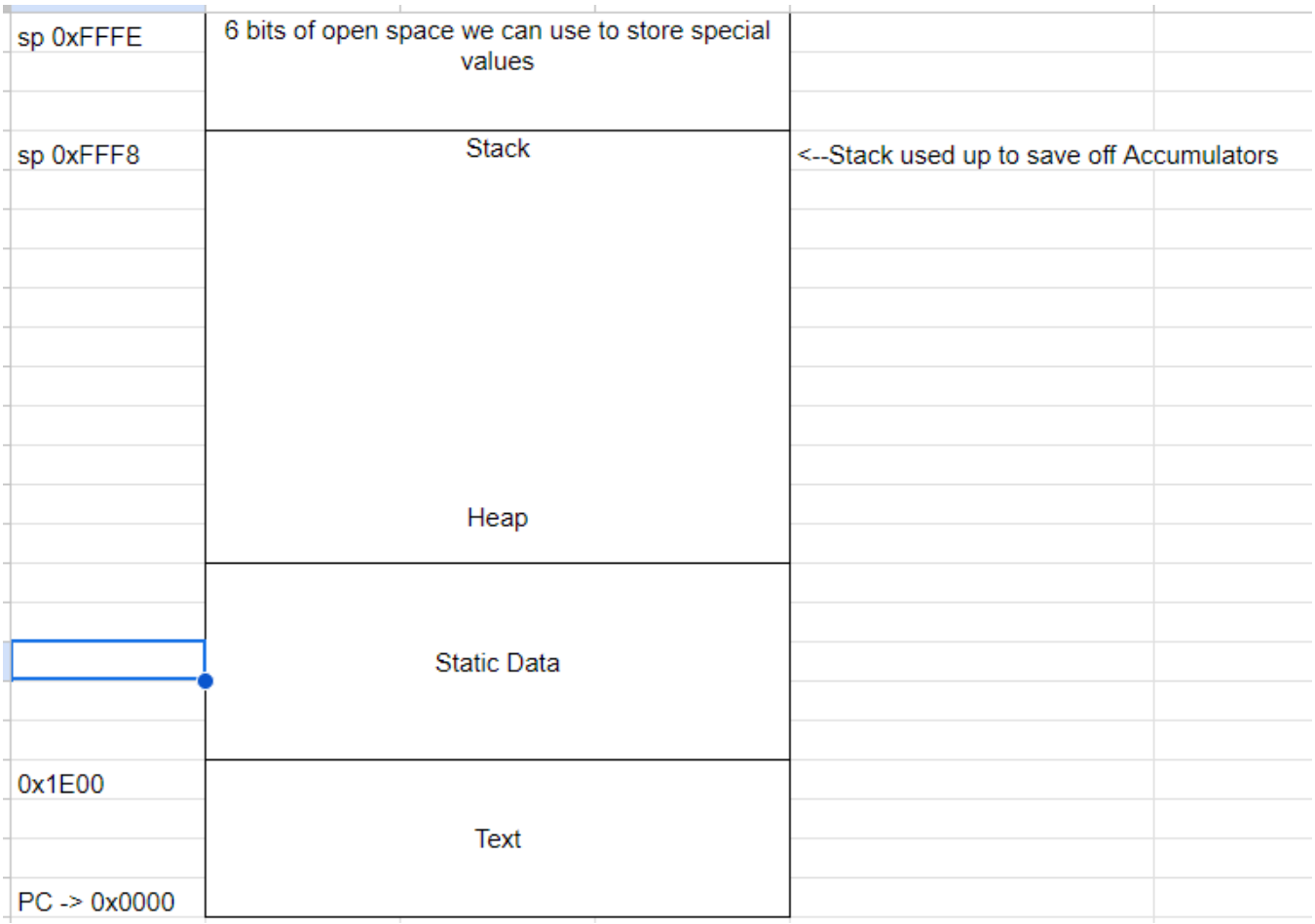
- The C type instruction is just a swap operation and does not use an immediate value or address fields.

- The A-type instructions are dedicated to arithmetic operations, using the Operation field to specify the particular arithmetic function, and utilizing two registers specified by the ID and ID2 fields.

Branching address

We use a 16-bit address space that is byte-addressed. Each line takes up 2 bytes of space. For all address-related instructions with an immediate, including beq, bne, blt, bge, and jal, PC will move relatively from the instruction line by 2 times the immediate (i.e. PC += SE(imm2) << 1). (jb just jumps to the address of the line) Therefore, immediate n means move forward by n lines, and -n means move backward by n lines. For example, immediate -1 means move backward by 1 line and immediate 2 means move forward by 2 lines.

Static data starts at 0x1E00. Sp starts at 0xFFF8. The memory map is below



Static Data Address

Some special addressing to note is that in any R-type instructions that reference static data, the instruction takes in a 9-bit value and the datapath will sudo code 0001 111 at imm[15:9]. For Lui instructions, an 11-bit value is given and the datapath will sudo code 0001 1 at [15:11].
Any variables declared using the .word declaration, will be saved into static memory. Static memory begins at 0x1E00 and ends at 0x2000. This memory is not guaranteed and can be overwritten without warning. To prevent this, users must be agile and store data that cannot be lost in the stack. This is done by opening up more than the approved amount of space on the stack and storing the data above the return address.

ALU

ALUOp	Name	Case	Operation
0000	add	0	"+"
0001	sub	1	-
0010	shift left logical	2	<<<
0011	shift right logical	3	>>>
0100	shift right arithmetic	4	>>
0101	or	5	|
0110	and	6	&
0111	xor	7	⊕

1000	equal	8	"=="
1001	not equal	9	!=
1010	less than	10	<
1011	greater than or equal to	11	>=
01100	load 0	12	L0
1101	load 1	13	L1
1110	swap	14	add0
1111	addressing addition	15	add8+

We need words here

Only the weak can’t understand the sheet

Alter:

Alter is an instruction that takes 3 registers and an operation that acts like another opcode. The ALU will use this separate opcode as a signal to specify the operation. So the ALU can get its arithmetic operation from the alter opcode or the general opcode (decided by the mux). An example of the instruction is “alter r2 r2 r1 +”. This sets reggie2 to be reggi2 + reggie1. R[2] = R[2] + R[1] The codes and symbols are as shown:

Operation	Code	Symbol	Note
add	00000	+	
sub	00001	-	
srl	00110	>>>	
sra	00111	>>	
load 0	01101	L0	we do not care about the args
load 1	01110	L1	we do not care about the args

Code -> Assembly & Machine Instructions For:

RelPrime:

```
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;
}

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
0x0000	input 0	0000000000011010	
0x0002	RP: loadsp r0 -8	1111110000011000	
0x0004	set r1 2	0000000100100110	
0x006	set r2 1	0000000011000110	

0x0008	SU: movesp -4	1111111110011001	
0x000A	storesp r0 0	0000000000010111	
0x000C	storesp r1 2	0000000100110111	
0x000E	jal GCD	0000000100010100	
0x0010	movesp 8	0000000100011001	
0x0012	beq r2 r3 finish	0000011111010000	
0x0014	addi r1 1	0000000010101011	
0x0016	beq r1 r1 SU	1111001010110000	
0x0018	finish: swap r1 r3	0000000110111101	
0x001A	output 0	0000000000011111	
0x001C	stop 0	0000000000000111	
0x001E	GCD: loadsp r0 0	0000000000011000	
0x0020	loadsp r1 2	0000000100111000	
0x0022	set r2 0	0000000001000110	
0x0024	bne r0 r2 continue	0000011100010001	
0x0026	swap r1 r3	0000000110111101	
0x0028	jb 0	0000000000011100	
0x002A	continue: beq r1 r2 done	0000110100110000	
0x002C	blt r1 r0 ChangeA	0000011000110010	
0x002E	alter r1 r1 r0 -	0000101000111110	
0x0030	beq r1 r1 continue	1111101010110000	
0x0032	ChangeA:	0000100010011110	

	alter r0 r0 r1 -		
0x0034	beq r0 r0 continue	1111011000010000	
0x0036	done: swap r0 r3	0000000110011101	
0x0038	jb 0	0000000000011100	

Recursion:

```
def fibonacci_recursive(n):
    if n <= 1:
        return n
    else:
        return fibonacci_recursive(n-1) + fibonacci_recursive(n-2)
```

Address	**Assembly**	**Machine Code**	**Comments**
0x0000	fib: loadsp r0 0x02		R[0] = n
0x0002	alter r2 r0 r0 L1		R[2] = 1
0x0004	bge r2 r0 ELSE		R[2] >= [0] ? 1 >= n
0x0006	swap r3 r0		R[3] = n R[0] = ?
0x0008	jb 0		Jumping back
0x000A	ELSE: addi r0 -1		n = n -1 R[0] = R[0] - 1
0x000C	opensp 4		Set up stack for fib call
0x000E	storesp r0 0x2		int n = R[0] (n-1)
0x0010	jal fib		Call fib

0x0000	closesp 4		Restore stack from fib call
0x0002	addi r0 -1		n = n - 1 Really is n = n - 2 since we have already subtracted 1
0x0004	opensp 4		Set up stack for fib call
0x0006	storesp r0 0x2		int n = R[0] (n-2)
0x0008	swap r3 r1		R[1] = R[3] Whatever fib(n-1) returned is in R[1]
0x000A	jal fib		Call fib
0x000C	closesp 4		Restore stack from fib call R[3] = fib(n -2)
0x000E	alter r3 r3 r1 +		R[3] = R[3] + R[1] return = fib(n-1) + fib(n -2)
0x0010	jb 0		End

While Loop:

```
public static int countEvens() {
int num = 1;
int count = 0;
        while (num <= 20) {
                if (num % 2 == 0) {
                        count++;
                }
                num++;
        }
return count;
}
```

Address	Assembly	Machine Code	Comments
0x0000	addi 0 1	01011 00 000000001	# num, assuming registers start at 0
0x0002	addi 3 0	01011 01 000000000	# count, assuming registers start at 0
0x0004	addi 2 20	01011 10 000010100	# immediate, assuming registers start at 0
0x0006	addi 3 1	01011 11 000000001	# store 1 into register 3, assuming registers start at 0
0x0008	store 0 0x0004	10111 00 000000100	# store num in memory
0x000A	store 1 0x0008	10111 11 000001000	# store 1 into memory
0x000C	WHILE: blt 0x0004 END	00100 010110 10010	# check if 20 is less than num
0x000E	CHECK: srli 0 0x0008	01101 00 000001000	# divide num by 2
0x0010	addi 2 -20	01011 10 111101100	# make register 2 equal to 0
0x0012	store 0 0x000C	10111 00 000001100	# store num after being divided into memory
0x0014	bne 0x000C NEXT	01100 000110 10001	# check is num is not 0
0x0016	load 0 0x0004	10110 00 000000100	# load num before division into register 0
0x0018	addi 3 1	01011 01 000000001	# add one to count
0x001A	NEXT: addi 0 1	01011 00 000000001	# add 1 to num
0x001C	store 0 0x0004	10111 00 000000100	# store new num after addition into memory
0x0001E	addi 2 20	01011 10 000010100	# put 20 back into register 2

0x0020	bge 0x0004 WHILE	00100 101100 10011	# go back to the while loop if 20 is greater than equal to num
0x0022	END: beq 0x0004 CHECK	00100 101100 10000	# check to make sure 20 and num aren't equal because if they are we aren't done

Conditionals:

```
public static int max(int a, int b) {
   if(a >= b)
      return a;
   Else
      return b;
}
```

Address	**Assembly**	**Machine Code**	**Comments**
0x0000	loadsp 1 -4	1111100 01 11001	# load b into register 1
0x0002	loadsp 2 -8	1111000 10 11001	# load a into register 2
0x0004	store 1 0x0004	10111 01 000000100	# store data in register 1 into memory
0x0006	bge 0x0004 EXIT	10011 00100	# check is a is greater than or equal to b
0x0008	load 3 0x0004	10110 11 000000100	# load b into register 3 which is used for return values
0x000A	blt 0x0004 DONE	10010 00100	# branch to done which is below all other instructions
0x000C	EXIT: store 2 0x0008	10111 10 000001000	# store a into memory
0x000E	load 3 0x0008	10110 11 000001000	# load a into register 3 which is used for return

			values

Basic Procedure Call:

```
foo(){
        int a = 3;
        int b = 4;
        int c = findSum(a, b) ;
        return c
}
```

Address	**Assembly**	**Machine Code**	**Comments**
0x0000	A .word 3		Sets A to set
0x0002	B .word 4		Sets B to 4
0x0004	Load r0 A		
0x0006	Load r1 B		
0x0008	Opensp 8		Opens stack for proc call
0x000A	Storesp r0 0x4		Stores r0 on stack
0x000C	Storesp r1 0x8		Stores r1 on stack
0x000E	Jal findSum		Opens stack for proc call

Multi Cycle Register Transfer Language Specification

Naming Conventions

- Modules: module_name
- Test Benches: tb_<module name>
- Inputs: Camel Case
- Outputs: All caps
- Wires: DOGtoCAT
- Registers: r<ID> (r1 for example)
- Clocks: _CLK
- Other: Kebab Case (Dog-Cat)

Multi-Cycle Components

Component	Inputs	Input Bit	Outputs	Output Bit	Behavior	RTL Symbols
Program Counter		16		16	*Keeps track of the address of the LOC we are running*	PC
Register File	ID	2	Reg[ID]	16	*Stores information in the register for easy access*	Reg Reg[0] Reg[1] Reg[2] Reg[3]
Registers	Data Starting address	16 16	Content in the Register	16	*These are the registers in the datapath.*	A B ALUOut MDR SP Output
ALU	ALUSrcA ALUSrcB, ALUOp	16 16 3	ALUSrcA op ALUSrcB shouldBranch	16 1	*Operation determined from ALU Control. Once ALUOp received, you will perform the operation on SrcA and SrcB*	+, -, <<<, >>>, >>, |, &, ⊕ , ==, !=, <, >=, L0, L1, add0, add6+

ALU Control	ALUOp Opcode Operation	2 5 5	Operation for ALU	4	*Either 00 for add, 01 for subtract, or a 10 for looking at instruction opcode, and 11 for alter opcode*	
Memory	Address Data	16	Instruction	16	*Pulls the instruction located at PC & returns it*	Mem[addr]
Immediate Generator	Instruction[15-0]	16	Instruction	16	*Takes in a 16-bit instruction and decodes it into a 16 bit immediate value. For R types it Address extends(hard codes 7 MSB to 0001 111.) and Sign-extends everything else*	SE() AE()
Mux				1 to 16	Uses a selector bit to go to another operation	
Control Unit	Instruction[4-0]	5	Branch MemRead MemtoReg ALUop MemWrite RegWrite	Depends on ALU	Takes in the opcode and interprets it. After that, the control unit will generate control signals.	
Instruction Register	Address	16	Address	16		inst
InputIO	n/a	n/a	Data input from user	16		inputio
OutputIO	Data output to user	16	n/a	n/a		outputio
...						

Multi-Cycle RTL Instructions

Name / Type	Instruction	Multi-Cycle RTL	Comment
CYCLE 1		**Cycle 1:** PC = PC + 2 Inst ← Mem[PC]	Fetch:
CYCLE 2		**Cycle 2:** A ← Reg[inst[6:5]] B ← Reg[inst[8:7]] MDR ← Mem[AE(inst[15:7])] ALUOut ← PC + SE(inst[15:9] << 1)	Decode:
Add / R	add	**Cycle 3:** ALUOut ← A + MDR **Cycle 4:** Reg[inst[6:5]] ← ALUOut	Perform Operation Return
Subtract / R	sub	**Cycle 3:** ALUOut ← A - MDR **Cycle 4:** Reg[inst[6:5]] ← ALUOut	Follows the same format as the previous R type instruction.
XOR / R	xor	**Cycle 3:** A ← A ⊕ MDR **Cycle 4:** Reg[inst[6:5]] ← ALUOut	Follows the same format as the previous R type instruction.
OR / R	or	**Cycle 3:** ALUOut ← A | MDR **Cycle 4:** Reg[inst[6:5]] ← ALUOut	Follows the same format as the previous R-type instruction.
AND / R	and	**Cycle 3:** ALUOut ← A & MDR **Cycle 4:** Reg[inst[6:5]] ← ALUOut	Follows the same format as the previous R type instruction.
Shift Left Logical / R	sll	**Cycle 3:** ALUOut ← A <<< MDR	Follows the same format as

Name / Type	Instruction	Multi-Cycle RTL	Comment
		Cycle 4: Reg[inst[6:5]] ← ALUOut	the previous R type instruction.
Set Less Than / R	slt	**Cycle 3:** ALUOut ← if(A < MDR)? 1 : 0 **Cycle 4:** Reg[inst[6:5]] ← ALUOut	In general similar to how the B-types are written out The ALU will subtract A and MDR and store it in an output "isNegative" The output will be the first bit in the operation A - MDR
Load / R	load	**Cycle 3:** Reg[inst[6:5]] ← MDR	Loads what's in MDR and puts into r1
Store / R	store	**Cycle 3:** Mem[AE(inst[15:7])] ← A	Stores what's in A and puts into MDR
Set / I	srl	**Cycle 3:** Reg[inst[6:5]] ← SE(inst[15:5])	Sets a register to a immediate
LoadSp / I	loadsp	**Cycle 3:** ALUOut ← SE(inst[15:5]) add8+ SP **Cycle 4:** MDR ← Mem[ALUOut] **Cycle 5:** Reg[inst[6:5]] ← MDR	Imm gen will have a signal to know to SE [15:7] Go into Regfile and write data into r1
StoreSp / I	storesp	**Cycle 3:** ALUOut ← SE(inst[15:5]) + SP **Cycle 4:** Mem[ALUOut] ← A	Calculate where in SP we are referencing

Name / Type	Instruction	Multi-Cycle RTL	Comment
			Write data into memory at the address SP
Or Immediate / I	ori	**Cycle 3:** ALUOut ← A | SE(inst[15:5]) **Cycle 4:** Reg[inst[6:5]] ← ALUOut	Perform Operation Return
Xor Immediate / I	xori	**Cycle 3:** ALUOut ← A ⊕ SE(inst[15:5]) **Cycle 4:** Reg[inst[6:5]] ← ALUOut	Follows the same format as the previous I type instruction.
And Immediate / I	andi	**Cycle 3:** ALUOut ← A & SE(inst[15:5]) **Cycle 4:** Reg[inst[6:5]] ← ALUOut	Follows the same format as the previous I type instruction.
Add Immediate / I	addi	**Cycle 3:** ALUOut ← A + SE(inst[15:5]) **Cycle 4:** Reg[inst[6:5]] ← ALUOut	Follows the same format as the previous I type instruction.
Shift Left Logical Imm / I	slli	**Cycle 3:** ALUOut ← A <<< SE(inst[15:5]) **Cycle 4:** Reg[inst[6:5]] ← ALUOut	Follows the same format as the previous I type instruction.
Shift Right Logical Imm / I	srli	**Cycle 3:** ALUOut ← A >>> SE(inst[15:5]) **Cycle 4:** Reg[inst[6:5]] ← ALUOut	Follows the same format as the previous I type instruction.
Shift Right Arith Imm / I	srai	**Cycle 3:** ALUOut ← A >> SE(inst[15:5]) **Cycle 4:**	

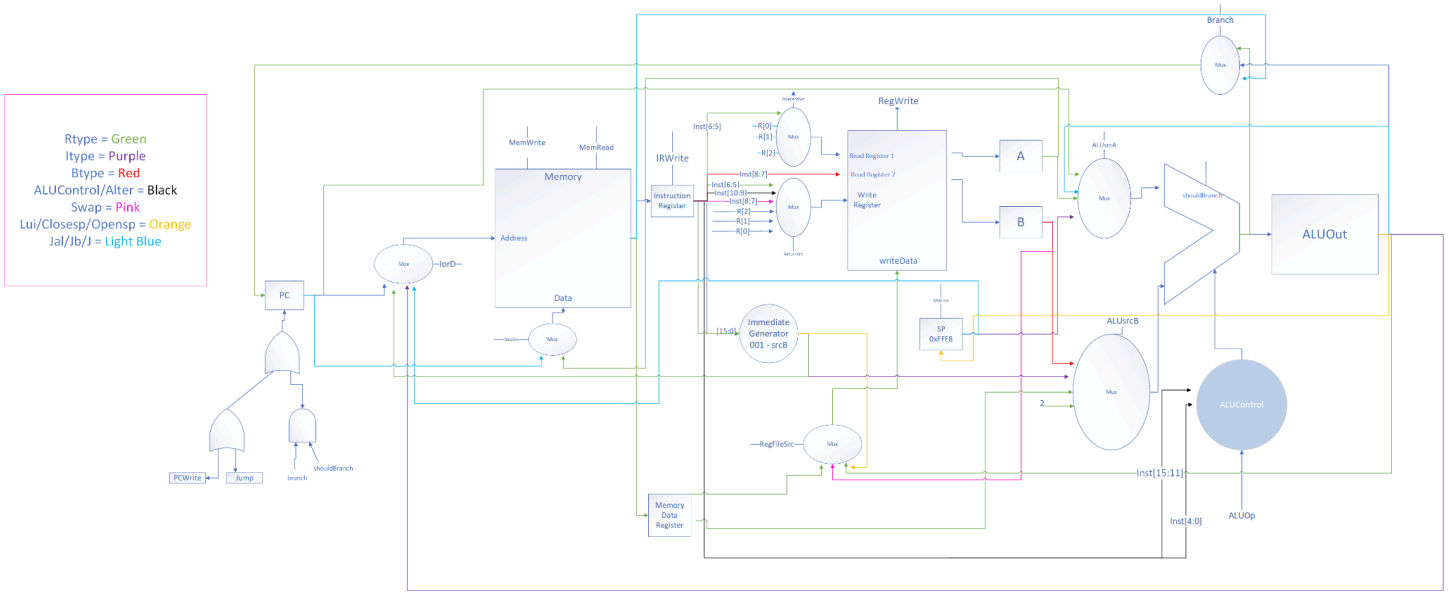
Name / Type	Instruction	Multi-Cycle RTL	Comment
		[inst[6:5]] ←ALUOut	
Branch == / B	beq	**Cycle 3:** if (A == B) PC ← ALUOut	SE(inst[15:9]) represents the # of lines to jump. Multiply by 2 since addresses are 2 bits
Branch != / B	bne	**Cycle 3:** if (A != B) PC ← ALUOut	The ALU will subtract A and B and store either a 1 or 0 in the output “isZero”.
Branch < / B	blt	**Cycle 3:** if (A < B) PC ← ALUOut	The ALU will subtract A and B and store the first bit of the operation in the output “isNegative”.
Branch >= / B	bge	**Cycle 3:** if (A >= B) PC ← ALUOut	The ALU will subtract A and B and store the first bit of the operation in the output “isNegative”.
Input / J	input	**Cycle 3:** Input 0x0	
Output / J	output		
Stop / J	stop		
Jump And Link / J	jal	**Cycle 3:** ALUOut ← SP - 2 **Cycle 4:** Mem[ALUOut] ← PC ALUOut ← ALUOut - 2	

Name / Type	Instruction	Multi-Cycle RTL	Comment
		A ← Reg[2] **Cycle 5:** Mem[ALUOut] ← A ALUOut ← ALUOut - 2 A ← Reg[1] **Cycle 6:** Mem[ALUOut] ← A ALUOut ← ALUOut - 2 A ← Reg[0] **Cycle 7:** Mem[ALUOut] ← A SP ← ALUOut PC = PC + SE(inst[15:5] << 1)	
Jump Back / J	jb	**Cycle 3:** MDR ← Mem[SP] ALUOut= SP + 2 **Cycle 4:** Reg[0] ← MDR MDR ←Mem[ALUOut] ALUOut ← ALUOut + 2 **Cycle 5:** Reg[1] ← MDR MDR ←Mem[ALUOut] ALUOut ← ALUOut + 2 **Cycle 6:** Reg[2] ← MDR MDR ←Mem[ALUOut] ALUOut ← ALUOut + 2 **Cycle 7:** PC ← MDR SP ← ALUOut	
MoveSp/ J	movesp	**Cycle 3:** ALUOut ← SP + SE(inst[15:5])	Move sp

Name / Type	Instruction	Multi-Cycle RTL	Comment
		Cycle 4: SP ← ALUOut	Store the value back in memory
Lui / J	lui	**Cycle 3:** Reg[2] ← SE(inst[15:5] <<< 5)	
Swap / C	swap	**Cycle 3:** ALUOut ← A Reg[inst[6:5]] ← B **Cycle 4:** Reg[inst[8:7]] ← ALUOut	RN is hard-coded to somewhere in top of memory
Alter Register / A	alter	**Cycle 3:** ALUOut ← A op B **Cycle 4:** Reg[inst[10:9]] ← ALUOut	op ← inst[15:11] will go into the ALU as the ALUOp Do the operation with A and B Reg[inst[10:9]] = rd
Input	input	**Cycle 3:** ALUOut ← SP + SE(inst[15:5]) **Cycle 4:** Mem[ALUOut] ← inputio	
Output	output	**Cycle 3:** A ← Reg[3] **Cycle 4:** Output ← A	

Cycle Decision for Processor

We have decided to implement a multi-cycle processor for out project. The data path is below



Our Signals include:

Signal	Desc
PCWrite	Controls when we write to PC 0 → Does not write onto PC 1 → Writes onto PC
Jump	Tells us if we are in a jump instruction 0 → Not a jump instruction 1 → Jump instruction
Branch	Tells us if we are in a branch instruction 00 → PC += 2 (we are not branching) 01 → PC = Jump/Branch to address 10 → PC = Jump back address
shouldBranch	Tells us if we should branch

	0 → Does not Branch 1 → Branches
IorD	Tells us what spot in memory we will be reading from 00 → PC Addr 01 → Immediate Generator 10 → ALUOut 11 → SP
MemWrite	Controls when we can write to memory 0 → Does not write to Mem 1 → Writes to Mem
MemRead	Controls when we can read from memory 0 → Does not read to Mem 1 → Reads to Mem
RegFileSrc	Controls what data gets sent to the regfile 00 → Reg file data = MDR 01 → Reg file data = B 10 → Reg file data = ALUOut 11 → Reg file data = immGen
IRWrite	Controls when we can write to the instruction register 0 → Does not write to IR 1 → Writes to IR
RegWrite	Controls when we can write onto the reg file 0 → Does not write to Reg 1 → Writes to Reg
ALUOp	Controls the ALU Control, allowing it to know what operation to send to the ALU 00 → Add 01 → Subtract 10 → Follow Instruction opcode 11 → Alter Opcode
ALUSrcA	Acts as input 1 to the ALU 00 → PC 01 → ALUOut

	10 → A 11 → SP
ALUSrcB	Acts as input 2 to the ALU 00 → B 01 → ImmGen 10 → MDR 11 → 2
Return SRC	Controls what register will get written onto for certain instructions 00 → Inst[6:5] 01 → Inst[8:7] 10 → Inst[10:9]
DataSrc	Controls what data gets read into memory 00 → A 01 → PC 10 → Inputio
OperandSrc	Controls what data gets read into A 00 - inst[6:5] 01 - r1 10 - r2 11 - r3

Testing Plan (Integration & Unit)

Components:	Test:
Memory	Load In Add inst
IorD Mux	Load In Subtract inst
Inst. Register	Load In XOR inst
MDR	Load In OR inst
Output Mux	Load In AND inst
PC	Load In Shift Left Logical inst
PCWrite/Jump Or	Load In Shift Right Logical inst
Branch/ShouldBranch And	Load In Shift Right Arithmetic inst
Branch Mux	Load In Or Immediate inst
	Load In Xor Immediate inst
	Load In And Immediate inst
	Load In Add Immediate inst
	Load In Shift Left Logical Imm inst
	Load In Shift Right Logical Imm inst
	Load In Shift Right Arith Imm inst
	Load In Set Less Than inst
	Load In Branch == inst
	Load In Branch != inst
	Load In Branch < inst
	Load In Branch >= inst
	Load In Jump And Link inst
	Load In Load inst
	Load In Store inst
	Load In StoreSp inst
	Load In LoadSp inst
	Load In OpenSP inst
	Load In CloseSP inst
	Load In Lui inst
	Load In Jump Back inst
	Load In Swap inst
	Load In Alter Registers inst

Regfile	Load in X for R1 confirm its in A
WriteRegMux	Load in X for R2 confirm its in B
ReadRegMux	Load 0 Selector Bit in RegFileSrc Mux
A	Load 1 Selector Bit in RegFileSrc Mux
B	Load 0 Selector Bit in IorD Mux
SP	Load 1 Selector Bit in IorD Mux
RegFileSrc Mux	Load 2 Selector Bit in IorD Mux
ImmGen	Load 0 Selector Bit in WriteReg Mux
	Load 1 Selector Bit in WriteReg Mux
	Load 2 Selector Bit in WriteReg Mux
	Load 3 Selector Bit in WriteReg Mux
	Load 0 Selector Bit in ReadReg Mux
	Load 1 Selector Bit in ReadReg Mux
	Load 2 Selector Bit in ReadReg Mux
	Load 3 Selector Bit in ReadReg Mux
ALUSrcA Mux	Perform add
ALUSrcB Mux	Perform sub
ALU	Perform XOR
ALU Control	Perform OR
ALUOut	Perform AND
	Perform Shift Left Log
	Perform Shift Right Log
	Perform Shift Right Arth
	Perform Ori
	Perform Xori
	Perform Andi
	Perform Slli
	Perform Srli
	Perform Srai
	Perform Slt
	Perform ==
	Perform !=
	Perform <
	Perform >=

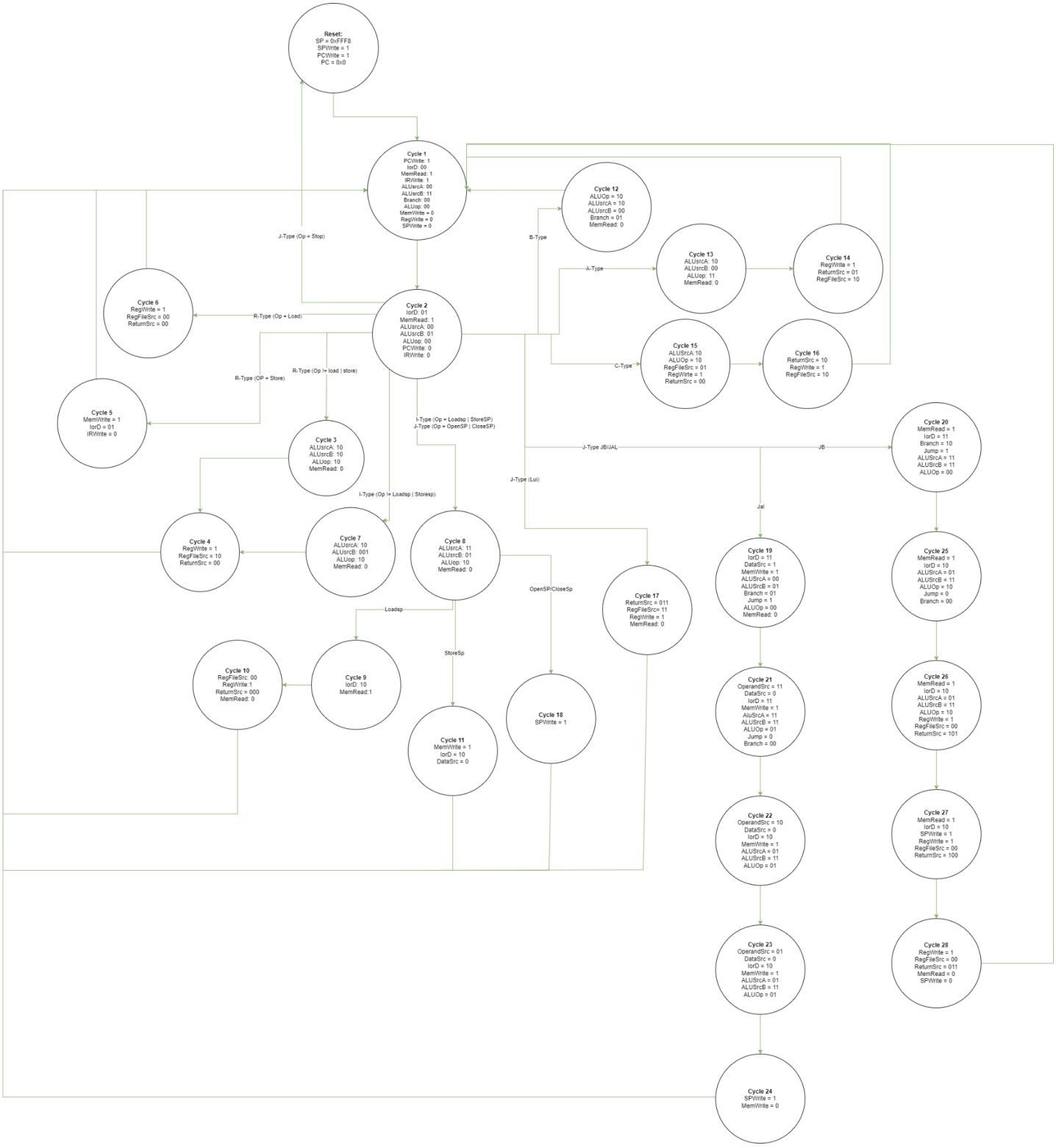
	Perform JAL
	Perform Load
	Perform Store
	Perform StoreSp
	Perform LoadSP
	Perform OpenSP
	Perform CloseSP
	Perform Lui
	Perform JumpBack
	Perform Swap
	Perform Alter
Memory	Load add inst
IorD Mux	Load sub inst
Inst. Register	Load XOR inst
MDR	Load OR inst
Output Mux	Load AND inst
PC	Load Shift Left Log inst
PCWrite/Jump Or	Load Shift Right Log inst
Branch/ShouldBranch And	Load Shift Right Arth inst
Branch Mux	Load Ori inst
Regfile	Load Xori inst
WriteRegMux	Load Andi inst
ReadReg1Mux	Load Slli inst
A	Load Srli inst
B	Load Srai inst
SP	Load Slt inst
RegFileSrc Mux	Load == inst
	Load != inst
	Load < inst
	Load >= inst
	Load JAL inst
	Load Load inst
	Load Store inst
	Load StoreSp inst
	Load LoadSP inst

	Load OpenSP inst
	Load CloseSP inst
	Load Lui inst
	Load JumpBack inst
	Load Swap inst
	Load Alter inst
Regfile	Perform add
WriteRegMux	Perform sub
ReadReg1Mux	Perform XOR
A	Perform OR
B	Perform AND
SP	Perform Shift Left Log
RegFileSrc Mux	Perform Shift Right Log
ImmGen	Perform Shift Right Arth
ALUSrcA Mux	Perform Ori
ALUSrcB Mux	Perform Xori
ALU	Perform Andi
ALU Control	Perform Slli
ALUOut	Perform Srli
	Perform Srai
	Perform Slt
	Perform ==
	Perform !=
	Perform <
	Perform >=
	Perform JAL
	Perform Load
	Perform Store
	Perform StoreSp
	Perform LoadSP
	Perform OpenSP
	Perform CloseSP
	Perform Lui
	Perform JumpBack
	Perform Swap

	Perform Alter
Memory	Confirm Add produces correct value
IorD Mux	Confirm Subtract produces correct value
Inst. Register	Confirm XOR produces correct value
MDR	Confirm OR produces correct value
Output Mux	Confirm AND produces correct value
PC	Confirm Shift Left Logical produces correct value
PCWrite/Jump Or	Confirm Shift Right Logical produces correct value
Branch/ShouldBranch And	Confirm Shift Right Arithmetic produces correct value
Branch Mux	Confirm Or Immediate produces correct value
Regfile	Confirm Xor Immediate produces correct value
WriteRegMux	Confirm And Immediate produces correct value
ReadReg1Mux	Confirm Add Immediate produces correct value
A	Confirm Shift Left Logical Imm produces correct value
B	Confirm Shift Right Logical Imm produces correct value
SP	Confirm Shift Right Arith Imm produces correct value
RegFileSrc Mux	Confirm Set Less Than produces correct value
ImmGen	Confirm Branch == produces correct value
ALUSrcA Mux	Confirm Branch != produces correct value
ALUSrcB Mux	Confirm Branch < produces correct value
ALU	Confirm Branch >= produces correct value
ALU Control	Confirm Jump And Link produces correct value
ALUOut	Confirm Load produces correct value
	Confirm Store produces correct value
	Confirm StoreSp produces correct value
	Confirm LoadSp produces correct value
	Confirm OpenSP produces correct value
	Confirm CloseSP produces correct value
	Confirm Lui produces correct value
	Confirm Jump Back produces correct value
	Confirm Swap produces correct value
	Confirm Alter Registers produces correct value
	Confirm Rel-Prime produces correct value

Control

Diagram for Main Control



ALU Control Truth Table

Inputs			Output
ALUOut	**Opcode**	**Alter Opcode**	**ALUCtrl**
00	-	-	0000 (+)
01	-	-	0001 (-)
10	00000 (add)	-	0000 (+)
10	00001 (sub)	-	0001 (-)
10	00010 (xor)	-	0111 (⊕)
10	00011 (or)	-	0101 (|)
10	00100 (and)	-	0110 (&)
10	00101 (sll)	-	0010 (<<<)
10	00110 (stop)	-	Won’t happen
10	01000 (ori)	-	0101 (|)
10	01001 (xori)	-	0111 (⊕)
10	01010 (andi)	-	0110 (&)
10	01011 (addi)	-	0000 (+)
10	01100 (slli)	-	0010 (<<<)
10	01101 (srli)	-	0011 (>>>)
10	01110 (srai)	-	0100 (>>)
10	01111 (slt)	-	1011 (<)
10	10000 (beq)	-	1000 (=)
10	10001 (bne)	-	1010 (!=)

10	10010 (blt)	-	1011 (<)
10	10011 (bge)	-	1011 (>=)
10	10111 (storesp)	-	0000 (+)
10	11000 (loadsp)	-	1111 (add8+)
10	11001 (movesp)	-	0000 (+)
10	11101 (swap)	-	1110 (add0)
11	-	00000 (+)	0000 (+)
11	-	00001 (-)	0001 (-)
11	-	01101 (L0)	1100 (L0)
11	-	01110 (L1)	1101 (L1)
11	-	00110 (srl)	0011 (>>>)
11	-	00111 (sra)	0100 (>>)

Testing

Under phase 1 of Integration Plan

Appendix

RTL Components Single-Cycle

Component	Inputs	Input Size (bits)	Outputs	Input Size (bits)	Behavior	RTL Symbols
Program Counter		16		16	*Holds next instruction*	PC
Register	Register ID	2	Data in register	16	*Stores information in the register for easy access*	Reg[ID]
ALU	1. Operand X 2. Operand Y 3. Operator	1 to 16	Result of ALU	16	*Sets PC to some immediate that it will possibly branch too*	+, -, ?==, ?>=, ?<, ?!=
Memory Data	Memory address	16	Data at the address	16	*Pulls the instruction located at PC & returns it*	Mem[addr]
Instruction Data	1. Start digit 2. End digit		Stores instructions into designated spots in memory	1 to 16	*Pulls the individual parts of each instructions & loads them into the designated spot*	inst[X:Y]
Immediate Generator	Immediate	16	Constant value specified in the instruction.	16	*Takes in the bits from our instruction representing the immediate and switches it into an actual immediate number.*	imm
Mux	Will take in as many arguments as needed along with selector bit	1 to 16	PC+=4 or PC += SE(imm)	1 to 16	*Uses a selector bit to go to another operation*	Mux[input1, input2, …](selector)
Control Unit	Control signal	4	Branch MemRead	Selector for all	*Takes in the opcode and interprets it. After that,*	Control

			MemtoReg ALUop MemWrite RegWrite	signal inputs	*the control unit will generate control signals.*	
...						

RTL Instruction Single-Cycle

Name / Type	Instruction	Single-Cycle RTL	Comment
Add / R	add	PC = PC + 2 Inst = Mem[PC] rID = Reg[inst[6:5]] a = Mem[inst[15:7]] result = rX + a rX = result	
Subtract / R	sub	result = rX - a rX = result	Follows the same format as the previous R type instruction.
XOR / R	xor	result = rX ⊕ a rX = result	Follows the same format as the previous R type instruction.
OR / R	or	result = rX | a rX = result	Follows the same format as the previous R type instruction.
AND / R	and	result = rX & a rX = result	Follows the same format as the previous R type instruction.
Shift Left Logical / R	sll	result = rX << a rX = result	Follows the same format as the previous R type instruction.

Name / Type	Instruction	Single-Cycle RTL	Comment
Shift Right Logical / R	srl	result = rX >> a rX = result	Follows the same format as the previous R type instruction.
Shift Right Arithmetic / R	sra	a = Mem[inst[15:7]] result = SE(rX >> a) rX = result	One is zero extend & one is sign extend
Or Immediate / I	ori	PC = PC + 2 Inst = Mem[PC] rX = Reg[inst[6:5]] imm =SE(inst[15:7]) result = rX | imm rX = result	
Xor Immediate / I	xori	result = rX + imm rX = result	Follows the same format as the previous I type instruction.
And Immediate / I	andi	result = rX & imm rX = result	Follows the same format as the previous I type instruction.
Add Immediate / I	addi	result = rX ⊕ imm rX = result	Follows the same format as the previous I type instruction.
Shift Left Logical Imm / I	slli	result = rX << imm rX = result	Follows the same format as the previous I type instruction.
Shift Right Logical Imm / I	srli	result = rX >> imm rX = result	Follows the same format as the previous I type instruction.

Name / Type	Instruction	Single-Cycle RTL	Comment
Shift Right Arith Imm / I	srai	imm = SE(inst[15:7]) result = SE(rX >> imm) rX = result	
Set Less Than / R	slt	PC = PC + 2 Inst = Mem[PC] rX = Reg[inst[6:5]] a = Mem[inst[15:7]] result = rX ?< a rX = result	Not sure about doing in an if in the RTL instruction If rX < a it will output 1 otherwise 0
Branch == / B	beq	PC = PC + 2 Inst = Mem[PC] rX = Reg[inst[6:5]] rX2 =Reg[inst[8:7]] a = SE(inst[15:9]) target = PC + imm result = (rX ?== a) if result PC = target	
Branch != / B	bne	PC = PC + 2 Inst = Mem[PC] rX = Reg[inst[6:5]] rX2 =Reg[inst[8:7]] a = SE(inst[15:9]) target = PC + imm result = (rX ?!= rX2) if result PC = target	
Branch < / B	blt	PC = PC + 2 Inst = Mem[PC] rX = Reg[inst[6:5]] rX2 =Reg[inst[8:7]] a = SE(inst[15:9]) target = PC + imm result = (rX ? < rX2) if result PC = target	
Branch >= / B	bge	PC = PC + 2 Inst = Mem[PC]	

Name / Type	Instruction	Single-Cycle RTL	Comment
		rX = Reg[inst[6:5]] rX2 =Reg[inst[8:7]] a = SE(inst[15:9]) target = PC + imm result = (rX ?>= rX2) if result PC = target	
Jump And Link / J	jal	PC = PC + 2 Inst = Mem[PC] Mem[SP] = PC + 2 HP += 6 Mem[HP] = Reg[0] HP2 = HP -2 Mem[HP2] = Reg[1] HP3 = HP - 4 Mem[HP3] = Reg[2] PC = SE(inst[15:5]) << 1	
Load / R	load	PC = PC + 2 inst = Mem[PC] rX = Reg[inst[6:5]] rX = Mem[inst[15:7]]	
Store / R	store	PC = PC + 2 inst = Mem[PC] Mem[inst[15:7]] = rX	
StoreSp / R	storesp	PC = PC + 2 inst = Mem[PC] rX = Reg[inst[6:5]] offset = SE(inst[15:7]) address = SP + offset Mem[address] = rX	
LoadSp / R	loadsp	PC = PC + 2 inst = Mem[PC] rX = Reg[inst[6:5]] offset = SE(inst[15:7]) address = SP + offset rX = Mem[address]	
OpenSP / S	opensp	PC = PC + 2 inst = Mem[PC] SP += SE(inst[15:5])	

Name / Type	Instruction	Single-Cycle RTL	Comment
CloseSP / S	closesp	PC = PC + 2 inst = Mem[PC] SP -= SE(inst[15:5])	
Lui / S	lui	PC = PC + 2 inst = Mem[PC] rX = Reg[2] result = inst[15:5] << 5 rX = result	
Jump Back / J	jb	PC = PC + 2 Inst = Mem[PC] Reg[0] = Mem[HP] HP2 = HP - 2 Reg[1] = Mem[HP2] HP3 = HP - 4 Reg[2] = Mem[HP3] HP -= 6 PC = Mem[SP]	
Swap / C	swap	PC = PC + 2 Inst = Mem[PC] rX = Reg[inst[6:5]] rY = Reg[inst[8:7]] rX = rY rY = rX	
Alter Register / A	alter	PC = PC + 2 Inst = Mem[PC] rX = Reg[inst[6:5]] rX2 = Reg[inst[8:7]] rd = Reg[inst[10:9]] op = imm(inst[11:15]) result = rX op rX2 Reg[rd] = result	

Checking for Errors:

To ensure our RTL instructions will work as intended, we have followed the same format as the instructions given to us in class and consulted with our Professor. We have also worked out our instructions to make sure they produced the expected results.