Team SID-Q

by Amity Stevens, Ben McDaniel, Tyler Moore and V Brunkow

Last Updated: 5/23/23

Table of Contents

Description of Design..... 2
Measuring Performance..... 2
SID++ Reference Sheet..... 3
SID++ Registers..... 4
SID++ Syntax Examples..... 5
SID++ Common Operations..... 6
SID++ Example Program..... 9
SID++ Naming Conventions..... 12
SID++ Datapath Diagram..... 13
SID++ Datapath Integration Plan..... 14
SID++ Datapath Component Specifications..... 17
SID++ Implementation Unit Tests..... 21
SID++ Instruction Set RTL..... 23
SID++ RTL Verification..... 24
SID++ Control Finite State Machine..... 31
SID++ Control Signal Specifications..... 32
SID++ Control Signal Tests..... 34
SID++ System Verification..... 37
SID++ Data..... 40

Description of Design

SID++ follows a simple design philosophy, attempting to keep as much similarity to RISC-V as possible with minor syntactic improvements. From a functional standpoint, SID++ functions exactly as a 16-bit word RISC-V would, aside from the addition of the TEST function, which similar to in other ISAs like ArmV7 compares 2 values and stores a comparison result in the assembler's register. The main benefit of this, is that it allows a much larger branch range, needing only a 4-bit opcode and leaving the rest to an immediate value. Syntactically, we have removed bloat and opted to allow the use of "SID" or "sid" anywhere within the program to facilitate readability. One example of this is when referencing registers. For those who appreciate a more strongly typed language, "SID" or "sid" may be inserted as a prefix, suffix, or simply inside any part of any line as long as all of the formatting around it is unchanged, eg. "SIDadd SID6, SID6, SID7". This is possible as our assembler begins by removing every instance of "SID" and "sid" before compiling.

Measuring Performance

The performance of our CPU will be measured using execution time. Insteading of basing it off of the number of instructions in the compilation, we will be basing it off of a standardized program. This program is the example program listed later in the document, and any changes will be compared using this program. We prioritize the speed of the CPU and thus the time is the best way to measure it. To figure this time out, multiply the number of instructions by the average number of clock cycles per instruction, and multiply that number by the time it takes per each clock cycle. For simplicity in our case, we will simply measure the time manually for this to occur. In addition, we will measure the performance based on how well our branch works. We do a different approach to how we branch by using a test instruction, so we value how well the CPU accomplishes the branch. Since it is a way to offer significantly longer branches than one should be able to do with 16 bits, we wanted to make sure we emphasized the significance of the functionality.

SID++ Reference Sheet

INSTRUCTION	NAME	FMT	OP	DESCRIPTION
SIDadd	ADD	R	0000	R[rd] = R[rs0] + R[rs1]
SIDsub	SUB	R	0001	R[rd] = R[rs0] - R[rs1]
SIDxor	XOR	R	0010	R[rd] = R[rs0] ^ R[rs1]
SIDor	OR	R	0011	R[rd] = R[rs0] | R[rs1]
SIDand	AND	R	0100	R[rd] = R[rs0] & R[rs1]
SIDslli	shift left logical imm	I	1110	R[rd] = R[rd] << SE[imm[8:0]]
SIDsrli	shift right logical imm	I	1101	R[rd] = R[rd] >> SE[imm[8:0]]
SIDaddi	ADD immediate	I	1100	R[rd] = R[rd] + SE[imm[8:0]]
SIDlui	load upper imm	I	1111	R[rd] = SE[imm] << 8
SIDlw	load word	S	0110	R[rd] = M[R[rs0] + imm[4:0]]
SIDsw	store word	S	0111	M[R[rs0] + imm[3:0] >> 1] = R[rd]
SIDbeq	branch equal	B	1000	if(R15 = 0) PC = PC + SE[imm<<1]
SIDbge	branch greater eq	B	1001	if(R15 >= 0) PC = PC + SE[imm<<1]
SIDblt	branch less than	B	1010	if(R15 < 0) PC = PC + SE[imm<<1]

SIDtst	test branch	RC	0101	if(R[rs0] = R[rs1]) R[15] = 0
				if(R[rs0] > R[rs1]) R[15] = 1
				if(R[rs0] < R[rs1]) R[15] = 2

SIDjalr	jump and link register	S	1011	R[rd] = PC + 2 PC = R[rs0] + SE[imm]

I Type:

OP [4 bits]	rd [4 bits]	immediate [8 bits]

R and RC Types:

OP [4 bits]	rd [4 bits]	rs0 [4 bits]	rs1 [4 bits]

B Type:

OP [4 bits]	immediate [12 bits]

S Type:

OP [4 bits]	rd [4 bits]	rs0 [4 bits]	immediate [4 bits]

SID++ Registers

Register	Register Name	Register Function	Saver
0	zero	zero constant	N/A
1	ra	return address	Caller
2	sp	stack pointer	Callee
3	t0	temporary reg	Caller
4	t1	temporary reg	Caller
5	t2	temporary reg	Caller
6	t3	temporary reg	Caller
7	t4	temporary reg	Caller
8	t5	temporary reg	Caller
9	t6	temporary reg	Caller
10	arg0, ret0	Fn arg, ret arg	Caller
11	arg1, ret1	Fn arg, ret arg	Caller
12	s0	saved reg	Callee
13	s1	saved reg	Callee
14	s2	saved reg	Callee
15	ass	Assembler temporary	Caller

SID++ Memory

SP -> 0xFFFF	Stack
Out -> 0x6001 In -> 0x6000	Output Input
PC -> 0x0000 - 0x4000	Text

SID++ Syntax Examples

INST	EXAMPLE	MEANING	COMMENTS
add	SIDadd SID3, SID4, SID5	x3 = x4 + x5	three reg operands, add
sub	SIDsub SID3, SID4, SID5	x3 = x4 - x5	three reg operands, sub
xor	SIDxor SID3, SID4, SID5	x3 = x4 ^ x5	three reg operands, xor
or	SIDor SID3, SID4, SID5	x3 = x4 | x5	three reg operands, or
and	SIDand SID3, SID4, SID5	x3 = x4 & x5	three reg operands, and
slli	SIDslli SID3, 4	x3 = x3 << 4	one reg, 8 bit imm, shift left
srli	SIDsrli SID3, 4	x3 = x3 >> 4	one reg, 8 bit imm, shift right
addi	SIDaddi SID3, 90	x3 = x3 + 90	one reg, 8 bit imm, add
lui	SIDlui SID3, 0xAB	x3 = 0xAB00	one reg, 8 bit imm, load upper imm
lw	SIDlw SID3, 4(SID4)	x3 = M[x4 + 4]	two reg, 4 bit imm, load from memory
sw	SIDsw SID3, 4(SID4)	M[x4 + 4] = x3	two reg, 4 bit imm, store to memory
beq	SIDbeq 4	if(x15 = 0) PC += 4	branch if eq after test
bge	SIDbge 4	if(x15>=0) PC+= 4	branch if greater or eq after test
blt	SIDblt 4	if(x15<0) PC+=4	branch if less than after test

tst	SIDtst SID3, SID4	if(x3 = x4) x15 = 0
		if(x3 > x4) x15 = 1
		if(x3 < x4) x15 = -1
			test before branching to do the logic for branching

jalr	SIDjalr SID1, 4(SID3)	x1 = PC + 2 PC = x3 + 2	jump and link from register, storing into ra

SID++ Common Operations

Loops

Address	Machine Code	Assembly	C Code
0x0000	0000010000000000	SIDadd SID4, SID0, SID0	
0x0002	1100010000000100	SIDaddi SID4, 4	int a = 4;
0x0004	0000010100000000	SIDadd SID5, SID0, SID0	
0x0006	1100010100001010	SIDaddi SID5, 10	int b = 10;
		SIDLOOP:	
0x0008	0101111101000101	SIDtst SID4, SID5	
0x000A	1001000000000100	SIDbge SIDEND	while(a <= b) {
0x000C	1100010000000001	SIDaddi SID4, 1	a++;
0x000E	0101111100000000	SIDtst SID0, SID0	}
0x0010	1001111111111100	SIDbeq SIDLOOP	
		SIDEND:	

Loading a Large Number

Address	Machine Code	Assembly	C Code
0x0000	1111010000000010	SIDlui SID4, 2	int a = 577;
0x0002	1100010001000001	SIDaddi SID4, 65	

Calling a Procedure

Address	Machine Code	Assembly	C Code
0x0000	0000101000000000	SIDadd SID10, SID0, SID0	int a = 0;
0x0002	0000101100000000	SIDadd SID11, SID0, SID0	int b = 0;
0x0004	1100101000000101	SIDaddi SID10, 5	int a = 5;
0x0006	1100101100000011	SIDaddi SID11, 3	int b = 3;
0x0008	0000001100000000	SIDadd SID3, SID0, SID0	int c = mult(a, b);
0x000A	1100001101011010	SIDaddi SID3, 0x005A	
0x000C	1011000100110000	SIDjalr SID1, 0(SID3)	

Recursion

Address	Machine Code	Assembly	C Code
0x0000	0000101000000000	add x10, x0, x0	recur(5);
0x0002	1100101000000101	addi x10, 5	
		RECUR:	recur(int a){

0x0004	0000001110100000	add x3, x10, x0	if(a < 0) {
			  return 0;
			}

0x0006	0101111100110000	TST x3, x0	
0x0008	1010000000001111	blt END	

0x000A	1100001011111100	addi sp, -4	else{
			  return a + recur(a - 1);
			}

0x000C	0111001100100000	sw x3, 0(sp)	
0x000E	0111000100100010	sw ra, 2(sp)	
0x0010	1100001111111110	addi x3, -1	
0x0012	0000101000110000	add x10, x3, x0	
0x0014	0000000111100000	add ra, x14, x0	
0x0016	1100000100000110	addi ra, 6	
0x0018	0101111100000000	TST x0, x0	
0x001A	1000111111110110	beq RECUR	

0x001C	0110001100100000	lw x3, 0(sp)	
0x001E	0110000100100010	lw ra, 2(sp)	
0x0020	1100001000000100	addi sp, 4	
0x0022	0000101010100011	add x10, x10, x3	
0x0024	1011000000010000	jalr x0, 0(ra)	
		END:	
0x0026	0000101000000000	add x10, x0, x0	return 0;
0x0028	1011000000010000	jalr x0, 0(ra)	

Reading Input

Address	Machine Code	Assembly	C Code
0x0000	1111001111111111	SIDlui SID3, 0xFF	int a = input;
0x0002	1100001111111110	SIDaddi SID3, 0xFE	
0x0004	0110011100110000	SIDsw SID3, SID3, 0	

Setting Output

Address	Machine Code	Assembly	C Code
0x0000	1111001111111111	SIDlui SID3, 0xFF	output = a;
0x0002	1100001111111111	SIDaddi SID3, 0xFF	
0x0004	0110010000110000	SIDlw SID4, SID3, 0	

SID++ Example Program

Address	Machine Code	Assembly	C Code
0x0000	1111001111111111	lui t0, XC0	relPrime(m);
0x0002	1100001111111111	addi t0, X00	
0x0004	0110101000110000	lw arg0, 0(t0)	
0x0006	0000010100000000	add t2, zero, zero	
0x0008	1100010000001010	addi t2, RelPrime	
0x000A	1011001001010000	jalr ra, 0(t2)	
0x000C	1111001111111111	lui t0, xC0	relPrime(m) -> Output
0x000E	1100001111111110	addi t0, x02	
0x0010	0111101000110000	sw arg0, 0(t0)	
		RelPrime:	
0x0012	0000001100001010	add t0, zero, arg0	int n;
0x0014	0000010000000000	add t1, zero, zero	
0x0016	1100010000000010	addi t1, 2	m = 2;
0x0018	1100001011111110	addi sp, -2	
0x001A	0111000100100000	sw ra, 0(sp)	
0x001C	1100001011111100	addi sp, -4	
		RelPrimeLoop:	gcd(n,m)
0x001E	0111001100100000	sw t0, 0(sp)	
0x0020	0111010000100010	sw t1, 2(sp)	
0x0022	0000101000000011	add arg0, zero, t0	
0x0024	0000101100000100	add arg1, zero, t1	
0x0026	0000010100000000	add t2, zero, zero	
0x0028	1100010101000110	addi t2, GCD	
0x002A	1011000101010000	jalr ra, 0(t2)	

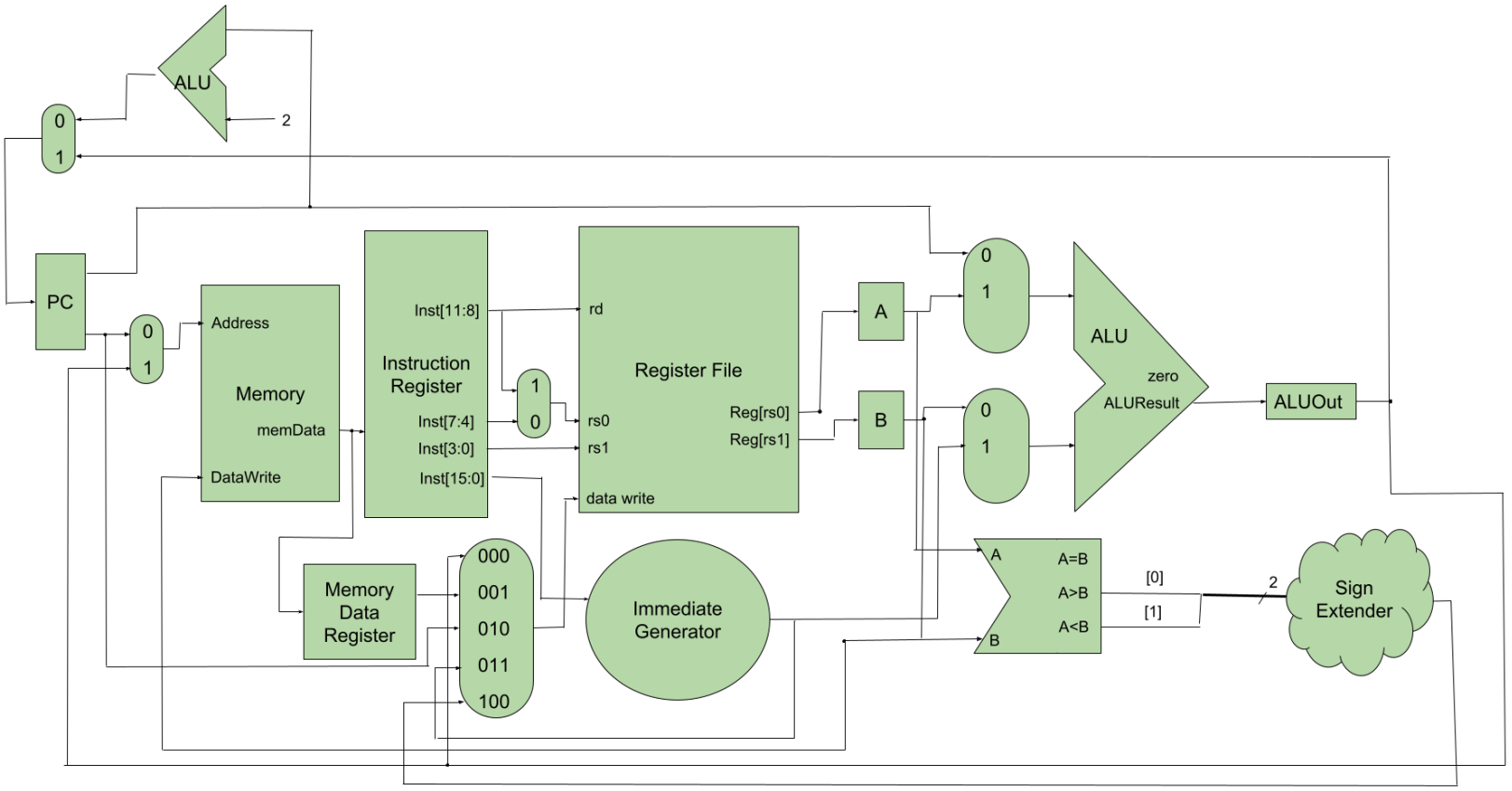
0x002C	0000010100000000	add t2, zero, zero	
0x002E	1100010100000001	addi t2, 1	gcd(n,m) != 1
0x0030	0101111110100101	tst arg0, t2	
0x0032	0110001100100000	lw t0, 0(sp)	
0x0034	0110010000100010	lw t1, 2(sp)	
0x0036	1000000000000100	beq RelPrimeReturn	
0x0038	1100010000000001	addi t1, 1	while(gcd(n,m) != 1) {
0x003A	0101111100000000	tst zero, zero	m = m + 1;
0x003C	1000111111110001	beq RelPrimeLoop	}
		RelPrimeReturn:	return m;
0x003E	0000101000000100	add arg0, zero, t1	
0x0040	1100001000000100	addi sp, 4	
0x0042	0110011000100000	lw t3, 0(sp)	
0x0044	1011000001100000	jalr zero, 0(t3)	
		GCD:	if(a == 0) {
0x0046	0101111110100000	tst arg0, zero	GCDRetB
0x0048	0000000000000000	NOP	
0x004A	1000000000001101	beq GCDRetB	}
0x004C	0000001110100000	add t0, arg0, zero	while (b != 0) {
0x004E	0000010010110000	add t1, arg1, zero	if(a > b) {
		GCDLoop:	a = a - b;
0x0050	0101111101000000	tst t1, zero	}
0x0052	1000000000001011	beq GCDRetA	} else {
0x0054	0101111101000011	tst t1, t0	b = b - a;
0x0056	1010000000000100	blt BLA	}
0x0058	0001010001000011	sub t1, t1, t0	}

0x005A	0101111100000000	tst zero, zero	
0x005C	1000111111111010	beq GCDLoop	
		BLA:	
0x005E	0001001100110100	sub t0, t0, t1	
0x0060	0101111100000000	tst zero, zero	
0x0062	1000111111110111	beq GCDLoop	
		GCDRetB:	return b;
0x0064	0000101010110000	add arg0, arg1, zero	
0x0066	1011000000010000	jalr zero, 0(ra)	
		GCDRetA:	return a;
0x0068	0000101000110000	add arg0, t0, zero	
0x006A	1011000000010000	jalr zero, 0(ra)	

SID++ Naming Conventions

Operation	Naming Convention
op	ALU Operation
<=	Save item to the right into item to the left
WireName[XX:XX]	A wire bus
WireName	A singular wire
+	Add the two values
-	Subtract the two values
<<	Shift left
>>	Shift right
Reg[XX]	Refers to register XX

SID++ Datapath Diagram

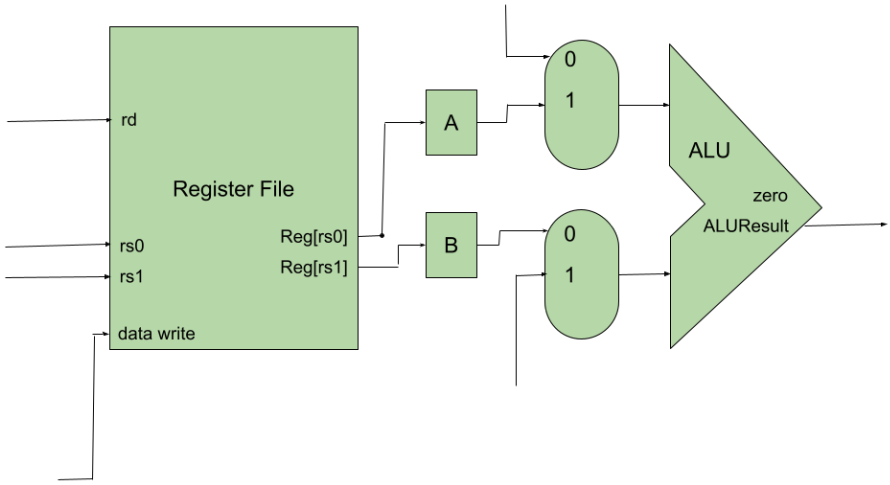


SID++ Datapath Integration Plan

Step 1

Implement ALU, Register File, and Registers A and B

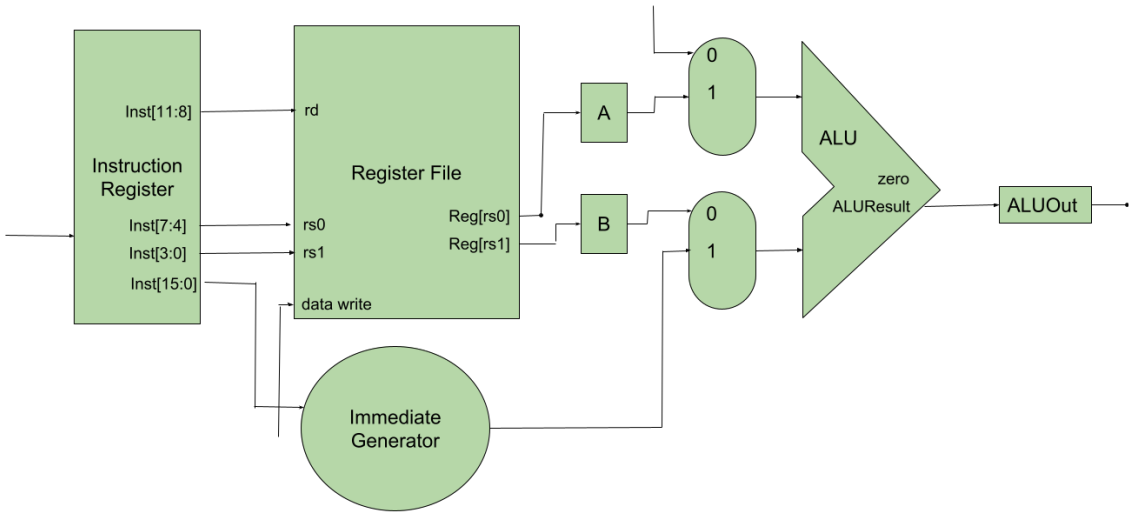
- Combine the components for a start of the project.
- Send a couple operations through to make sure the ALU produces the correct values for just the registers. Must only take 2 cycles.
- Send in an immediate and a PC to make sure both of those cases work correctly and produce the correct value.



Step 2

Implement Instruction Register, Immediate Generator, and Register ALUOut

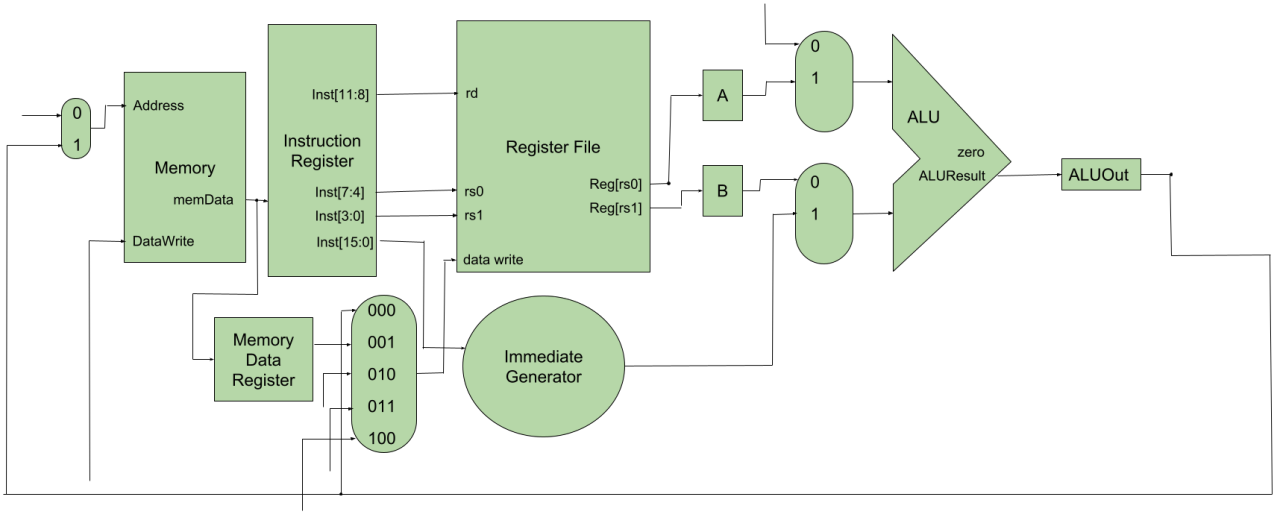
- Add the instruction register to the front of the immediate generator, the immediate generator between the IR and the imm wire, and ALUOut after the ALU.
- Test the same instructions from Step 1, but make sure the immediate is produced this time, and write back works.
- Test number of cycles, it is extremely important in this step.



Step 3

Implement Memory and Register MDR

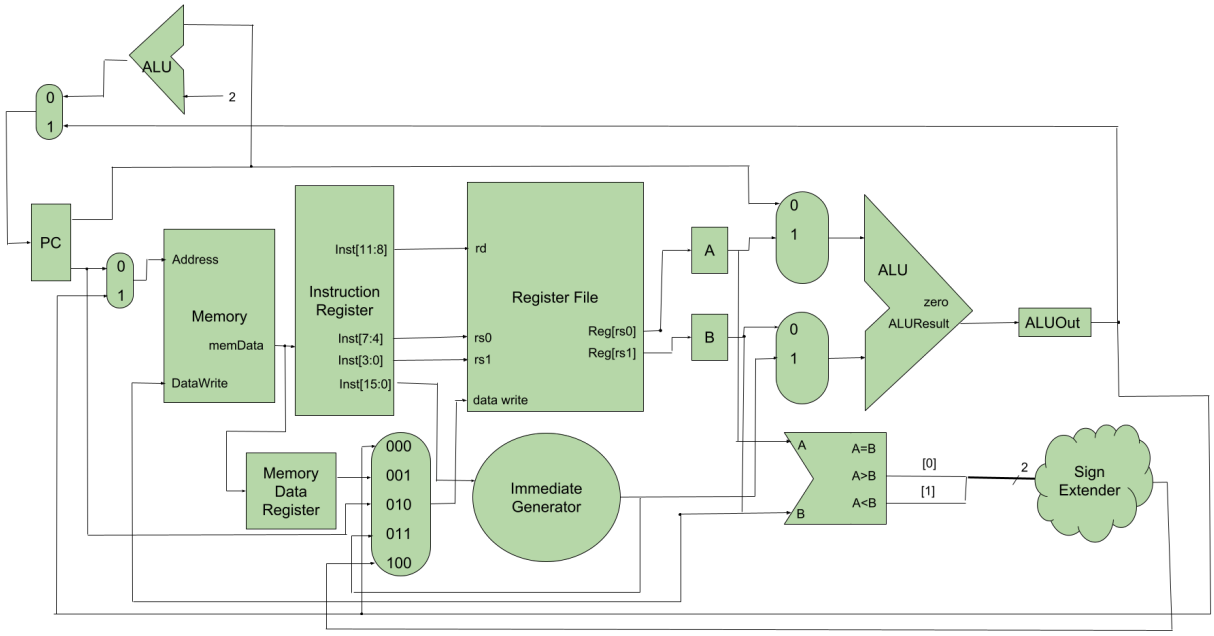
- Add the memory module and MDR before the instruction register.
- Run some tests that read instructions from memory and make sure it has the correct operations as well as timing.
- Run some tests that load and save values from memory, and dive deep into ram to make sure they are processed correctly.
- Test input/output controls



Step 4

Implement PC, PC Adder, Comparator, and the Sign Extender

- Add the PC and its adder to the front of the data path, and the comparator and the sign extender synchronously with the ALU.
- Run tests reading multiple instructions from memory, making sure the PC is changing between them.
- Run a couple test instructions and make sure the compare value is correct. It should then return the two bit value to the test bench, and save the value in the register.
- Check timing on the datapath to ensure it is correct as this is the final step.



SID++ Datapath Component Specifications

Component	Inputs	Outputs	Behavior	RTL Symbols	Control Signals
Register	input_value[15:0], CLK	output_value[15:0]	On the rising edge of the clock (*CLK*) the register reads the *input_value* port and outputs the new value. If the *reset* signal is high on the rising clock edge the register wipes its data and outputs *0x0000* instead.	rd, A, B, ALUOut, PC, IR,MDR	reset: 1 if reset 0 if not
Register File	rd[0:4], rs1[0:4], rs2[0:4], writeData[15:0], CLK	A[15:0], B[15:0]	On the rising edge of the clock *(CLK)* the register file outputs the data stored in the registers designated by either 1111 or [3:0] (depending on which value the mux selects), and [7:4]. If the write signal is enabled, it will take the	Reg	RegWrite: 1 if write 0 if not reset: 1 if reset 0 if not

			data in writeData[15:0], and write it to the register specified by [11:8].		
Memory	address[3:0], writeData[15:0], CLK, inputWire[15:0]	readData[15:0], outputWire[15:0]	On the rising edge of the clock *(CLK)* the memory module will either read or write data from memory. The address[3:0] specifies where the data will be read from or saved to. If the control signal signifies it needs to write, the memory at the address specified will be changed to the writeData[15:0]. If it is set to read, it simply returns the data in the address.	Mem	

Comparator	wireA[15:0], wireB[15:0]	isA>B, isA=B, isA<B	The comparator starts to compare the values and outputs a flag depending on the result. The output values will be zero until their conditions are met, and will return to zero after their condition is no longer valid.	Comp	
ALU	srcA[15:0], srcB[15:0]	ALUOut[15:0]	The ALU takes the two data values in the input and performs a specific operation on them. This operation will be decided by the operation code provided by the control module.	+, -, op	ALUOp: 000 if add 001 if sub 010 if and 011 if or 100 if xor 101 if slli 110 if srli

Immediate Generator	instruction[15:0]	imm[15:0]	The immediate generator will take the instruction and generate an immediate out of it depending on the opcode, and then sign extend it.	SE, <<, >>	numBits: 00 if 0 01 if 4 10 if 8 11 if 12 immShift: 00 if not 01 if 1 10 if 8
1-bit MUX	0[15:0], 1[15:0]	Address[15:0], data write[15:0], srcA[15:0], srcB[15:0], new_PC[15:0]	Selects between 2 input signals corresponding to their input names, and outputs the chosen one.		All output 0[15:0] when control is 0 and 1[15:0] when control is 1 PCSource, memAddrSel, memToReg, ALUSrcA, ALUSrcB,

SID++ Implementation Unit Tests

Register Verification

Steps

- Save a numerical value not 0 to the register
- Turn off writing and make sure the output is not changed
- Test the reset signal
- Test if the value is changed ONLY on the rising clock edge

Register File Verification

Steps

- Load Reg[0] to make sure both signals offer a zero.
- Load another register to ensure both signals offer the same value
- Write another value to this register and make sure it changed
- Repeat to have two registers with different values
- Load both of these registers in one cycle and ensure the outputs correlate
- Change one of the registers again to ensure the values can be changed more than once

Memory Verification

Steps

- Load and address and ensure its output is correct based on what is stored there
- Load the address again to ensure consistency
- Write a different value to the same address, and then read it to make sure it changed
- Ensure nothing happens when read and write control signals are set to zero
- Read a handful of addresses from memory
- Change about half of those values
- Read all of the same addresses again to make sure results are consistent with what is expected
- Put in input address and make sure value is loaded
- Put in output addres and make sure value is sent

Comparator Verification

Steps

- Start testing by using 1 bit to decrease the number of tests needed to start
- Test duplicate inputs of 0 and 1 to make sure the equals flag works
- Test input A = 0 and B = 1 to insure A < B flag is produced
- Test input A = 1 and B = 0 to insure A > B flag is produced
- Write a Verilog script for 32 bits to check each combination of possibilities

ALU Verification

Steps

- Test the operation for two inputs of zero and ensure it produces a zero result
- Test a few known combinations of the operation and make sure the result is what you expected
- Repeat for all operations listed in the control signal specifications

Immediate Generator Verification

Steps

- Pass in machine code with the immediate bits set to 0, ensure the immediate is zeros
- Keeping the rest of the machine code the same, change the immediate bits a few times to make sure it produces the correct result
- Pass in a new machine code with an opcode that requires a shift, with an immediate value of zero and ensure it is correct
- Change the immediate in the machine code to a few other values and ensure those are correct without forgetting the shift

SID++ Instruction Set RTL

Inst.	R-Type	TST	JALR	B-Type
Fetch	IR <= MEM[PC]			
Decode	rd <= Reg[IR[8:11]] A <= Reg[IR[4:7]] ALUOut <= PC + SE(imm[1]<<1) PC <= PC + 2 B <= Reg[IR[0:3]]			
Execute	ALUOut <= A op B		ALUOut <= A + SE(imm[2])	
Memory				
Write Back	Reg[IR[8:11]] <= ALUOUT	Reg[15]<= A compare[3] B	Reg[IR[8:11]] <= PC PC <= ALUOut	if(B compare Reg[15]) == 0 PC <= ALUOut

Inst.	SLLI/SRLI/ADDI	LUI	LW/SW
Fetch	IR <= MEM[PC]		
Decode	rd <= Reg[IR[8:11]] A <= Reg[IR[4:7]] ALUOut <= PC + SE(imm[1]<<1) PC <= PC + 2 B <= Reg[R[0:3]]		
Execute	ALUOut <= rd op SE(imm[4])		ALUOut <= A + SE(imm[3])
Memory			sw: Mem[ALUOut] <= B lw: MDR <= Mem[ALUOut]
Write Back	Reg[IR[8:11]] <= ALUOut	REG[IR[8:11]] <= SE(imm[4]<<8)	lw: Reg[IR[8:11]] <= MDR

[1] IR[0:11]
[2] IR[0:3]
[3] compare: if A and B are equal TSTOut = 0; if A >= B, TSTOut = 1; if A < B, TSTOut = -2
[4] IR[0:7]

SID++ RTL Verification

R-Type Verification

Setup

Running instruction “add 3, 4, 5” with Registers 4 and 5 loaded with small integers.

Expected Behavior

The values stored in Registers 3 and 4 will be added and stored in register 3.

Pass Condition

The value of Reg4 + Reg5 will be stored in Reg3

State	Executing	Effects
Fetch	IR <= MEM[PC]	The instruction is loaded into the *Instruction Register*
Decode	rd <= Reg[IR[8:11]] A <= Reg[IR[4:7]] B <= Reg[IR[0:3]] ALUOut <= PC + SE(imm<<1) PC <= PC + 2	Stores the return register’s number in rd Stores the value of Reg 4 in A Stores the value of Reg 5 in B imm adding (Unused for this inst) Increment PC
Execute	ALUOut <= A op B	Adds A and B (values of Registers 4 and 5) and stores value in ALUOut
Memory		
Write Back	Reg[IR[8:11]] <= ALUOut	Stores the result of adding Registers 4 and 5 in the return register

Result: **PASS**

TST Verification

Setup

Running “TST 3, 4”, with a larger value placed in register 3 than 4.

Expected Behavior

The values in registers 3 and 4 will be compared, and will output the result to Reg15, with the condition:

```
if(R[rs1] = R[rs2]) R[15] = 0
if(R[rs1] > R[rs2]) R[15] = 1
if(R[rs1] < R[rs2]) R[15] = 2
```

Pass Condition

Since the value in Reg3 is larger, and it corresponds to rs1, the value 1 must be stored in R15.

State	Executing	Effects
Fetch	IR <= MEM[PC]	The instruction is loaded into the *Instruction Register*
Decode	rd <= Reg[IR[8:11]] A <= Reg[IR[4:7]] B <= Reg[IR[0:3]] ALUOut <= PC + SE(imm<<1) PC <= PC + 2	Stores the return register’s number in rd, this value is never ysed Stores the value of Reg 3 in A Stores the value of Reg 4 in B imm adding (Unused for this inst) Increment PC
Execute		
Memory		
Write Back	Reg[15] <= A Compare B	Stores the result of comparing the values in A and B, in this case 1 since Reg3 < Reg4, in Reg15

Result: PASS

JALR Verification

Setup

Running instruction “jalr 1, 4(3)” with Register 3 loaded with a valid instruction address.

Expected Behavior

The value of PC will be updated to the address stored in Register 3 + the imm, which in this case is 4.

Pass Condition

PC ends with the value in Register 3 plus 4.

State	Executing	Effects
Fetch	IR <= MEM[PC]	The instruction is loaded into the *Instruction Register*
Decode	rd <= Reg[IR[8:11]] A <= Reg[IR[4:7]] B <= Reg[IR[0:3]] ALUOut <= PC + SE(imm<<1) PC <= PC + 2	Stores the return register’s number in rd Stores the value of Reg 3 in A Stores the imm value imm adding (Unused for this inst) Increment PC
Execute	ALUOut <= A + SE(imm)	Stores the value of Register 3 + the imm in ALUOut
Memory		
Write Back	PC <= ALUOut rd <= PC	Stores the calculated correct value in PC Store PC + 2 in rd

Result: **PASS**

B-Type Verification

Setup

Running instruction “beq 3”, with TST already having been done on some unknown registers, leaving a negative number if less than, 0 if equal to, and a positive number if greater than in R15 based on the values in the registers.

Expected Behavior

The PC will equal the expected address if r15 is equal to 0.

Pass Condition

PC skips 3 instructions if r15 = 0.

State	Executing	Effects
Fetch	IR <= MEM[PC]	The instruction is loaded into the *Instruction Register*
Decode	rd <= Reg[IR[8:11]] A <= Reg[IR[4:7]] B <= Reg[IR[0:3]] ALUOut <= PC + SE(imm<<1) PC <= PC + 2	Stores the return register’s number in rd, this value is never used Stores the value of Reg 0 in A (not used) Stores the value of Reg 0 in B(not used) imm adding (Unused for this inst) Increment PC
Execute		
Memory		
Write Back	if(BC compare Reg[15]) == 0 PC <= ALUOut	if BC compare == 0, then the PC goes to the new address

Result: **PASS**

SLLI/SRLI/ADDI Verification

Setup

Running instruction slli 4, 1, with a number to be shifted left once placed into register 4.

Expected Behavior

The value in register 4 will be shifted left once.

Pass Condition

The value stored in Reg 4 will be the old value * 2.

State	Executing	Effects
Fetch	IR <= MEM[PC]	The instruction is loaded into the *Instruction Register*
Decode	rd <= Reg[IR[8:11]] A <= Reg[IR[4:7]] B <= Reg[IR[0:3]] ALUOut <= PC + SE(imm<<1) PC <= PC + 2	Stores the return register's number in rd Stores the value of Reg 0 in A (unused) Stores the value of Reg 1 in B (unused) imm adding (Unused for this inst) Increment PC
Execute	ALUOut <= rd op SE(imm)	This takes the value in rd and does the correct operation to the value.
Memory		
Write Back	Reg[IR[8:11]] <= ALUOut	This stores the value back into the destination register.

Result: **PASS**

LUI Verification

Setup

Running instruction lui x3, 0x12.

Expected Behavior

The upper half of the register x3 will have the value 0x12.

Pass Condition

The value stored in x3 is 0x1200.

State	Executing	Effects
Fetch	IR <= MEM[PC]	The instruction is loaded into the *Instruction Register*
Decode	rd <= Reg[IR[8:11]] A <= Reg[IR[4:7]] B <= Reg[IR[0:3]] ALUOut <= PC + SE(imm<<1) PC <= PC + 2	Stores the return register's number in rd Stores the value of Reg 1 in A (unused) Stores the value of Reg 2 in B (unused) imm adding (Unused for this inst) Increment PC
Execute		
Memory		
Write Back	REG[IR[8:11]] <= SE(imm<<8)	This stores the immediate into the top half of the register by shifting it left 8 bits.

Result: PASS

LW/SW Verification

Setup

Running instruction lw x3, 0(x4), where x4 has an address from memory in the register.

Expected Behavior

The value from the address in x4 + the immediate in memory will be stored in x3.

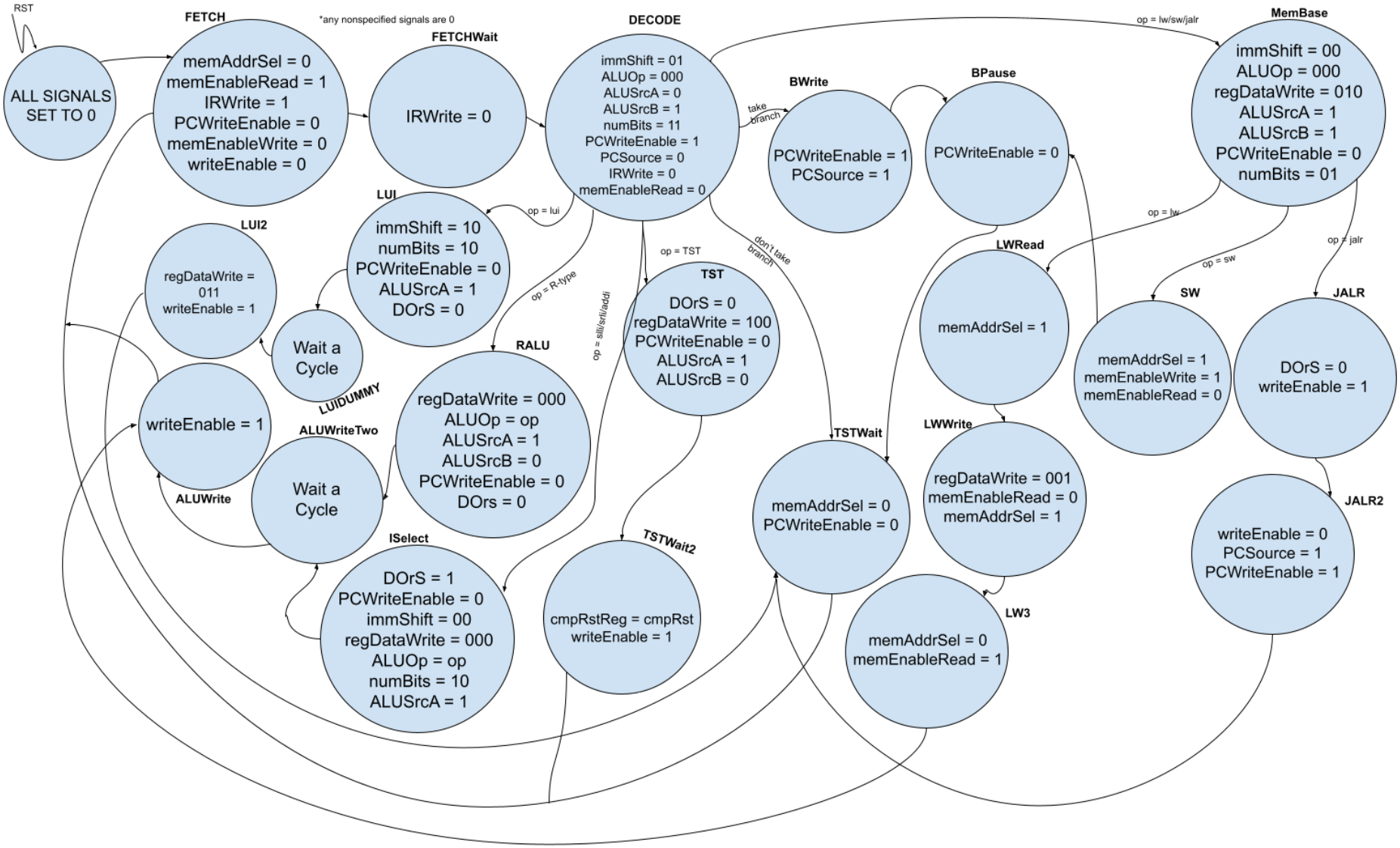
Pass Condition

The value in x3 will be the same as the value in memory at the address in x4 + 0.

State	Executing	Effects
Fetch	IR <= MEM[PC]	The instruction is loaded into the *Instruction Register*
Decode	rd <= Reg[IR[8:11]] A <= Reg[IR[4:7]] B <= Reg[IR[0:3]] ALUOut <= PC + SE(imm<<1) PC <= PC + 2	Stores the return register's number in rd, this value is never used Stores the value of Reg 4 in A Stores the value of Reg 0 in B (unused) imm adding (Unused for this inst) Increment PC
Execute	ALUOut <= A + SE(imm)	This determines the byte address in memory
Memory	lw: MDR <= Mem[ALUOut] sw: Mem[ALUOut] <= B	This puts the value from memory at the address of ALUOut into MDR.
Write Back	lw: Reg[IR[8:11]] <= MDR	This sets the value of x3 to the value of MDR.

Result: **PASS**

SID++ Control Finite State Machine



SID++ Control Signal Specifications

Signal Name	**Description**
immShift	Tells ‘Imm Gen’ to shift or not, and by how much: Don’t shift: 00 Shift by 1: 01 Shift by 8: 10
op	Sends the OPcode to the control unit from the IR
cmpRst	Sends the result of a compare operation to the control unit: A = B: 00 A > B: 01 A < B: 10
ALUOp	Tells the ALU which operation to complete. Add: 000 Sub: 001 And: 010 Or: 011 Xor: 100 Shift left 1: 101 Shift right 1: 110
writeEnable	Enables writing to ‘rd’ register when ‘1’
numBits	Tells the ‘Imm Gen’ how many bits to use when creating the immediate Output 0: 00 4 bits: 01 8 bits: 10 12 bits: 11
memAddrSel	Chooses between ‘PC’ and ‘ALUOut’ as the input for memory address
ALUSrcA	Choose input A for the ALU, ‘0’ for ‘PC’, ‘1’ for ‘Reg[A]’
ALUSrcB	Choose input B for the ALU, ‘0’ for ‘Reg[B]’, ‘1’ for ‘Imm Gen’

memEnableRead	Enables reading of the memory at the given address when ‘1’
memEnableWrite	Enables writing to the given address in memory when ‘1’
PCWriteEnable	Allows the PC to be written to when ‘1’
PCSource	Switches PC input between ‘PC+2’ when ‘0’ and ‘ALUOut’ when ‘1’
regDataWrite	Switches data written to the register file: ALUOut: 000 MDR: 001 PC: 010 Imm Gen: 011 Comparator Output: 100
IRWrite	Allows writing to the instruction register when 1.

SID++ Control Signal Tests

immShift

- Input an immediate-necessary instruction that would not require a shift. Ensure the immediate was not shifted and that *immShift* was 0.
- Input an immediate-necessary instruction that *requires* a shift. Check the immediate value to make sure it was shifted and that *immShift* was 1.
- Input an instruction that does not need an immediate. Ensure *immShift* in this case is a don't care.

op

- Send each of the 16 opcodes once. Ensure that the other control signals are being set to the correct value and that the opcode is recognized as the correct instruction.

cmpRst

- Input an instruction that compares values, making sure you get each of the three cases once.
- If the values are equal, make sure *cmpRst* is 00. If A is greater, check that *cmpRst* is 01. If B is greater, ensure *cmpRst* is 10.

ALUOp

- Input an instruction for each operation of the ALU
- Make sure *ALUOp* is giving the correct value based on the control signal specifications as specified above.

writeEnable

- Input an instruction that requires writing to a register.
- Ensure *writeEnable* is set to 1, and that rd is changed at the end of the instruction.
- Input an instruction the will **not** write to a register.
- Ensure *writeEnable* is set to 0, and the value of rd has stayed the same.

numBits

- Input each instruction that does not require an immediate. *numBits* should be 00, and should force the immediate to be 0 just in case.
- Input each instruction that has a 4 bit immediate. *numBits* should be 01, and the immediate should be equal to the value given in the instruction.
- Input each instruction that has an 8 bit immediate. *numBits* should be 10, and the immediate should be equal to the value given in the instruction.
- Input each instruction that has a 12 bit immediate. *numBits* should be 11, and the immediate should be equal to the value given in the instruction.

memAddrSel

- Input each instruction that requires the PC to be used as the memory address. *memAddrSel* should be set to 0 for each of these.
- Input each instruction that requires ALUOut to be the memory address. *memAddrSel* should be set to 1 for each of these.
- Input each instruction that does not require the memory module. *memAddrSel* is a don't care in this case.

ALUSrcA

- Feed the control unit a few instructions that require register A to be PC based to ensure value 1 is outputted.
- Send a few instructions to the control unit that require register A to be a register from the register file and ensure they output 0.

ALUSrcB

- Feed the control unit a few instructions that require the immediate to be utilized. Ensure the value output is 1.
- Feed the control unit a few instructions that require a register from the register file to be used. Ensure the value output is 0.

memEnableRead

- Send the control unit opcodes that require memory to be read. Ensure this value is 1.
- Send all the opcodes that do not require memory to be accessed. Ensure these opcodes produce a value 0.

memEnableWrite

- Send the control unit opcodes that require memory addresses to be changed. Make sure the output is 1.
- Send the control unit all of the opcodes where memory is not changed. Make sure the output is 0.

PCWriteEnable

- Make sure *PCWriteEnable* starts out as 1 for the first cycle of the multi cycle clock.
- For all subsequent cycles of the instruction it must be set to 0.

PCSource

- Send the control unit opcodes that simply increment the PC. Make sure the output is 0.
- Send the control unit all of the opcodes where the PC is written to from ALUOut. Make sure the output is 1.

regDataWrite

- Send the control unit opcodes that require memory to be saved as a register. Make sure the output is 1.
- Send the control unit all of the opcodes where memory is not saved to a register. Make sure the output is 0.

IRWrite

- Make sure *loadInst* starts out as 1 for the first cycle of the multi cycle clock.
- For all subsequent cycles of the instruction it must be set to 0.

SID++ System Verification

SYS_TEST_add_0_0_into_t1 - adds two empty registers, t1 and t2, and puts the result in t1.
Expected: t1 = 0

SYS_TEST_add_1_0_into_t1 - adds 1 into t1 with addi, and adds t1 and r0 and puts the result in t1.
Expected: t1 = 1

SYS_TEST_add_1_2_into_t1 - adds 1 into t1 and adds 2 into t2 with addi, and adds t1 and t2 and puts the result in t1.
Expected: t1 = 3, t2 = 2

SYS_TEST_addi_0_into_t1 - adds 0 into t1 with addi.
Expected: t1 = 0

SYS_TEST_addi_1_into_t1 - adds 1 into t1 with addi.
Expected: t1 = 1

SYS_TEST_addi_2_into_loaded_t1 - adds 1 into t1 and then adds 2 into t1.
Expected: t1 = 3

SYS_TEST_addi_neg1_into_t1 - adds -1 into t1 with addi.
Expected: t1 = -1

SYS_TEST_and_0_0_into_t1 - ands the zero register with itself and puts the result in t1.
Expected: t1 = 0

SYS_TEST_and_7_0_into_t1 - addis 7 into t1, ands it with 0, and puts the result in t1.
Expected: t1 = 0

SYS_TEST_and_7_3_into_t1 - addis 7 into t1 and 3 into t2, ands them together, and puts the result in t1.
Expected: t1 = 3, t2 = 3

SYS_TEST_and_7_7_into_t1 - addis 7 into t1, ands t1 with itself, and puts the result in t1.

Expected: t1 = 7

SYS_TEST_lui_1_into_t1 - luis 1 into t1.

Expected: t1 = 256

SYS_TEST_lui_255_into_t1 - luis 255 into t1.

Expected: t1 = 65280

SYS_TEST_or_0_0_into_t1 - ors the zero register with itself and puts the result in t1.

Expected: t1 = 0

SYS_TEST_or_7_0_into_t1 - addis 7 into t1 and ors it with the zero register, and puts the result in t1.

Expected: t1 = 7

SYS_TEST_or_7_3_into_t1 - addis 7 into t1 and 3 into t2, ors them together, and puts the result in t1.

Expected: t1 = 7

SYS_TEST_or_7_7_into_t1 - addis 7 into t1 and ors it with itself, and puts the result in t1.

Expected: t1 = 7

SYS_TEST_slli_1_by1_into_t1 - addis 1 into t1 and shifts it left by 1 bit.

Expected: t1 = 2

SYS_TEST_slli_1_by4_into_t1 - addis 1 into t1 and shifts it left by 4 bits.

Expected: t1 = 16

SYS_TEST_slli_2^15_by1_into_t1 - luis 128 into t1 and shifts it left by 1 bit.

Expected: t1 = 0

SYS_TEST_srli_1_by1_into_t1 - addis 1 into t1 and shifts it right by 1 bit.

Expected: t1 = 0

SYS_TEST_srli_2^15_by1_into_t1 - luis 128 into t1 and shifts it right by 1 bit.

Expected: t1 = 16384

SYS_TEST_srli_2^15_by4_into_t1 - luis 128 into t1 and shifts it right by 4 bits

Expected: t1 = 2048

SYS_TEST_sub_0_0_into_t1 - subtracts two empty registers, t1 and t2,and puts the result in t1.

Expected: t1 = 0

SYS_TEST_sub_1_5_into_t3 - addis 1 into t1 and 5 into t2, subtracts t1 - t2, and puts the result in t3.

Expected: t3 = -4

SYS_TEST_sub_5_1_into_t3 - addis 1 into t1 and 5 into t2, subtracts t2 - t1, and puts the result in t3.

Expected: t3 = -4

SYS_TEST_xor_0_0_into_t1 - xors the zero register with itself and puts the result in t1.

Expected: t1 = 0

SYS_TEST_xor_7_0_into_t1 - addis 7 into t1, xors it with the zero register, and puts the result in t1.

Expected: t1 = 7

SYS_TEST_xor_7_3_into_t1 - addis 7 into t1 and 3 into t2, xors them together, and puts the result in t1.

Expected: t1 = 4

SYS_TEST_xor_7_7_into_t1 - addis 7 into t1 and t2, xors them together, and puts the result in t1.

Expected: t1 = 0

SID++ Data

Total Storage = 128 Bytes

Flow Status	Successful - Mon May 22 19:14:37 2023
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	processorDatapath
Top-level Entity Name	CPU
Family	Cyclone IV E
Total logic elements	1,084 / 22,320 (5 %)
Total registers	555
Total pins	34 / 80 (43 %)
Total virtual pins	0
Total memory bits	524,288 / 608,256 (86 %)
Embedded Multiplier 9-bit elements	0 / 132 (0 %)
Total PLLs	0 / 4 (0 %)
Device	EP4CE22E22C6
Timing Models	Final

Total Logic Elements = 1,084
Total Registers = 555
Total Memory bits = 524,288

Total Cycles = 378,443
Total Instructions = 71,565
Average CPI = 5.288

	Fmax	Restricted Fmax	Clock Name	Note
1	94.8 MHz	94.8 MHz	CLK	
2	159.54 MHz	159.54 MHz	controlUnit:UTT|current_state.MemBase	
3	159.69 MHz	159.69 MHz	controlUnit:UTT|current_state.DECODE	
4	170.18 MHz	170.18 MHz	ALUOpReg[0]	
5	621.12 MHz	621.12 MHz	controlUnit:UTT|current_state.FETCH	

Clock Frequency = 94.8 MHz
Cycle Time = 10.5 nS

Runtime for 0x13B0 = 3.97 mS