DaVinci 1.0 System design and implementation

Catalina Lamboglia
San Jose State University
catalina.lamboglia@sjsu.edu

Introduction

This report will describe the requirements for the DaVinci 1.0 system, the implementation, and the used testing strategies.
Structure:

1. Explanation of “CS147DV” and DaVinci 1.0 system
2. ALU design, implementation, and testing
3. Memory design, implementation, and testing
4. Register file design, implementation, and testing
5. Control unit
6. Entire system testing
7. Conclusion

This report will cover the ALU's functions and requirements, every command within the “CS147DV” instruction set, how the memory, register file, and control unit for DaVinci 1.0 were designed and implemented, a breakdown for the control signals for “CS147DV”, and how the entire system is tested.

I. Explanation of “CS147DV”'s and DaVinci 1.0 system

The DaVinci 1.0 system is a basic computing system which can compute the instruction set outlined within CS147DV. The instructions are as follows:

Name	Mnemonic	Format	Operation	funct – Operation code
Addition	add	R	R[rd]=R[rs]+R[rt]	0x20 funct
Subtraction	sub	R	R[rd]=R[rs]-R[rt]	0x22 funct
Multiplication	mul	R	R[rd]=R[rs]*R[rt]	0x2c funct
Shift Right Logical	srl	R	R[rd]=R[rs] >> shamt	0x02 funct
Shift Left Logical	sll	R	R[rd]=R[rs] << shamt	0x01 funct
Logical AND	and	R	R[rd]=R[rs]&R[rt]	0x24 funct
Logical OR	or	R	R[rd]=R[rs] | R[rt]	0x25 funct
Logical NOR	nor	R	R[rd]=~(R[rs] | R[rt])	0x27 funct
Set Less Than	slt	R	R[rd]=(R[rs]<R[rt])?1:0	0x2a funct
Jump Register	jr	R	PC = R[rs]	0x08 funct
Addition immediate	addi	I	R[rt] = R[rs] + SignExtImm	0x08 (opcode)
Multiplication Immediate	muli	I	R[rt] = R[rs] * SignExtImm	0x1d
Logical AND	andi	I	R[rt] = R[rs] & ZeroExtImm	0x0c

Immediate				
Logical OR Immediate	ori	I	R[rt] = R[rs] | ZeroExtImm	0x0d
Load upper Immediate	lui	I	R[rt] = {imm, 16'b0}	0x0f
Set less than immediate	slti	I	R[rt] = (R[rs] < SignExtImm)? 1:0	0x0a
Branch on equal	beq	I	If (R[rs] == R[rt]) PC = PC + 1 + BranchAddress	0x04
Branch on not equal	bne	I	If (R[rs] != R[rt]) PC = PC + 1 + BranchAddress	0x05
Load word	lw	I	R[rt] = M[R[rs]+SignExtImm]	0x23
Store word	sw	I	M[R[rs]+SignExtImm] = R[rt]	0x2b
Jump to address	jmp	J	PC = JumpAddress	0x02
Jump and Link	jal	J	R[31] = PC + 1; PC = JumpAddress	0x03
Push to Stack	push	J	M[$sp] = R[0] $sp = $sp - 1	0x1b
Pop from Stack	pop	J	$sp = $sp + 1 R[0] = M[$sp]	0x1c

The DaVinci 1.0 System is composed of a microprocessor and memory unit. Within the microprocessor is the ALU, register file, and controller. The memory is indirectly addressed.

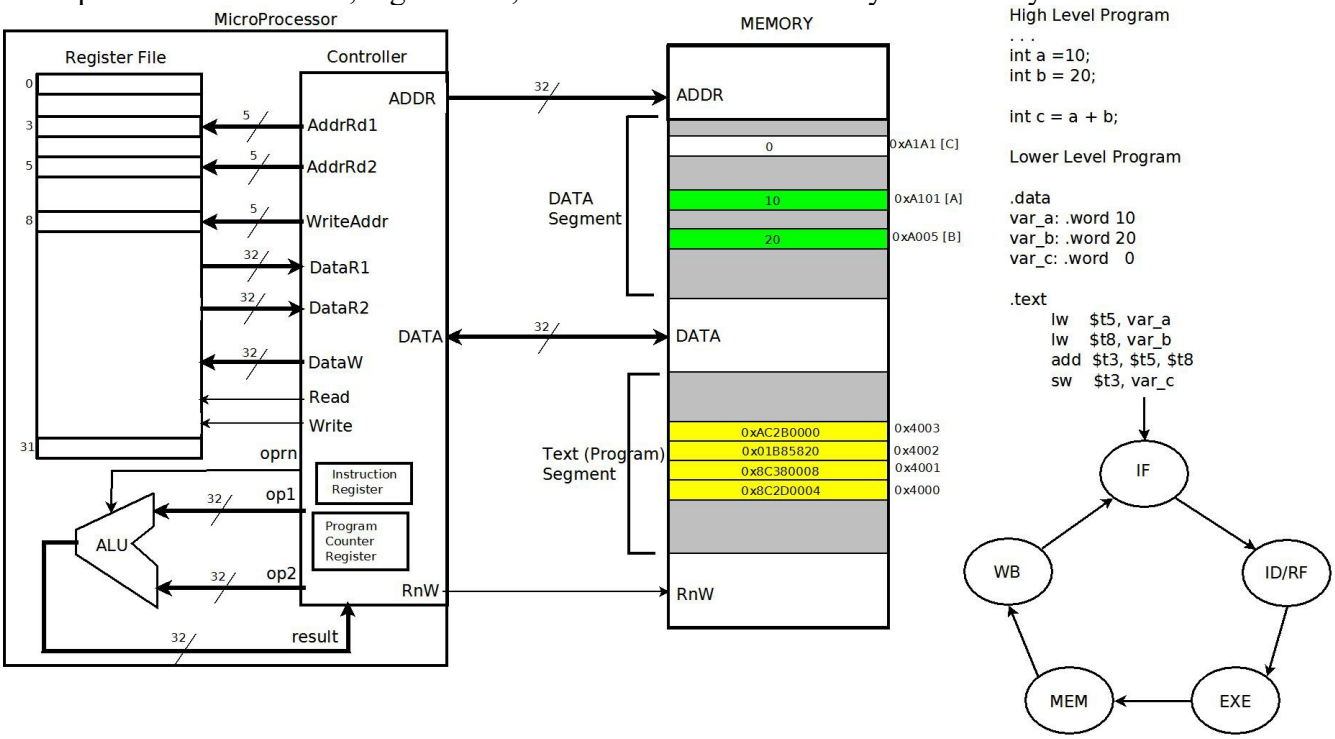


Image of DaVinci 1.0 system by Kaushik Patra (kaushik.patra@sjsu.edu) supplied through canvas.

II. ALU design, implementation, and testing

The ALU is an arithmetic logic unit that can perform operations such as addition, subtraction, multiplication, bitwise and, or, nor, bit shifts to left or right, set less than.

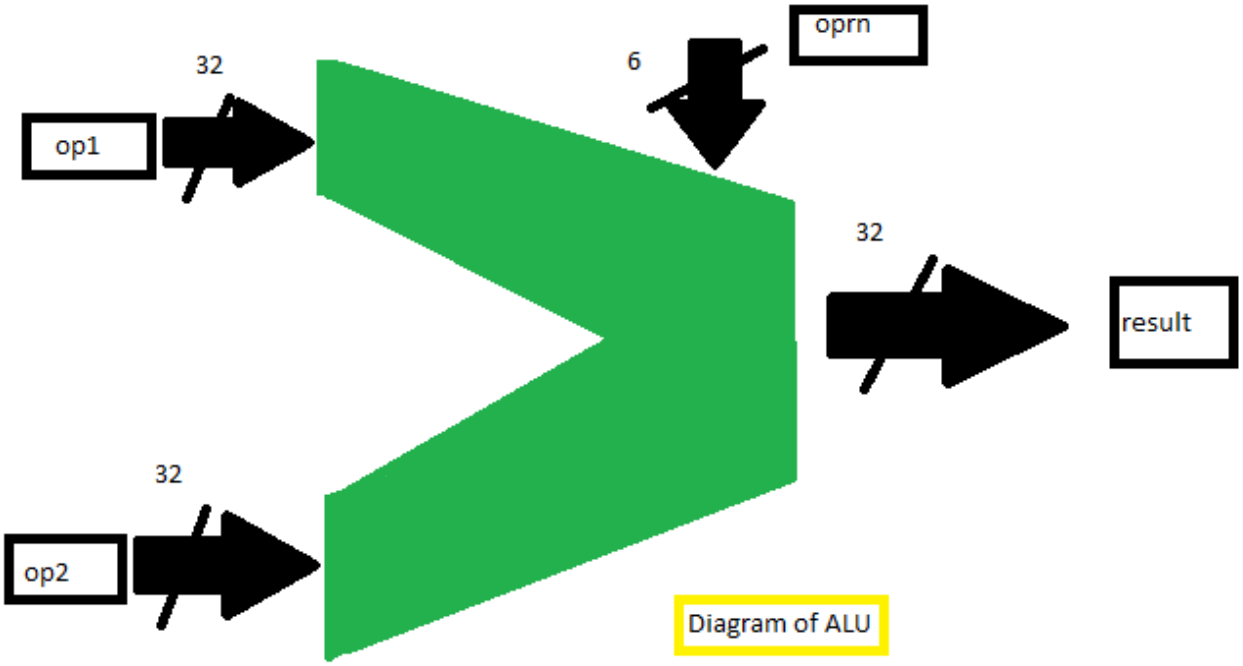


Diagram of ALU

The ALU takes in two 32 bit operations as inputs, a 6 bit oprn code, performs the operation based on the oprn code, and then produces an output.

The above listed operations can be replicated/implemented using Verilog. For example, R[rd] = R[rs]+R[rt] can be implemented as simply result = op1 + op2; where each register is respectively the other (R[rd] is result, etc). The controller of course controls what goes in and how the result is used. Therefore, the ALU can be used as an incrementer or to test for equality, etc. For example, when using a shift operation, op2 becomes shamt in that case. A zero flag/output returns 1 if the result is ever 0, else it returns 0. (This is useful if you need to check if op1 = op2 or not).

Operation (Operation code)	Verilog Implementation
Addition (0x1)	`ALU_OPRN_WIDTH'h01 : result = op1 + op2;
Subtraction (0x2)	`ALU_OPRN_WIDTH'h02 : result = op1 - op2;
Multiplication (0x3)	`ALU_OPRN_WIDTH'h03 : result = op1 * op2;
Shift Right Logical (0x4)	`ALU_OPRN_WIDTH'h04 : result = op1 >> op2;
Shift Left Logical (0x5)	`ALU_OPRN_WIDTH'h05 : result = op1 << op2;
Bitwise And (0x6)	`ALU_OPRN_WIDTH'h06 : result = (op1 & op2);
Bitwise Or (0x7)	`ALU_OPRN_WIDTH'h07 : result = (op1 | op2);
Bitwise Nor (0x8)	`ALU_OPRN_WIDTH'h08 : result = ~(op1 | op2);

Set Less Than (0x9)	`ALU_OPRN_WIDTH'h09 : result = op1 < op2;

Each operation is performed when oprn is equivalent to the operation's respective operation code. In a case where oprn is out of this range a default case is invoked. Full code is as follows:

```
`include "prj_definition.v"
module ALU(OUT, ZERO, OP1, OP2, OPRN);
// input list
input [`DATA_INDEX_LIMIT:0] OP1; // operand 1
input [`DATA_INDEX_LIMIT:0] OP2; // operand 2
input [`ALU_OPRN_INDEX_LIMIT:0] OPRN; // operation code

// output list
output [`DATA_INDEX_LIMIT:0] OUT; // result of the operation.
output ZERO;

// reg list
reg [`DATA_INDEX_LIMIT:0] OUT;
reg ZERO;

always @ (OUT)
begin
    if (OUT === 0)
        ZERO = 1;
    else
        ZERO = 0;
end

always @ (OP1 or OP2 or OPRN)
begin

    case (OPRN)
        `ALU_OPRN_WIDTH'h01 : OUT = OP1 + OP2; // addition
        `ALU_OPRN_WIDTH'h02 : OUT = OP1 - OP2; // Subtraction
        `ALU_OPRN_WIDTH'h03 : OUT = OP1 * OP2; // Multiplication
        `ALU_OPRN_WIDTH'h04 : OUT = OP1 >> OP2; // Shift right
        `ALU_OPRN_WIDTH'h05 : OUT = OP1 << OP2; // Shift left
        `ALU_OPRN_WIDTH'h06 : OUT = (OP1 & OP2); // Bitwise AND
        `ALU_OPRN_WIDTH'h07 : OUT = (OP1 | OP2); // Bitwise OR
        `ALU_OPRN_WIDTH'h08 : OUT = ~(OP1 | OP2); // Bitwise NOR
        `ALU_OPRN_WIDTH'h09 : OUT = OP1 < OP2;    // Set Less Than
        default: OUT = `DATA_WIDTH'hxxxxxxxx;
    endcase
end

endmodule
```

When op1, op2, or oprn change in value this block of code is invoked and the operation is determined by oprn.

Addition (h01): A simple addition of op1 and op2 is performed then the result is saved to the result register.

Subtraction (h02): A simple subtraction of op2 from op1 is performed then the result saved to the result register.

Multiplication (h03): A simple multiplication of op1 by op2 is performed then the result is saved to the result register.

Shift Right (h04): op1 is shifted by the amount of op2 (acting as shamt). For example, if you have op1=4 and op2=2 then op1 is equivalent to 100 in binary and is shifted to the right twice resulting in 001. In decimal that is equivalent to 1. Essentially this is a division by 2*shamt ((op1)/(op2*2)).

Shift Left (h05): Similar to Shift Right, using the previous example, op1=4 and op2=2, op1 would then become 10000 in binary (16 in decimal). Essentially a multiplication by 2 ((op1)*(op2*2)).

Bitwise AND (h06): Compares the bits of op1 and op2 and returns a value, where every bit was the same, otherwise 0. Example: Assume op1=1101001 (in binary) and op2=0111101 (in binary). These two are compared bit by bit and the result is saved to result. For this example 0101001 would be the result.

Bitwise OR (h07): Compares the bits of op1 and op2 and returns a value, where any bit is 1 then that bit is 1 as a result. Example: op1=1110001 (in binary) and op2 = 0010010 (in binary) then

result = 1110011.

Bitwise NOR (h08): Is the not operation on OR. Perform the OR operation first, then perform a NOT on it (flip each bit). From the previous example, the NOT operation on result would be 0001100.

Set Less Than (h09): Set result to 1 if op1 is less than op2, otherwise set result to 0.

ALU Testing strategy and Implementation

Test Overview

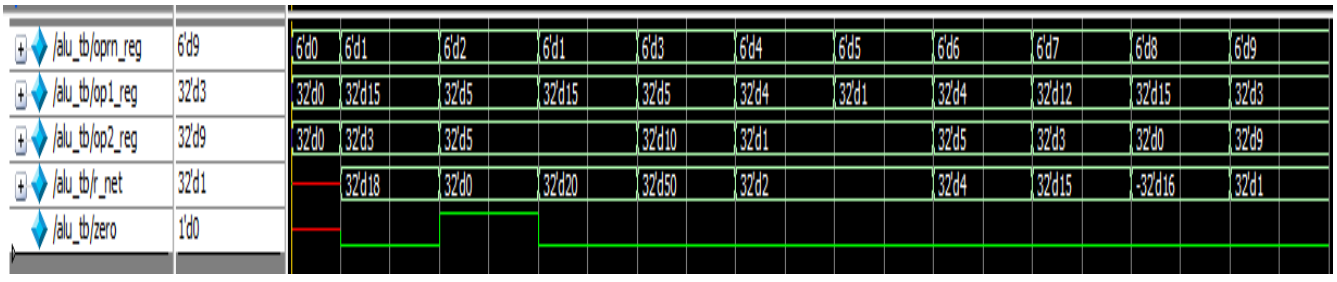
```
# [TEST] 15 + 3 = 18 , got 18 ... [PASSED]
# [TEST] 5 - 5 = 0 , got 0 ... [PASSED]
# [TEST] 15 + 5 = 20 , got 20 ... [PASSED]
# [TEST] 5 * 10 = 50 , got 50 ... [PASSED]
# [TEST] 4 >> 1 = 2 , got 2 ... [PASSED]
# [TEST] 1 << 1 = 2 , got 2 ... [PASSED]
# [TEST] 4 AND 5 = 4 , got 4 ... [PASSED]
# [TEST] 12 OR 3 = 15 , got 15 ... [PASSED]
# [TEST] 15 NOR0 = 4294967280 , got 4294967280 ... [PASSED]
# [TEST] 3 <9 = 1 , got 1 ... [PASSED]
#
#       Total number of tests          10
#       Total number of pass           10
```

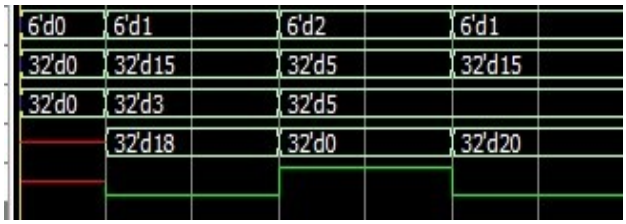
Test Cases:

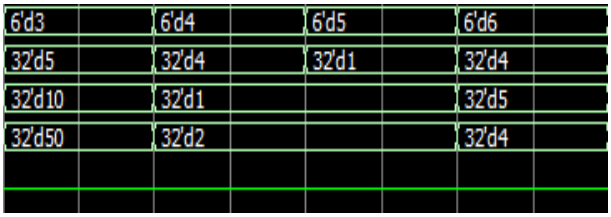
(Values in parenthesis are in binary, decimal otherwise)

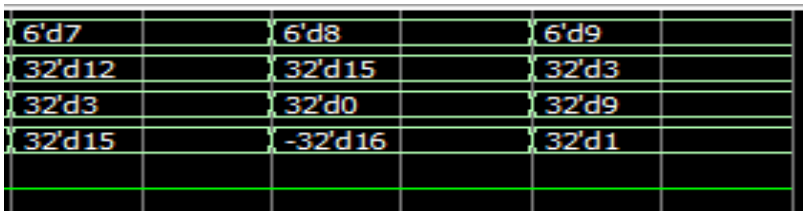
Operation	op1	op2/shamt	result
1 (add)	15	3	18
2 (sub)	5	5	0
1 (add)	15	5	20
3 (mul)	5	10	50
4 (srl)	4 (100)	1	2 (010)
5 (sll)	1 (001)	1	2 (010)
6 (and)	4 (0100)	5 (0101)	4 (0100)
7 (or)	12 (1100)	3 (0011)	15 (1111)
8 (nor)	15 (1111)	0 (0000)	[max int value]
9 (slt)	3	9	1

Waveform:









Operations were run on the ALU via the test bench and the testing code/block and ALU wavelength output verified the correctness of the operation set's implementation.

III. Memory design, implementation, and testing

The 64MB memory unit preloads data upon initialization or reset. Memory can be changed or accessed with sw and lw. The memory unit acts as “permanent” storage, as opposed to the more temporary nature of the register file. The only instructions that access the memory are lw, sw, push, and pop. (Implementation, testing, and preload data already predefined and supplied by Professor Kaushik Patra (kaushik.patra@sjsu.edu).

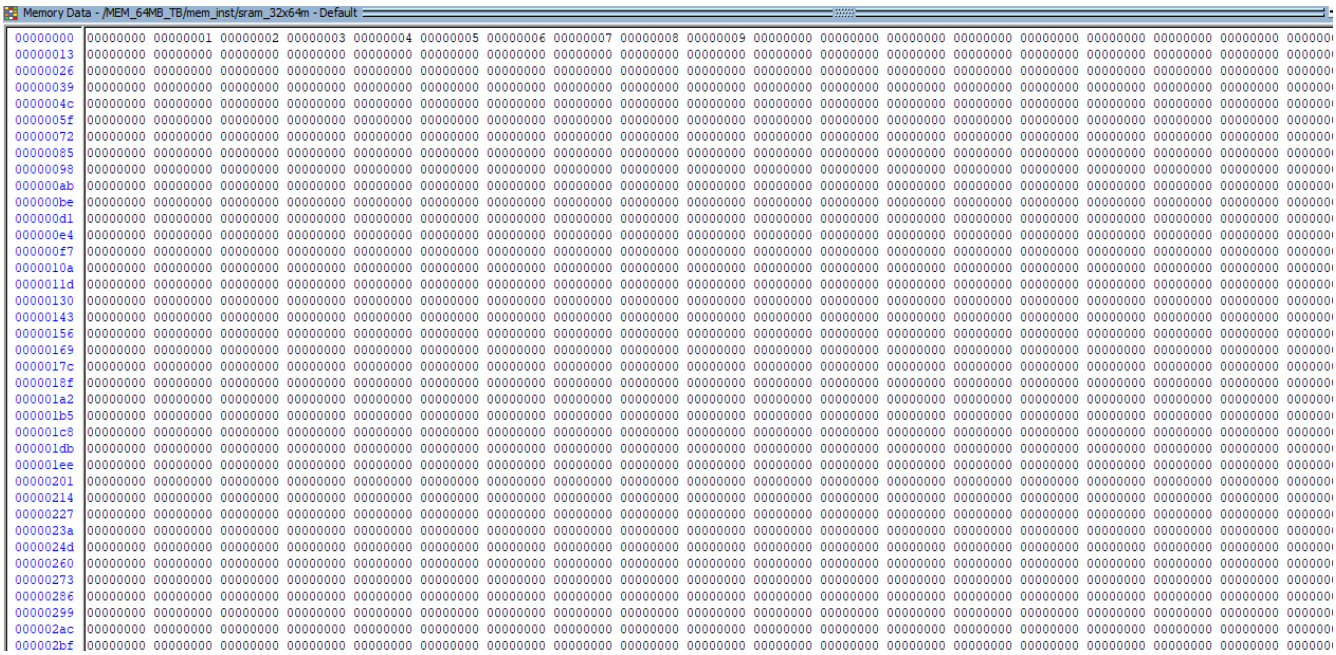
Preloaded data: (Contents of "mem_content_01.dat")

```
@0001000
00414020 00414021 00414022 00414023
00414024 00414025 00414026 00414027
00414028 00414029 0041402a 0041402b
0041402c 0041402d 0041402e 0041402f

@002f00a
00514020 00514021 00514022 00514023
00514024 00514025 00514026 00514027
00514028 00514029 0051402a 0051402b
0051402c 0051402d 0051402e 0051402f
```

Implementation is as follows:

```
`include "prj_definition.v"
module MEMORY_64MB(DATA, READ, WRITE, ADDR, CLK, RST);
// Parameter for the memory initialization file name
parameter mem_init_file = "mem_content_01.dat";
// input ports
input READ, WRITE, CLK, RST;
input [`ADDRESS_INDEX_LIMIT:0] ADDR;
// inout ports
inout [`DATA_INDEX_LIMIT:0] DATA;

// memory bank
reg [`DATA_INDEX_LIMIT:0] sram_32x64m [0:`MEM_INDEX_LIMIT]; // memory storage
integer i; // index for reset operation

reg [`DATA_INDEX_LIMIT:0] data_ret; // return data register

assign DATA = ((READ===1'b1)&&(WRITE===1'b0))?data_ret:{`DATA_WIDTH{1'bz} };

always @ (negedge RST or posedge CLK)
begin
if (RST === 1'b0)
begin
for(i=0;i<=`MEM_INDEX_LIMIT; i = i +1)
    sram_32x64m[i] = { `DATA_WIDTH{1'b0} };
$readmemh(mem_init_file, sram_32x64m);
end
else
begin
 if ((READ===1'b1)&&(WRITE===1'b0)) // read operation
        data_ret =  sram_32x64m[ADDR];
 else if ((READ===1'b0)&&(WRITE===1'b1)) // write operation
        sram_32x64m[ADDR] = DATA;
end
end
endmodule
```

Memory contents from testbench and output text:



The test strategy is to simply load some data into memory and then read it back while checking if it's equivalent to what was loaded into memory. For this test bench all that was done was loading in locations 0-9 with the data 0-9 respectively for each memory location (as in memory location 9 has the data "9"). So long as you can put data in and then retrieve it the memory unit is functioning properly.

IV. Register file design, implementation, and testing

The register file contains all the registers for the DaVinci 1.0 system. It consists of 32 32-bit registers. These registers are used mainly for temporary storage between operations. These registers are typically passed to other units of the system (ALU, memory, etc) to carry out some operation or to be stored to memory, etc.

The testing strategy is the same for the memory file in that you simply need to load some data in and then retrieve it and then register file will be functioning properly.

Implementation:

```
`include "prj_definition.v"
module REGISTER_FILE_32x32(DATA_R1, DATA_R2, ADDR_R1, ADDR_R2,
                           DATA_W, ADDR_W, READ, WRITE, CLK, RST);

// input list
input READ, WRITE, CLK, RST;
input [`DATA_INDEX_LIMIT:0] DATA_W;
input [`REG_ADDR_INDEX_LIMIT:0] ADDR_R1, ADDR_R2, ADDR_W;
```

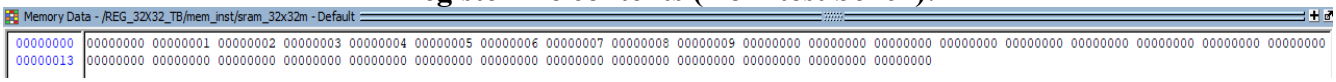
```
// output list
output [`DATA_INDEX_LIMIT:0] DATA_R1;
output [`DATA_INDEX_LIMIT:0] DATA_R2;

reg [`DATA_INDEX_LIMIT:0] DATA_R1;
reg [`DATA_INDEX_LIMIT:0] DATA_R2;

// memory bank
reg [`DATA_INDEX_LIMIT:0] sram_32x32m [0:`REG_INDEX_LIMIT]; // memory storage
integer i; // index for reset operation

initial
begin
for(i=0;i<=`REG_INDEX_LIMIT; i = i +1)
   sram_32x32m[i] = { `DATA_WIDTH{1'b0} };
end

always @ (negedge RST or posedge CLK)
begin
if (RST === 1'b0)
begin
for(i=0;i<=`REG_INDEX_LIMIT; i = i +1)
   sram_32x32m[i] = { `DATA_WIDTH{1'b0} };
end
else
begin
 if ((READ===1'b1)&&(WRITE===1'b0)) // read operation
 begin
          DATA_R1 =  sram_32x32m[ADDR_R1];
       DATA_R2 =  sram_32x32m[ADDR_R2];
 end
 else if ((READ===1'b0)&&(WRITE===1'b1)) // write operation
          sram_32x32m[ADDR_W] = DATA_W;
end
end

endmodule
```

Output text:

```
# [TEST] Read 1, Write 0, expecting 00000000, got xxxxxxxx [FAILED]
# [TEST] Read 1, Write 0, expecting 00000001, got 00000000 [FAILED]
# [TEST] Read 1, Write 0, expecting 00000002, got 00000001 [FAILED]
# [TEST] Read 1, Write 0, expecting 00000003, got 00000002 [FAILED]
# [TEST] Read 1, Write 0, expecting 00000004, got 00000003 [FAILED]
# [TEST] Read 1, Write 0, expecting 00000005, got 00000004 [FAILED]
# [TEST] Read 1, Write 0, expecting 00000006, got 00000005 [FAILED]
# [TEST] Read 1, Write 0, expecting 00000007, got 00000006 [FAILED]
# [TEST] Read 1, Write 0, expecting 00000008, got 00000007 [FAILED]
# [TEST] Read 1, Write 0, expecting 00000009, got 00000008 [FAILED]
# [TEST] Read 1, Write 0, expecting 32'hzzzzzzzz, got 00000001 [FAILED]
#
#       Total number of tests          11
#       Total number of pass            0
```

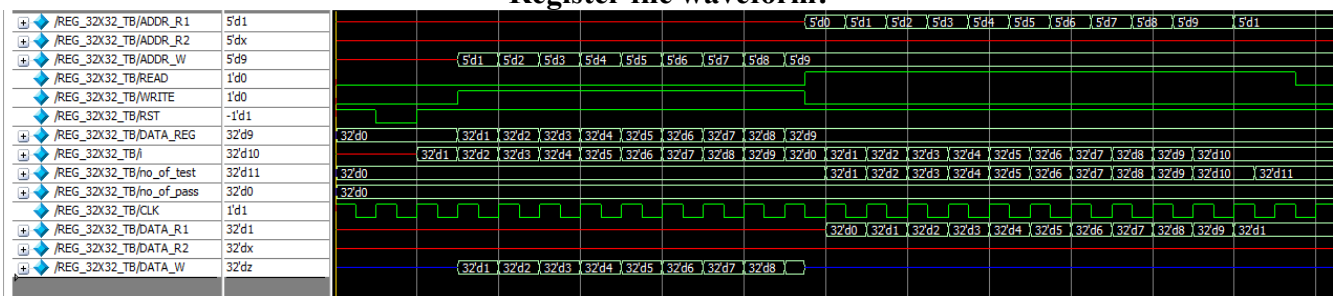
My test bench has a clock issue and the data checking is off by a cycle but from the memory contents it can be seen that the memory is stored and retrieved properly regardless of the test bench, therefore my register file is functioning properly.

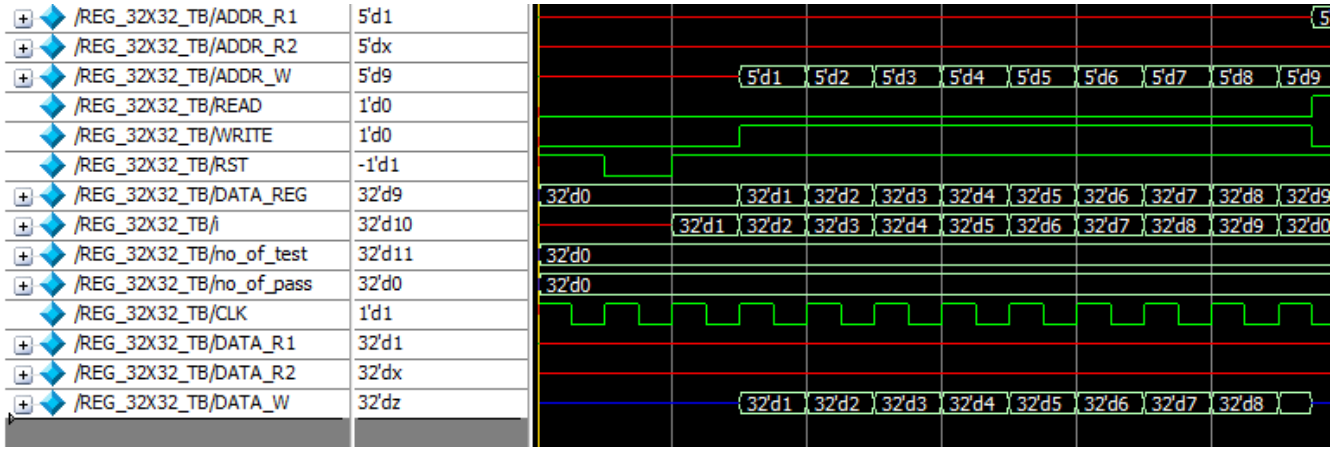
Register file contents (from test bench):

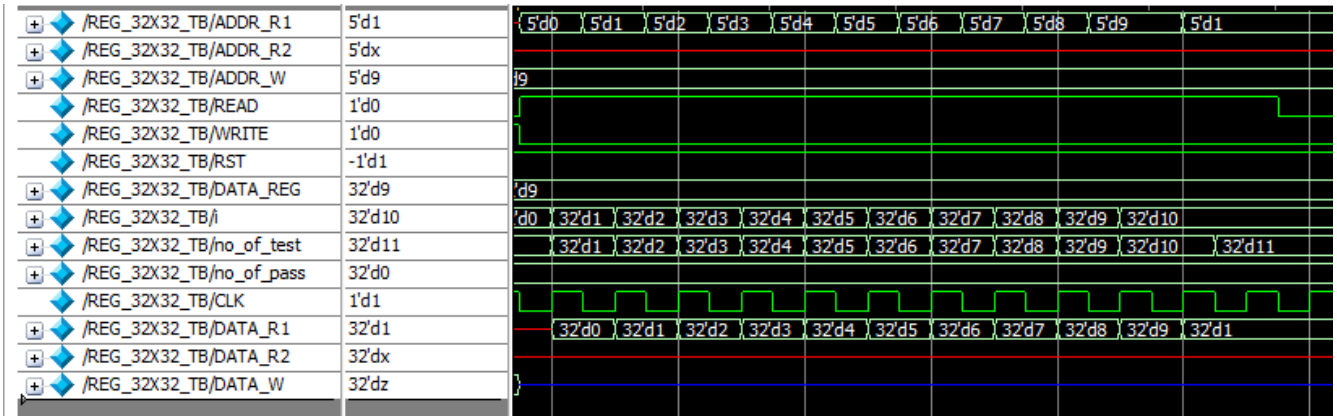



```
00000000 00000001 00000002 00000003 00000004 00000005 00000006 00000007 00000008 00000009
```

Register file waveform:







In the wave form the desync between clocks can be seen so the error is solely in my test bench and not the register file's implementation.

V. Controller

The controller's primary operation is passing and receiving data from the other components of the DaVinci 1.0 system. The controller acts as the brain/CPU of the system in that it can interpret instructions and control the data path. The controller processes instructions through instruction cycles.

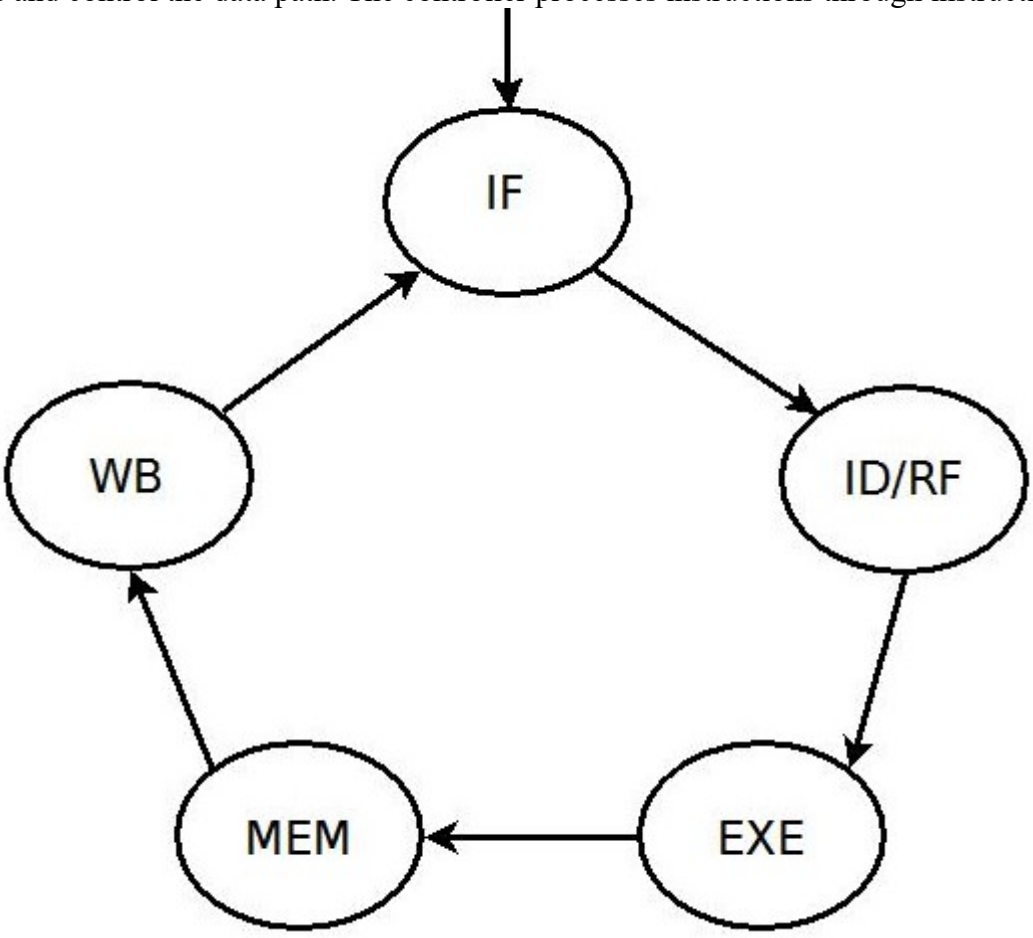


Image of DaVinci 1.0 system by Kaushik Patra (kaushik.patra@sjsu.edu) supplied through canvas.

These instruction cycles are as follows:
Instruction fetch (IF): The controller fetches the instruction from inputting the value within the PC register passed as an address to memory.

```
if (proc_state === `PROC_FETCH)
begin
    MEM_ADDR = PC_REG;
    MEM_READ = 1;
    MEM_WRITE = 0;
    RF_READ = 0;
    RF_WRITE = 0;
end
```

Instruction decode/register fetch (ID/RF): Decodes the value retrieved from memory and parses the retrieved machine code into the subsequences (opcode, rt, rs, etc). Also retrieves data from within registers.

```
if (proc_state === `PROC_DECODE)
begin
INST_REG = MEM_DATA;
{opcode, rs, rt, rd, shamt, funct} = INST_REG;
// I-type
{opcode, rs, rt, immediate } = INST_REG;
// J-type
{opcode, address} = INST_REG;

signExtImm = {{16{immediate[15]}}, immediate};
zeroExtImm = {{16{1'b0}}, immediate};
branchAddress = signExtImm + PC_REG + 1;
// Branch address is signExtImm + PC + 1

case(opcode)
// R-Type
6'h00 : begin
    case(funct)
        6'h20: begin RF_READ = 1; RF_WRITE = 0; RF_ADDR_R1 = rs; RF_ADDR_R2 = rt; end // add
        6'h22: begin RF_READ = 1; RF_WRITE = 0; RF_ADDR_R1 = rs; RF_ADDR_R2 = rt; end  // sub
        6'h2c: begin RF_READ = 1; RF_WRITE = 0; RF_ADDR_R1 = rs; RF_ADDR_R2 = rt; end  // mul
        6'h24: begin RF_READ = 1; RF_WRITE = 0; RF_ADDR_R1 = rs; RF_ADDR_R2 = rt; end  // and
        6'h25: begin RF_READ = 1; RF_WRITE = 0; RF_ADDR_R1 = rs; RF_ADDR_R2 = rt; end  // or
        6'h27: begin RF_READ = 1; RF_WRITE = 0; RF_ADDR_R1 = rs; RF_ADDR_R2 = rt; end // nor
        6'h2a: begin RF_READ = 1; RF_WRITE = 0; RF_ADDR_R1 = rs; RF_ADDR_R2 = rt; end  // slt
        6'h01: begin RF_READ = 1; RF_WRITE = 0; RF_ADDR_R1 = rs; end  // sll
        6'h02: begin RF_READ = 1; RF_WRITE = 0; RF_ADDR_R1 = rs; end  // srl
        6'h08: begin RF_READ = 1; RF_WRITE = 0; RF_ADDR_R1 = rs; end  // jr
    endcase
end
// I-type
6'h08 :  begin RF_READ = 1; RF_WRITE = 0; RF_ADDR_R1 = rs; end  // addi
6'h1d :  begin RF_READ = 1; RF_WRITE = 0; RF_ADDR_R1 = rs; end // multi
6'h0c :  begin RF_READ = 1; RF_WRITE = 0; RF_ADDR_R1 = rs; end // andi
6'h0d :  begin RF_READ = 1; RF_WRITE = 0; RF_ADDR_R1 = rs; end // ori
6'h0f :  ; // lui -- do nothing
6'h0a :  begin RF_READ = 1; RF_WRITE = 0; RF_ADDR_R1 = rs; end // slti
6'h04 :  begin RF_READ = 1; RF_WRITE = 0; RF_ADDR_R1 = rs; RF_ADDR_R2 = rt; end // beq
6'h05 :  begin RF_READ = 1; RF_WRITE = 0; RF_ADDR_R1 = rs; RF_ADDR_R2 = rt; end // bne
6'h23 :  begin RF_READ = 1; RF_WRITE = 0; RF_ADDR_R1 = rs; end // lw
6'h2b :  begin RF_READ = 1; RF_WRITE = 0; RF_ADDR_R1 = rs; RF_ADDR_R2 = rt; end // sw
// J-Type
6'h02 : ; // jmp -- do nothing
6'h03 : ; // jal -- do nothing
6'h1b : begin RF_READ = 1; RF_WRITE = 0; RF_ADDR_R1 = 0; end // push
6'h1c : ; // pop -- do nothing

endcase
end
```

Execution (EXE): The main function of the operation is performed unless it is memory access or updating the PC register. Essentially, any function requiring the ALU happens here. Pass the OPRN code and the values to OP1 and OP2.

```
if (proc_state === `PROC_EXE)
begin
case(opcode)
// R-Type
6'h00 : begin
    case(funct)
        6'h20:  begin ALU_OPRN = `ALU_OPRN_WIDTH'h01; ALU_OP1 = RF_DATA_R1; ALU_OP2 = RF_DATA_R2; end // add
        6'h22:  begin ALU_OPRN = `ALU_OPRN_WIDTH'h02; ALU_OP1 = RF_DATA_R1; ALU_OP2 = RF_DATA_R2; end // sub
        6'h2c:  begin ALU_OPRN = `ALU_OPRN_WIDTH'h03; ALU_OP1 = RF_DATA_R1; ALU_OP2 = RF_DATA_R2; end // mul
        6'h24:  begin ALU_OPRN = `ALU_OPRN_WIDTH'h06; ALU_OP1 = RF_DATA_R1; ALU_OP2 = RF_DATA_R2; end // and
        6'h25:  begin ALU_OPRN = `ALU_OPRN_WIDTH'h07; ALU_OP1 = RF_DATA_R1; ALU_OP2 = RF_DATA_R2; end // or
        6'h27:  begin ALU_OPRN = `ALU_OPRN_WIDTH'h08; ALU_OP1 = RF_DATA_R1; ALU_OP2 = RF_DATA_R2; end // nor
        6'h2a:  begin ALU_OPRN = `ALU_OPRN_WIDTH'h09; ALU_OP1 = RF_DATA_R1; ALU_OP2 = RF_DATA_R2; end // slt
        6'h01:  begin ALU_OPRN = `ALU_OPRN_WIDTH'h05; ALU_OP1 = RF_DATA_R1; ALU_OP2 = shamt; end // sll
        6'h02:  begin ALU_OPRN = `ALU_OPRN_WIDTH'h04; ALU_OP1 = RF_DATA_R1; ALU_OP2 = shamt; end // srl
        6'h08:  ; // jr -- do nothing
    endcase
end
// I-type
6'h08 :  begin ALU_OPRN = `ALU_OPRN_WIDTH'h01; ALU_OP1 = RF_DATA_R1; ALU_OP2 = signExtImm; end // addi
6'h1d :  begin ALU_OPRN = `ALU_OPRN_WIDTH'h03; ALU_OP1 = RF_DATA_R1; ALU_OP2 = signExtImm; end // multi
6'h0c :  begin ALU_OPRN = `ALU_OPRN_WIDTH'h06; ALU_OP1 = RF_DATA_R1; ALU_OP2 = zeroExtImm; end // andi
6'h0d :  begin ALU_OPRN = `ALU_OPRN_WIDTH'h07; ALU_OP1 = RF_DATA_R1; ALU_OP2 = zeroExtImm; end // ori
6'h0f :  ; // lui -- do nothing
6'h0a :  begin ALU_OPRN = `ALU_OPRN_WIDTH'h09; ALU_OP1 = RF_DATA_R1; ALU_OP2 = signExtImm; end // slti
6'h04 :  begin ALU_OPRN = `ALU_OPRN_WIDTH'h02; ALU_OP1 = RF_DATA_R1; ALU_OP2 = RF_DATA_R2; end // beq
6'h05 :  begin ALU_OPRN = `ALU_OPRN_WIDTH'h02; ALU_OP1 = RF_DATA_R1; ALU_OP2 = RF_DATA_R2; end // bne
6'h23 :  begin ALU_OPRN = `ALU_OPRN_WIDTH'h01; ALU_OP1 = RF_DATA_R1; ALU_OP2 = signExtImm; end // lw
6'h2b :  begin ALU_OPRN = `ALU_OPRN_WIDTH'h01; ALU_OP1 = RF_DATA_R1; ALU_OP2 = signExtImm; end // sw
// J-Type
6'h02 : ; // jmp -- do nothing
6'h03 : ; // jal -- do nothing
6'h1b : ; // push -- do nothing
6'h1c : ; // pop -- do nothing
endcase
end
```

Memory access (MEM): Memory is read from or written to. Sw, lw, pop and push are the only operations that access the memory. All other functions should have their memory access sections set a hold condition to the memory unit (read = 0, write = 0).

```
if (proc_state === `PROC_MEM)
begin
case(opcode)
// R-Type
6'h00 : begin
    case(funct)
        6'h20:  begin MEM_READ = 0; MEM_WRITE = 0; end // add
        6'h22:  begin MEM_READ = 0; MEM_WRITE = 0; end // sub
        6'h2c:  begin MEM_READ = 0; MEM_WRITE = 0; end // mul
        6'h24:  begin MEM_READ = 0; MEM_WRITE = 0; end // and
        6'h25:  begin MEM_READ = 0; MEM_WRITE = 0; end // or
        6'h27:  begin MEM_READ = 0; MEM_WRITE = 0; end // nor
        6'h2a:  begin MEM_READ = 0; MEM_WRITE = 0; end // slt
        6'h01:  begin MEM_READ = 0; MEM_WRITE = 0; end // sll
        6'h02:  begin MEM_READ = 0; MEM_WRITE = 0; end //srl
        6'h08:  begin MEM_READ = 0; MEM_WRITE = 0; end // jr
    endcase
end
// I-type
6'h08 :  begin MEM_READ = 0; MEM_WRITE = 0; end // addi
6'h1d :  begin MEM_READ = 0; MEM_WRITE = 0; end // multi
6'h0c :  begin MEM_READ = 0; MEM_WRITE = 0; end // andi
6'h0d :  begin MEM_READ = 0; MEM_WRITE = 0; end // ori
6'h0f :  begin MEM_READ = 0; MEM_WRITE = 0; end // lui
6'h0a :  begin MEM_READ = 0; MEM_WRITE = 0; end // slti
6'h04 :  begin MEM_READ = 0; MEM_WRITE = 0; end // beq
6'h05 :  begin MEM_READ = 0; MEM_WRITE = 0; end // bne
6'h23 :  begin MEM_READ = 1; MEM_WRITE = 0; MEM_ADDR = ALU_RESULT; end // lw
6'h2b :  begin MEM_READ = 0; MEM_WRITE = 1; MEM_ADDR = ALU_RESULT; write_data = RF_DATA_R2; end // sw
// J-Type
6'h02 : begin MEM_READ = 0; MEM_WRITE = 0; end // jmp
6'h03 : begin MEM_READ = 0; MEM_WRITE = 0; end // jal
6'h1b : begin MEM_READ = 0; MEM_WRITE = 1; MEM_ADDR = SP_REF; write_data = RF_DATA_R1; SP_REF = SP_REF - 1; end // push
6'h1c : begin MEM_READ = 1; MEM_WRITE = 0; SP_REF = SP_REF + 1; MEM_ADDR = SP_REF; end // pop
endcase
end
```

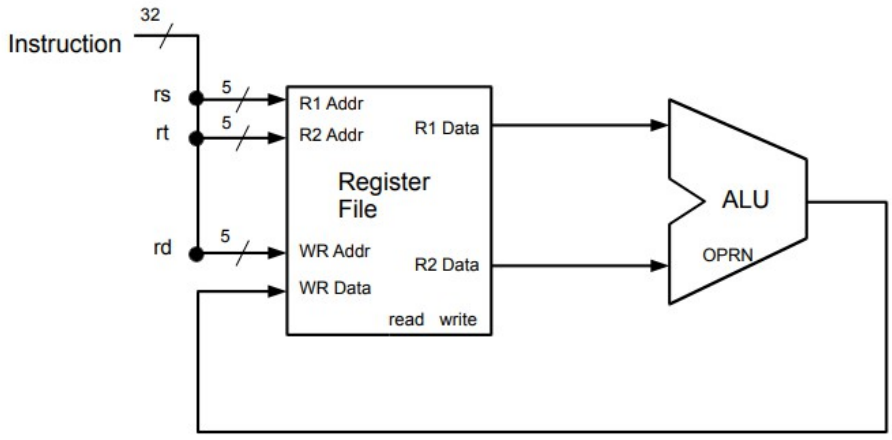
Write Back (WB): Memory is set to hold, register results (from alu operations) are written back to registers rd or rt (rt for any of the immediate functions and load word), the PC register is incremented or written back to.

```
if (proc_state === `PROC_WB)
begin
case(opcode)
// R-Type
6'h00 : begin
    case(funct)
        6'h20: begin RF_READ = 0; RF_WRITE = 1; RF_ADDR_W = rd; RF_DATA_W = ALU_RESULT;
               PC_REG = PC_REG + 1; MEM_READ = 0; MEM_WRITE = 0; end // add
        6'h22: begin RF_READ = 0; RF_WRITE = 1; RF_ADDR_W = rd; RF_DATA_W = ALU_RESULT;
               PC_REG = PC_REG + 1; MEM_READ = 0; MEM_WRITE = 0; end // sub
        6'h2c: begin RF_READ = 0; RF_WRITE = 1; RF_ADDR_W = rd; RF_DATA_W = ALU_RESULT;
               PC_REG = PC_REG + 1; MEM_READ = 0; MEM_WRITE = 0; end // mul
        6'h24: begin RF_READ = 0; RF_WRITE = 1; RF_ADDR_W = rd; RF_DATA_W = ALU_RESULT;
               PC_REG = PC_REG + 1; MEM_READ = 0; MEM_WRITE = 0; end // and
        6'h25: begin RF_READ = 0; RF_WRITE = 1; RF_ADDR_W = rd; RF_DATA_W = ALU_RESULT;
               PC_REG = PC_REG + 1; MEM_READ = 0; MEM_WRITE = 0; end // or
        6'h27: begin RF_READ = 0; RF_WRITE = 1; RF_ADDR_W = rd; RF_DATA_W = ALU_RESULT;
               PC_REG = PC_REG + 1; MEM_READ = 0; MEM_WRITE = 0; end // nor
        6'h2a: begin RF_READ = 0; RF_WRITE = 1; RF_ADDR_W = rd; RF_DATA_W = ALU_RESULT;
               PC_REG = PC_REG + 1; MEM_READ = 0; MEM_WRITE = 0; end // slt
        6'h01: begin RF_READ = 0; RF_WRITE = 1; RF_ADDR_W = rd; RF_DATA_W = ALU_RESULT;
               PC_REG = PC_REG + 1; MEM_READ = 0; MEM_WRITE = 0; end // sll
        6'h02: begin RF_READ = 0; RF_WRITE = 1; RF_ADDR_W = rd; RF_DATA_W = ALU_RESULT;
               PC_REG = PC_REG + 1; MEM_READ = 0; MEM_WRITE = 0; end //srl
        6'h08: begin RF_READ = 0; PC_REG = RF_DATA_R1; MEM_READ = 0; MEM_WRITE = 0; end // jr
    endcase
```

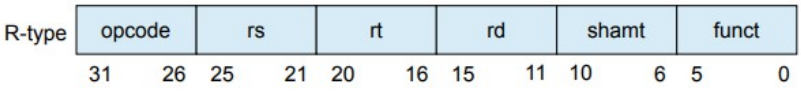
```
// I-type
6'h08 : begin RF_READ = 0; RF_WRITE = 1; RF_ADDR_W = rt; RF_DATA_W = ALU_RESULT;
              PC_REG = PC_REG + 1; MEM_READ = 0; MEM_WRITE = 0; end // addi
6'h1d : begin RF_READ = 0; RF_WRITE = 1; RF_ADDR_W = rt; RF_DATA_W = ALU_RESULT;
              PC_REG = PC_REG + 1; MEM_READ = 0; MEM_WRITE = 0; end // multi
6'h0c : begin RF_READ = 0; RF_WRITE = 1; RF_ADDR_W = rt; RF_DATA_W = ALU_RESULT;
              PC_REG = PC_REG + 1; MEM_READ = 0; MEM_WRITE = 0; end // andi
6'h0d : begin RF_READ = 0; RF_WRITE = 1; RF_ADDR_W = rt; RF_DATA_W = ALU_RESULT;
              PC_REG = PC_REG + 1; MEM_READ = 0; MEM_WRITE = 0; end // ori
6'h0f : begin RF_READ = 0; RF_WRITE = 1; RF_ADDR_W = rt; RF_DATA_W = {immediate, {16{1'b0}}};
              PC_REG = PC_REG + 1; MEM_READ = 0; MEM_WRITE = 0; end // lui
6'h0a : begin RF_READ = 0; RF_WRITE = 1; RF_ADDR_W = rt; RF_DATA_W = ALU_RESULT;
              PC_REG = PC_REG + 1; MEM_READ = 0; MEM_WRITE = 0; end // slti
6'h04 : begin if (ZERO == 1) PC_REG = branchAddress; // if rs=rt then rs-rt == 0
              MEM_READ = 0; MEM_WRITE = 0; end // beq
6'h05 : begin if (ZERO == 0) PC_REG = branchAddress; // if rs=rt then rs-rt == 0
              MEM_READ = 0; MEM_WRITE = 0; end // bne
6'h23 : begin RF_READ = 0; RF_WRITE = 1; RF_ADDR_W = rt; RF_DATA_W = ALU_RESULT;
              PC_REG = PC_REG + 1; MEM_READ = 0; MEM_WRITE = 0; end  // lw
6'h2b : begin MEM_READ = 0; MEM_WRITE = 1; MEM_ADDR = ALU_RESULT; write_data = RF_DATA_R1; end  // sw
// J-Type
6'h02 : begin PC_REG = {6'b0, address}; MEM_READ = 0; MEM_WRITE = 0; end // jmp
6'h03 : begin RF_READ = 0; RF_WRITE = 1; RF_ADDR_W = 31; RF_DATA_W = PC_REG + 1;
              PC_REG = {6'b0, address}; MEM_READ = 0; MEM_WRITE = 0; end // jal
6'h1b : begin MEM_READ = 0; MEM_WRITE = 0; end // push
6'h1c : begin RF_READ = 0; RF_WRITE = 1; RF_ADDR_W = 0; RF_DATA_W = MEM_DATA; end// pop
endcase
end
```

Following the above code, there may be some errors as I haven't tested my commands yet, you can see the data paths from the following images: **(All subsequent images are from Lecture 11 of Kaushik Patra's CS147 class, the page/slide number of the lectures are on the bottom right of the images.)**

General R-type Instruction Data Path

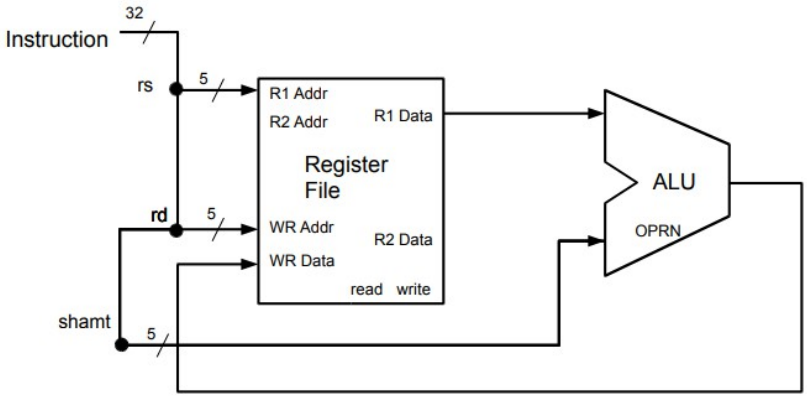


Operation: R[rd] = R[rs] (op) R[rt]

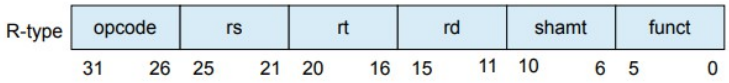


Once the instruction has been decoded (ID), the rs and rt can fetched from the register file (RF), then r1 and r2 can be passed to the ALU (EXE), and the resultant is returned to the RF (WB) and written to rd.

Shift Instruction Data Path



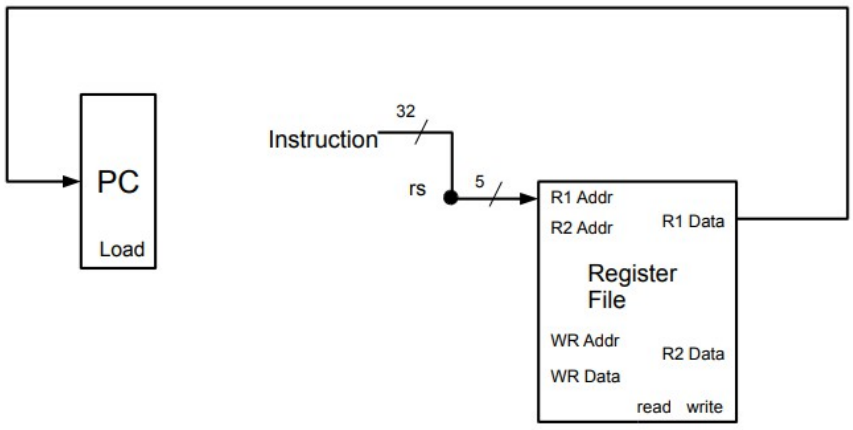
Operation: R[rd] = R[rs] (op) shamt



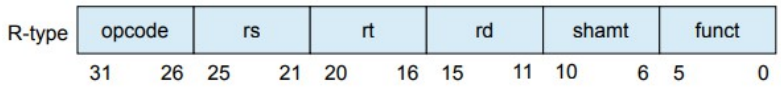
16

Essentially the same as before but in the decode step shamt will be op2 (or rs as op2 and shamt at op1, implementation here doesn't matter so long as it's properly controlled that shamt is shamt and not the data to be shifted).

Jump Reg Instruction Data Path

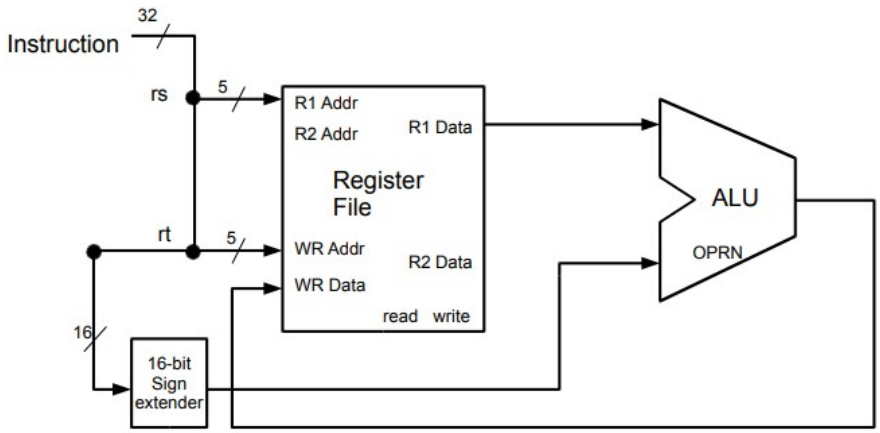


Operation: PC = R[rs]

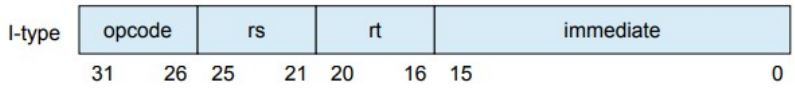


Jump register only has the instruction decoding (ID), register fetch, then write back the retrieved register data into PC.

Arithmetic I-type Instruction Data Path

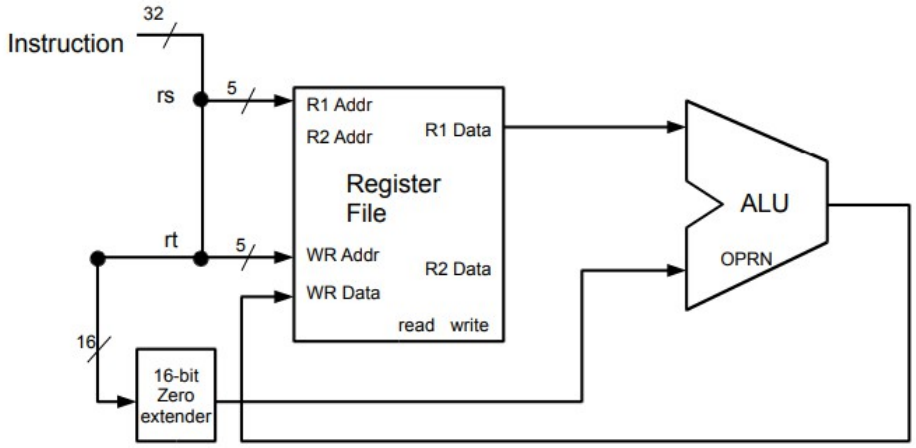


Operation: R[rt] = R[rs] (op) SignExtImm

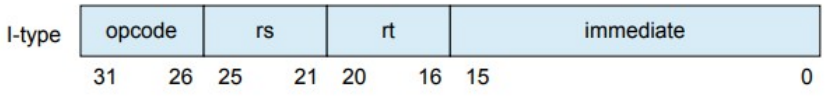


This data path is very similar to the arithmetic R type instructions, the decode, fetch, execute, etc phases are all the same. The only difference is that the second operation for the ALU is an immediate value that is extended. For sign extension, the most significant bit is padded onto the leading 16 bits followed by the inputted immediate value. (signExtImm = {{16{immediate[15]}}, immediate};

Logical I-type Instruction Data Path

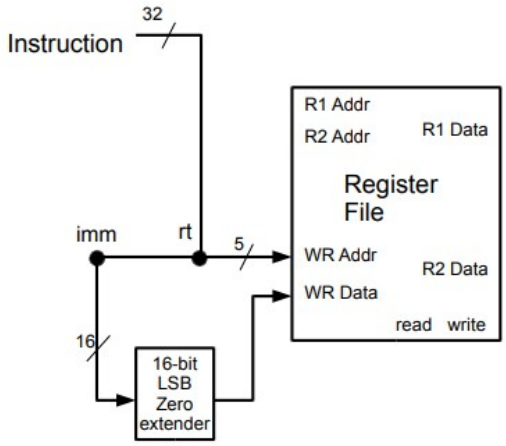


Operation: R[rt] = R[rs] (op) ZeroExtImm

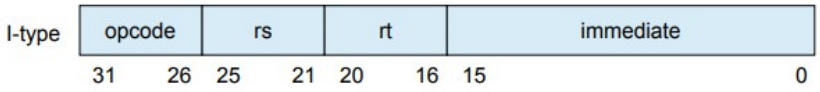


Once again, essentially the same as before except zero extender pads the leading 16 bits with 0's. (zeroExtImm = {{16{1'b0}}, immediate};)

LUI Instruction Data Path

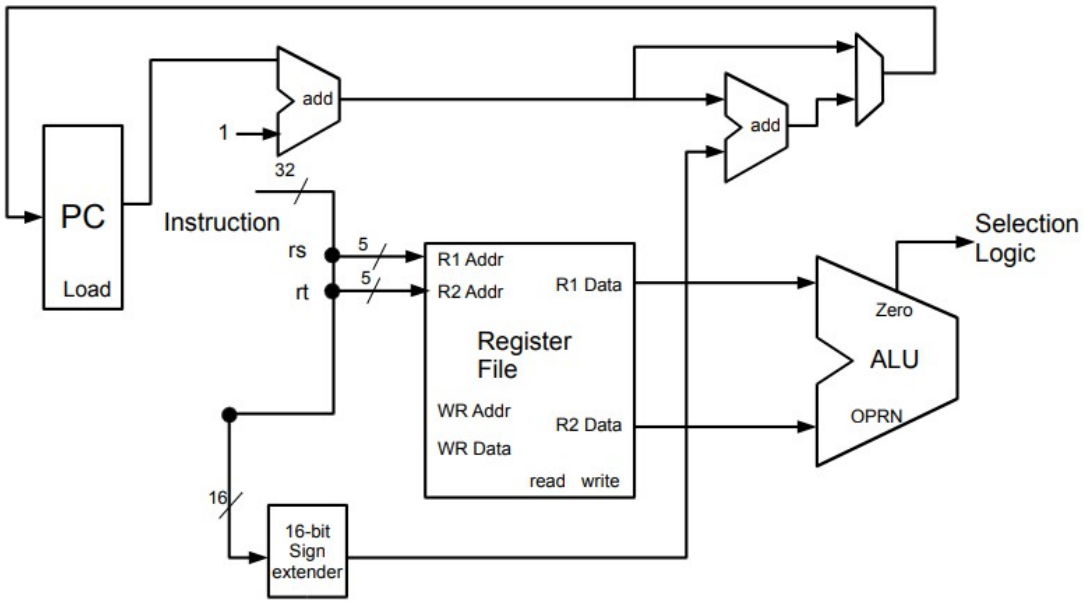


Operation: R[rt] = {imm, 16b'0}

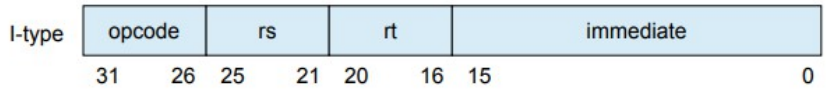


LUI simply decodes the instruction, extendes the immediate value's lower 16 bits with 0's, and then writes the extended immediate into the register with rt's address (WB).

Branch I-type Instruction Data Path

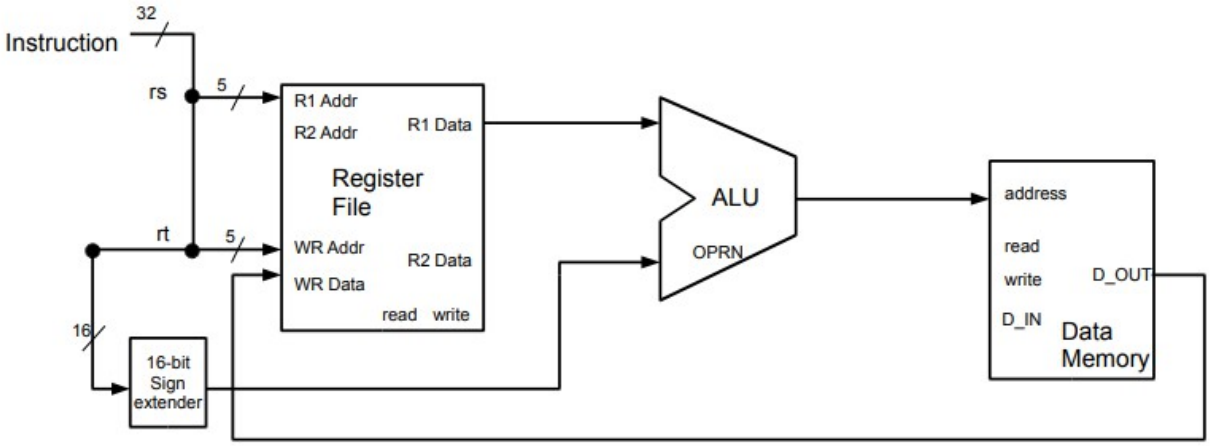


Operation: PC = PC + 1 + SignExtImm;
if R[rs] == R[rt] or R[rs] != R[rt]

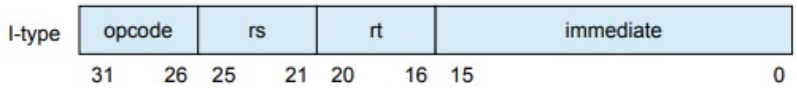


Rs and rt are compared (if rs-rt==0 then rs=rt) and based on if its beq or bne, then pc is set to PC+ 1 +SignExtImm. Fetch rs and rt during decode/fetch, compare them in the execution phase, check if zero == 1|0, then in write back change pc accordingly.

LW Instruction Data Path



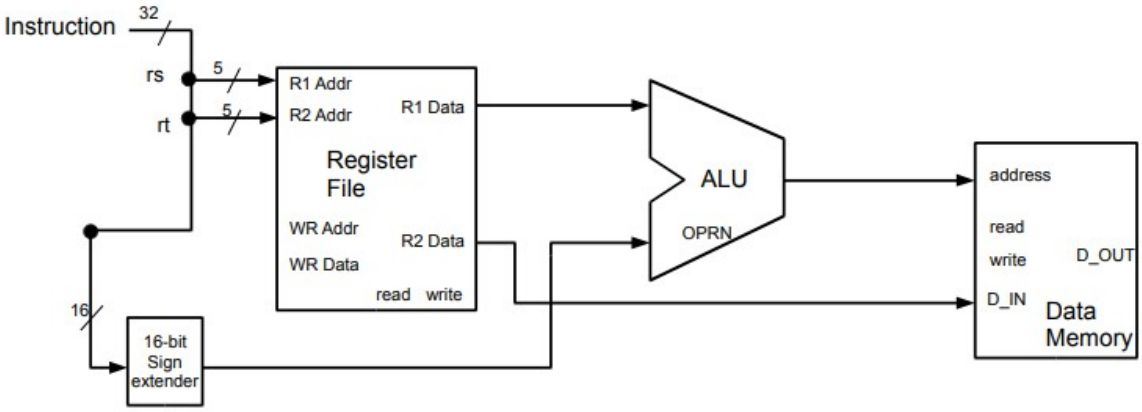
Operation: R[rt] = M[R[rs] + SignExtImm]



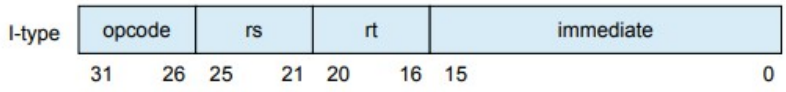
25

Decode as usual, fetch rs from RF, compute SignExtImm in decode, compute R[rs]+SignExtImm in EXE (pass to alu), then use the resultant as the address to fetch data from memory (MEM), and write back the fetched data into rt. Basically, you're loading some memory into a register.

SW Instruction Data Path



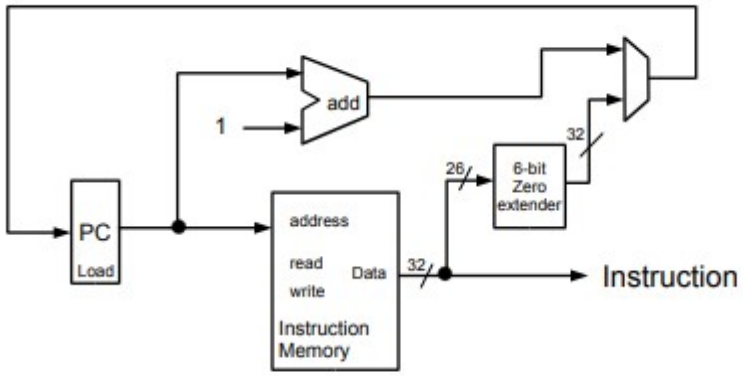
Operation: M[R[rs] + SignExtImm] = R[rt]



26

Essentially the reverse of LW: Fetch rt and rs during fetch while also computing SignExtImm. Then during EXE, compute R[rs]+SignExtImm, and write the data from rt's address (from the RF) into the resulting address from the ALU as the address to the memory file (WB). Basically, you're writing a register into memory.

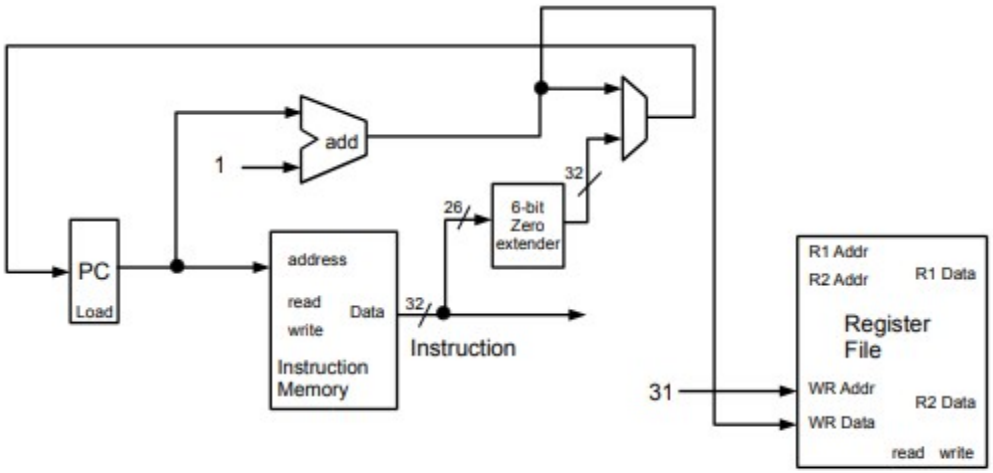
JMP Instruction Data Path



Operation: PC = {6'b0, address}

Jmp's data path's multiplexer is handled/implemented in that all other instructions increment the PC by 1. Jump simply extends the bits and writes back the extended address to PC (WB). Since I am checking if the instruction inputted is jmp, you don't need to have any other option than simply setting pc to the address.

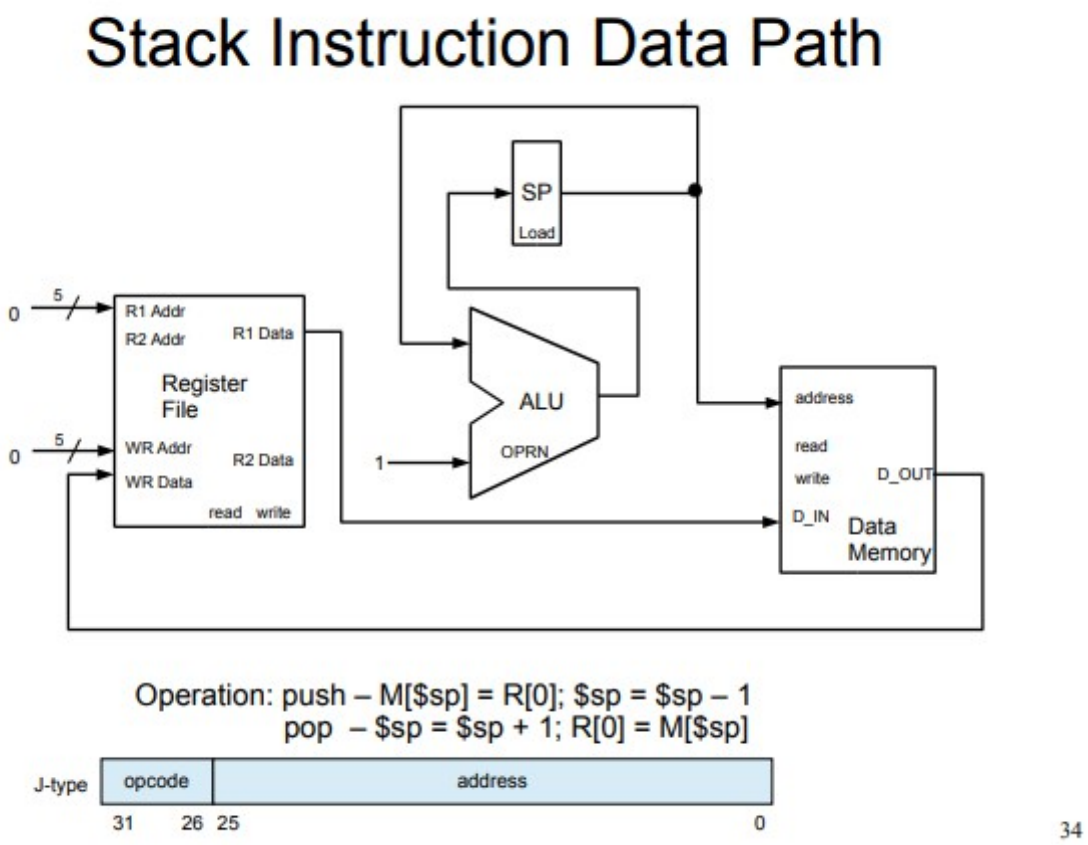
JAL Instruction Data Path



Operation: R[31] = PC + 1; PC = {6'b0, address}

32

Jal simply computes PC+1 during exe and during decode you extend the bits (or for verilog you can just extend at the WB phase but in theory this should happen during decode). During write back you set R[31] and PC respectively.



For push, since you can't compute sp-1 after write back, do sp=sp-1 during write back. Fetch R[0], use sp as the write address into memory and R[0] as the write data (WB) and decrement sp.
For pop, you can compute sp=sp+1 during EXE as you need to use it before write back, pull m[sp] during MEM, then write it back to R[0].

VI. Entire System Testing Strategy

(My system has yet to be tested so I will be outlining the theory.) For every phase for each command you would compute what every instruction's control signals should be and then convert it all into hex. Then pass it into the da_vinci_tb.v. You can then read out the results within RevFib_mem_dump.dat. Essentially, if you get back all 0's your test has failed.
Code from da_vinci_tb.v provided by Kaushik Patra.

```
#5000  $writememh("RevFib_mem_dump.dat", da_vinci_inst.memory_inst.sram_32x64m, 'h03fffff0, 'h03ffffff);
       //$writememh("fibonacci_mem_dump.dat", da_vinci_inst.memory_inst.sram_32x64m, 'h01000000, 'h0100000f);
       $stop;
```

Since my system is untested as of typing this, follow the data paths instead and test through the da_vinci_tb.v file. (My file returns:
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000

00000000
00000000
00000000
00000000
00000000
00000000
00000000

So I know there is an issue somewhere; but, due to lack of time, I will not be correcting it for this report. Since all other modules work, the issue must be within my control_unit.v file.

VII. Conclusion

The DaVinci 1.0 system supports the CS147DV, has an ALU, Register File, 64MB of memory, a state machine, a PC register, and supports the instruction cycle. Everything has been tested and works properly (except the control unit but it's likely a minor issue and not an issue with the data path). A basic microprocessor has been constructed and been detailed within the report.