DaVinci v1.0: Computer System Behavioral Model

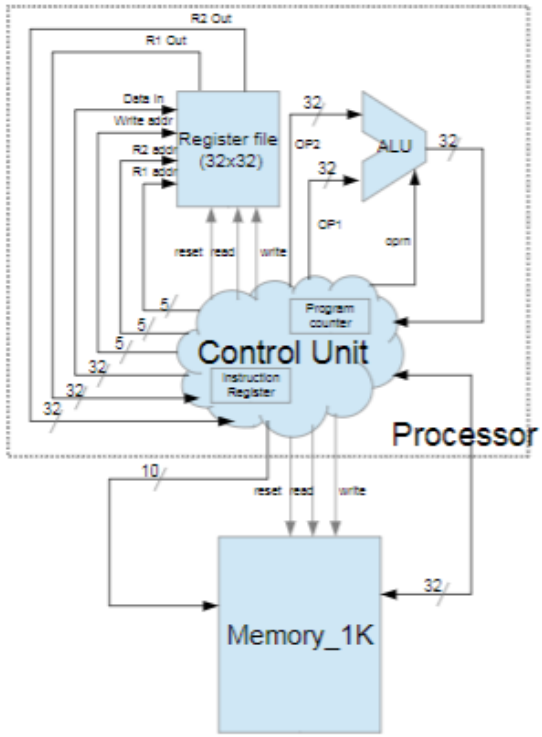
CS147 Project 2: Combining the ALU with Memory and Register File

Ryan Tran
College of Science: Computer Science
San José State University
San José, United States
tranryanp@gmail.com

***Abstract*—This report explains the creation of the DaVinci v1.0 computer system using Verilog. The DaVinci v1.0 system supports the 'CS147DV' instruction set, while also containing memory models, and a processor. The following sections will cover each component individually and then later demonstrating how all the components work together to simulate a working DaVinci v1.0 system. This system requires setup and knowledge from the first CS147 project.**

I. Introduction

This project is the second assignment of CS147. The objective is to implement a behavioral model of a computer system named DaVinci v1.0. This system is composed of a 32-bit processor and 256MB memory created in Verilog. Within a processor also contains an Arithmetic Logic Unit (ALU), Control Unit, Register File, and memory models. DaVinci v1.0 will demonstrate how all components of a processor work together alongside memory. Shown below is a diagram of the completed project:



The results of this project are dependent on the first CS147 project in which ModelSim was setup and the understanding of using the application, such as using the Wave forms to test results, are necessary and will not be re-included in this project report. This report contains the requirements, designs, and test results of such tasks.

II. Project Requirements

Understanding and having completed the setup of the first project is necessary in proceeding with this project. Without it, knowing how to use the waves to troubleshoot and working on the project will be extremely difficult, if not possible at all. The files that need to be completed are:

1) alu.v
2) register_file.v
3) control_unit.v
4) testbench files for each of the above

All other code has already been provided in order to avoid tedious work of connecting ports and wires.

III. Implementation

A. Arithmatic Logic Unit (ALU)

Included code segment from the first project that is needed is the ALU. This component will be able to complete nine operations in the 'CS147DV' instruction set.

1) Addition
2) Subtraction
3) Multiplication
4) Shift right
5) Shift left
6) Bitwise AND
7) Bitwise OR
8) Bitwise NOR
9) Set less than

Aside from editing port names to match the new port names existing in this project, nearly everything is the same. However, included with this ALU versus the one implemented in the first project is the 'ZERO' value. When the result of the ALU operation is zero, the 'ZERO' value will be set to 1. If the result of the ALU is anything but zero, 'ZERO' is set as zero. The

implementation of the new ALU with the ‘ZERO’ output is shown below:

```
// simulator internal storage for results
reg [`DATA_INDEX_LIMIT:0] OUT;
reg ZERO;

always @(OP1 or OP2 or OPRN)
begin
    case (OPRN) // edited operation codes to match 'CS147' instruction set
        `ALU_OPRN_WIDTH'h20 : OUT = OP1 + OP2; // addition
        `ALU_OPRN_WIDTH'h22 : OUT = OP1 - OP2; // subtraction
        `ALU_OPRN_WIDTH'h2c : OUT = OP1 * OP2; // multiplication
        `ALU_OPRN_WIDTH'h02 : OUT = OP1 >> OP2; // shift right
        `ALU_OPRN_WIDTH'h01 : OUT = OP1 << OP2; // shift left
        `ALU_OPRN_WIDTH'h24 : OUT = OP1 & OP2; // bitwise and
        `ALU_OPRN_WIDTH'h25 : OUT = OP1 | OP2; // bitwise or
        `ALU_OPRN_WIDTH'h27 : OUT = ~(OP1 | OP2); // bitwise nor
        `ALU_OPRN_WIDTH'h2a : OUT = OP1 < OP2; // set less than
        default: OUT = `DATA_WIDTH'hxxxxxxxx; // default

    endcase
        ZERO = OUT == 0 ? 1 : 0;
end

endmodule
```

The first line with the comment “simulator internal storage for results” initializes two registers ‘OUT’ and ‘ZERO’ in order to store results for each respective value. Without these registers, it wouldn’t be possible to store the result of the ALU operation nor the ‘ZERO’ output immediately following the operation.

Next, aside from the result register now named ‘OUT’, it can be seen that the ALU operations being performed are nearly identical to that of the first project’s, except that the hexadecimal value for the operations has been changed. The reason for this change is that the new values are the ‘funct’ codes of the corresponding ‘CS147 Instruction Set’ instructions, improving continuity.

Lastly, it can be seen at the second-to-last line of the code that the ‘ZERO’ output is being set. As detailed earlier, the value that ‘ZERO’ will have is either 1 or 0. By this simplistic definition, it can be seen that its value can be set with a conditional. Rather than using an if-then-else statement, we can instead use a conditional operator (?:) in order to save space and be concise.

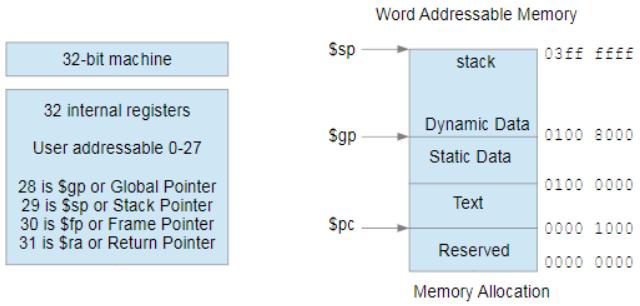
With that, the new ‘alu.v’ file is complete. It must be tested for similar functionality as the first project’s ALU, but also for the new ‘ZERO’ output. A testbench for the newly completed ALU can be repurposed from the first project’s testbench. Editing it to include and test the ‘ZERO’ output, the same file can be reused to test the DaVinci v1.0’s ALU.

```
`include "prj_definition.v"
module ALU_TB;

integer total_test;
integer pass_test;

reg [`ALU_OPRN_INDEX_LIMIT:0] oprn_reg;
reg [`DATA_INDEX_LIMIT:0] op1_reg;
reg [`DATA_INDEX_LIMIT:0] op2_reg;

wire [`DATA_INDEX_LIMIT:0] r_net;

// added wire for ZERO
wire ZERO;

// Instantiation of ALU
ALU ALU_INST_01(.OUT(r_net), .ZERO(ZERO), .OP1(op1_reg), // ZERO added
                .OP2(op2_reg), .OPRN(oprn_reg)); // input and reg names edited to match names in 'alu.v'
```

The code shown is near identical to the code seen in project one. Aside from the name of the module being changed, the noteworthy difference is that a wire named ‘ZERO’ is added, along with including the ‘ZERO’ wire in the ALU’s instantiation. Additionally, the instantiation of the ALU begins with full uppercase ‘ALU’ to match the name of the ‘alu.v’ module. After that, the rest of the testbench code, including the test cases used in project one, are changed to support the new hexadecimal operation value. Testing results are shown in the “Testing” section later in the report.

B. Memory

For DaVinci v1.0, memory is a 32x64M memory space. Though it has already been completed for us, it is provided as reference because its model and structure is vital for the register to perform as necessary due to the memory and register having very similar characteristics. A diagram of the implemented memory is as shown:



C. Register File

Having a 32x32 register file model, it is necessary to properly implement a model that will be able to create and access each of the 32 doubleword-addressible registers. Shown below is the implemented code to complete the register file:

```
// simulator internal storage for data
reg [`DATA_INDEX_LIMIT:0] addr_return1;
reg [`DATA_INDEX_LIMIT:0] addr_return2;

// default case: return z if no read
assign DATA_R1 = (READ === 1'b1 && WRITE === 1'b0) ? addr_return1 : {`DATA_WIDTH{1'bz}};
assign DATA_R2 = (READ === 1'b1 && WRITE === 1'b0) ? addr_return2 : {`DATA_WIDTH{1'bz}};

// 32x32 memory storage
reg [`DATA_INDEX_LIMIT:0] reg_32x32 [0:`REG_INDEX_LIMIT];

// initial value for reset
integer i;

// sets all registers as 0
initial
begin
        for(i = 0; i <= `REG_INDEX_LIMIT; i = i + 1)
            reg_32x32[i]={ `DATA_WIDTH{1'b0} };
end

always @ (negedge RST or posedge CLK)
begin
        if (RST === 1'b0) // reset on negative edge of RST
                for (i = 0; i <= `REG_INDEX_LIMIT; i = i + 1)
                        reg_32x32[i] = { `DATA_WIDTH{1'b0} };
        else
        begin
                if (READ === 1'b1 && WRITE === 1'b0) // if read
                begin
                        addr_return1 = reg_32x32[ADDR_R1];      // return content of address ADDR_R1
                        addr_return2 = reg_32x32[ADDR_R2];      // and address ADDR_R2
                end
                else if (READ === 1'b0 && WRITE === 1'b1)       // else if write
                        reg_32x32[ADDR_W] = DATA_W;     // place DATA_W in address ADDR_W
        end // only handles read or write; doesnt handle read, write = 00 or 11

end
endmodule
```

Once again, similar to the ALU, simulator internal storage is created, but for data this time. These internal storages ‘addr_return1’, and ‘addr_return2’ will be used to contain data from specific registers. These two internal storages will then be assigned to ‘DATA_R1’ and ‘DATA_R2’ to be returned. If the operation is not a read, ‘z’ will be returned. After that, another storage is initialized called ‘reg_32x32’. This storage is additionally given “[0:`REG_INDEX_LIMIT]”, which allows it to have 0 up to the register limit. In the case of this file, ‘reg_32x32’ will have 0 up to 31 registers due to the limit being 31. This information is found in the ‘prj_defintion.v’ file and is as follows:

```
`define NUM_OF_REG 32
`define REG_INDEX_LIMIT (`NUM_OF_REG -1)
```

Next, an integer ‘i’ will be used as an index for the for-loop immediately following. Every register in ‘reg_32x32’ is set as 0 as the initial register before any modification. Following the similar code from the memory file, ‘reg_32x32’ is looped through and initialized with 0’s. When the RST signal is on the negative edge, then the register block needs to be reset. This can be seen in the “always @ (negedge RST or posedge CLK)” block in the “if (RST == 1’b0)” statement that the entire register block is reset to 0, another snippet of code near identical to the code in memory. Otherwise, we check if a ‘read’ or ‘write’ operation is being performed. If it is, then we return data if it is ‘read’ or change a register’s data if it is ‘write’. Cases where read = 00 or write = 11 are not handled, allowing the register file to hold previously read data. Once again, this code is based on the memory file’s code.

With this, ‘register_file.v’ has been completed with a similar structure to ‘memory.v’. Now it must be tested with a testbench file similar to the memory testbench where data is written is read back to confirm that the register file works. Similar to the testing of the ALU, we are able to repurpose the testbench file for ‘memory.v’, editing it to handle more address registers.

```
module REGISTER_FILE_TB;
// Storage list
reg [`REG_ADDR_INDEX_LIMIT:0] ADDR_W, ADDR_R1, ADDR_R2;
// reset
reg READ, WRITE, RST;
// data register
reg [`DATA_INDEX_LIMIT:0] DATA_W;
integer i; // index for memory operation
integer no_of_test, no_of_pass;

// wire lists
wire  CLK;
wire [`DATA_INDEX_LIMIT:0] DATA_R1, DATA_R2;

// Clock generator instance
CLK_GENERATOR clk_gen_inst(.CLK(CLK));

// 32x32 register file instance
REGISTER_FILE reg_inst(.DATA_R1(DATA_R1), .DATA_R2(DATA_R2), .ADDR_R1(ADDR_R1), .ADDR_R2(ADDR_R2), .DATA_W(DATA_W),
                       .ADDR_W(ADDR_W), .READ(READ), .WRITE(WRITE), .CLK(CLK), .RST(RST));
```

The start of the testbench begins by adding additional address registers, rather than the memory testbench’s single register. Other registers are renamed to have the same names as ones in the register file for continuity and ease of understanding.

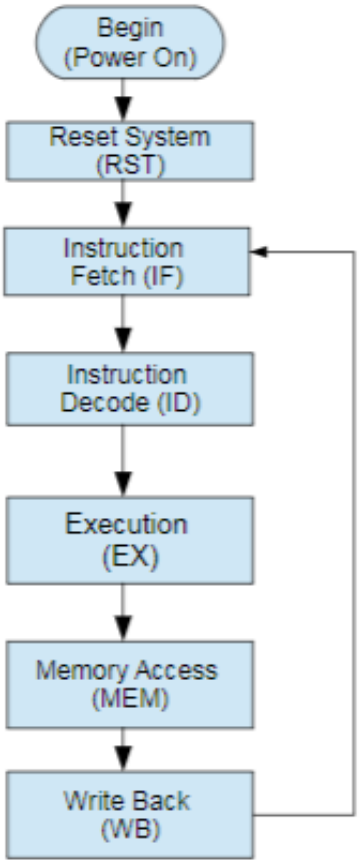
```
initial
begin
RST=1'b1;
READ=1'b0;
WRITE=1'b0;
DATA_W = {`DATA_WIDTH{1'b0} };
no_of_test = 0;
no_of_pass = 0;

// Start the operation
#10    RST=1'b0;
#10    RST=1'b1;
// Write cycle
for (i = 0; i <= `REG_INDEX_LIMIT; i = i + 1)
begin
#10      DATA_W = i; READ = 1'b0; WRITE = 1'b1; ADDR_W = i;
end

// Read Cycle
#10   READ=1'b0; WRITE=1'b0;
#5    no_of_test = no_of_test + 1;
      if (DATA_R1 !== {`DATA_WIDTH{1'bz}} || DATA_R2 !== {`DATA_WIDTH{1'bz}})
        $write("[TEST] Read %1b, Write %1b, expecting 32'hzzzzzzzz, got %8h and %8h [FAILED]\n", READ, WRITE, DATA_R1, DATA_R2);
      else
        no_of_pass  = no_of_pass + 1;

// test of write data
for(i = 0; i <= `REG_INDEX_LIMIT; i = i + 1)
begin
#5      READ=1'b1; WRITE=1'b0; ADDR_R1 = i; ADDR_R2 = i;
#5      no_of_test = no_of_test + 1;
        if (DATA_R1 !== i || DATA_R2 !== i)
            $write("[TEST] Read %1b, Write %1b, expecting %8h, got %8h and %8h [FAILED]\n", READ, WRITE, i, DATA_R1, DATA_R2);
        else
            no_of_pass  = no_of_pass + 1;

end

#10    READ = 1'b0; WRITE = 1'b0; // No op

#10 $write("\n");
    $write("\tTotal number of tests %d\n", no_of_test);
    $write("\tTotal number of pass  %d\n", no_of_pass);
    $write("\n");
    $stop;

end
```

The code shown above is the remainder of the testbench code. It uses nearly the same code from the memory testbench. Aside from the renaming of registers, the differences are that data is no longer being loaded, each segment is extended to handle multiple registers rather than just one, and the print statements will now print two values instead of one in the case of an error. The results of the completed register file testbench will be shown later in the report.

D. Control Unit (CU)

The control unit is the main component of this project. It is in charge of handling the state machine, which includes all the processes involving in running an instruction, and handles control signal generation. Included within the instruction set are Register (R-type), Immediate (I-type), and Jump (J-type) instructions, all of which require their own unique implementation due to their different structures.

Before the instruction sets are implemented, the state machine must first be implemented. The state machine consists of five states that make up the instruction cycle: instruction fetch, instruction decode, execution, memory access, and write-back. The same respective states that the control unit will cycle through are: ‘PROC_FETCH’, ‘PROC_DECODE’, ‘PROC_EXE’, ‘PROB_MEM’, and ‘PROC_WB’. The structure of the cycle is a sequential behavioral model, introduced in ‘Lab_08’, one of the labs provided on CS147’s Canvas page. Within the lab is abstract code to demonstrate how the state machine will be implemented. The cycle of the states is as shown below:



Following this cycle, the state machine will loop through it infinitely as long as the clock continues to run and there are instructions to process.

```
// list of registers
reg [2:0] STATE;
reg [2:0] next_state;

// initialization
initial
begin
        STATE = 3'bxx;
        next_state = `PROC_FETCH;
end

// reset
always @ (negedge RST)
begin
        STATE = 3'bxx;
        next_state = `PROC_FETCH;
end

// state switch on clock cycle
always @ (posedge CLK)
begin
        STATE = next_state; // set current_state to next_state
        case (STATE) // next_state is moved one forward in the cycle
                `PROC_FETCH: next_state = `PROC_DECODE;
                `PROC_DECODE: next_state = `PROC_EXE;
                `PROC_EXE: next_state = `PROC_MEM;
                `PROC_MEM: next_state = `PROC_WB;
                `PROC_WB: next_state = `PROC_FETCH;
        endcase
end

endmodule;
```

To start, two registers, ‘STATE’ and ‘next_state’ are created to contain the address values for the states that is current and will be updated to on the next cycle. These registers will initially be set to ‘3’bxx’ and ‘`PROC_FETCH’ respectively as dictated by the state machine requirements.

If the reset signal is received, ‘STATE’ is set back to ‘3’bxx’ while ‘next_state’ is set back to ‘`PROC_FETCH’. When the clock is on the positive edge, the state will change following the order of the diagram. Before the change, ‘STATE’ is first set to ‘next_state’. After that, ‘next_state’ changes correspondingly with the diagram. The values for ‘`PROC_FETCH’, ‘`PROC_DECODE’, ‘`PROC_EXE’, ‘`PROCE_MEM’, and ‘`PROC_WB’ are all detailed within ‘prj_definition.v’.

The state machine has now been properly implemented. Now it must undergo testing to ensure correct results. Aside from the tedious task of connecting wires for the correct resources as well as the instantiation of the clock and control unit, the test for the state machine is very simple: all that needs to be done is have a reset signal and clock signal since those are the only signals needed for the state machine to behave.

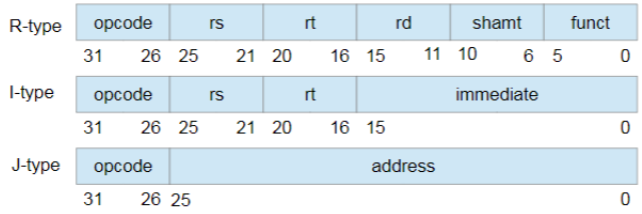
```
initial
begin
RST = 1'b0;

#15     RST = 1'b1;
#20     RST = 1'b0;

#100    $write("State machine testing completed. Check the waveform for results.");
        $stop;
end
endmodule
```

As shown in the short code, reset starts at 0 and is then set to 1 to reset the state machine from the first state. When reset is set to 0, the states will then cycle whenever the clock is on its positive edge. The testing will be shown further in the report.

However, the control unit is still not done yet. As mentioned earlier, the R-type, I-type, and J-type instructions must now be implemented. Below is a diagram that shows the difference between each instruction type as well as what they each require:

'CS147DV' Instruction Set

- 3 types of instructions.
  - Register or R type
  - Immediate or I type
  - Jump or J type



Aside from the bitwise differences shown in the diagram, the similaries and differences of the three instruction types can be further detailed.

R-Types have an ‘opcode’ value of h’00 and varying ‘funct’ values. All instructions under R-Types require the usage of the ALU aside from one. R-Types are handled in only execution and write-back state.

I-Types have different ‘opcode’ values and no ‘funct’ value, contrasting greatly from R-Types. Instructions under I-Type may need the ALU and are the only instruction type that may use sign-extended or zero-extended immediates. I-Types are handled in execution, memory access, and write-back state.

J-Types, similar to I-Types, have different ‘opcode’ values and no ‘funct’ values. However, J-Types take in only one other field, which is a 26-bit immediate value. J-Types are handled in execution, memory access, and write-back state.

Within the same ‘control_unit.v’ file, the three instructions make use of the state machine. In each cycle of the state machine, the appropriate actions are taken for each instruction.

```
// simulator internal storage for outputs
reg [`DATA_INDEX_LIMIT:0] RF_DATA_W, ALU_OP1, ALU_OP2;
reg [`REG_ADDR_INDEX_LIMIT:0] RF_ADDR_W, RF_ADDR_R1, RF_ADDR_R2;
reg RF_READ, RF_WRITE, MEM_READ, MEM_WRITE;
reg  [`ALU_OPRN_INDEX_LIMIT:0] ALU_OPRN;
reg [`ADDRESS_INDEX_LIMIT:0] MEM_ADDR;

// register for writing data
reg [`DATA_INDEX_LIMIT:0] REG_MEM_DATA;
assign MEM_DATA = (MEM_READ === 1'b0 && MEM_WRITE === 1'b1) ? REG_MEM_DATA : {`DATA_WIDTH{1'bz}};

// simulator internal storage for registers
reg [`ADDRESS_INDEX_LIMIT:0] PC_REG, SP_REF;
reg [`DATA_INDEX_LIMIT:0] INST_REG;
reg [5:0] opcode, funct;
reg [4:0] rs, rt, rd, shamt;
reg [15:0] immediate;
reg [25:0] address;
reg [`DATA_INDEX_LIMIT:0] sign_extend, zero_extend, lui, jump_address;

PROC_SM state_machine(.STATE(proc_state),.CLK(CLK),.RST(RST));

initial
begin
        PC_REG = `INST_START_ADDR;
        SP_REF = `INIT_STACK_POINTER;
end
```

Initially, registers are added for all output ports. A new one that will be added is a register for writing data to memory named ‘REG_MEM_DATA’. ‘MEM_DATA’ is assigned to

appropriately depending on if 'MEM_READ' and 'MEM_WRITE' are 0 and 1 respectively. To start, 'PC_REG' and 'SP_REF' are set to their correct starting address and pointer location respectively.

Each instruction type, as well as specific functions in each instruction type, must be handled uniquely due to their difference in bitwise structure and responsibilities. In each state of the intruction cycle, certain intructions may be handled while others may not.

1) Instruction Fetch State

The instruction fetch state does exactly as it is named: it fetches instructions. The instructions fetched are from the address that the program counter points at.

```
always @ (proc_state)
begin
        if (proc_state === `PROC_FETCH)
        begin
                MEM_ADDR = PC_REG;
                MEM_READ = 1'b1;
                MEM_WRITE = 1'b0;
                RF_READ = 1'b0;
                RF_WRITE = 1'b0;
        end
```

It is important to note the 'always @ (proc_state)' line. This line and everything contained under the 'begin' keyword immediately afterwards is the block in which the instruction states will be processed. Whenever the state changes, this block of code will run. The block of code will halt at its corresponding 'end' keyword after the write-back state.

The instruction fetch state is the shortest and simplest. 'MEM_ADDR' is set to the address of 'PC_REG,' and then 'MEM_READ' is set to 1 to prepare to read in the upcoming instruction. 'MEM_WRITE', 'RF_READ', and 'RF_WRITE' are all set to 0 as they are not to be used until later states.

2) Instruction Decode State

The instruction decode state occurs after the instruction fetch state takes the next instruction. Now, the instruction will be read and processed, determining what the specific instruction is.

```
if (proc_state === `PROC_DECODE)
begin
        INST_REG = MEM_DATA;

        // R-type
        {opcode, rs, rt, rd, shamt, funct} = INST_REG;

        // I-type
        {opcode, rs, rt, immediate } = INST_REG;

        // J-type
        {opcode, address} = INST_REG;

        // sign-extended immediate
        sign_extend = {{16{immediate[15]}}, immediate};

        // zero-extended immediate
        zero_extend = {16'b0, immediate};

        // I-type LUI value
        lui = {immediate, 16'b0};

        // J-type jump address
        jump_address = {6'b0, address};

        // registers for operations
        RF_ADDR_R1 = rs;
        RF_ADDR_R2 = rt;
        RF_READ = 1'b1;
        RF_WRITE = 1'b0;
end
```

'INST_REG' is first set to 'MEM_DATA', the data read in that contains the instruction. The bits of 'INST_REG' are then parsed and split appropriately into the different fields of each instruction. Sign-extended, zero-extended, high value, and jump address values are also calculated at this state in preparation to be potentially used in future states. These future states will also determine which type of instruction is read in depending on the parsed fields.

3) Execution State

The execution state will take the now decoded instruction and run it. Most instructions will be seen handled in this state with only a few exceptions, all of which are J-Types.

```
// any instructions with similar structures are grouped together for efficiency and code conciseness
if (proc_state === `PROC_EXE)
begin
        // R-types for execution state
        if (opcode === 6'h00) // R-Type opcode
        begin
                // funct code for shift left logical (sll) or shift right logical (srl)
                if (funct === 6'h01 || funct === 6'h02)
                begin
                        ALU_OP1 = RF_DATA_R1;
                        ALU_OP2 = shamt;
                        ALU_OPRN = funct;
                end

                else if (funct === 6'h08) // funct code for jump register (jr)
                begin
                        PC_REG = RF_DATA_R1;
                end

                else // the rest of the R-Types: add, sub, mul, and, or, nor, slt
                begin
                        ALU_OP1 = RF_DATA_R1;
                        ALU_OP2 = RF_DATA_R2;
                        ALU_OPRN = funct;
                end
        end

        // I-types for execution state
        // add immediate (addi), load word (lw), or store word (sw)
        else if (opcode === 6'h08 || opcode === 6'h23 || opcode === 6'h2b)
        begin
                ALU_OP1 = RF_DATA_R1;
                ALU_OP2 = sign_extend;
                ALU_OPRN = `ALU_OPRN_WIDTH'h20; // alu operation code for addition
        end

        else if (opcode === 6'h1d) // multiply immediate (muli)
        begin
                ALU_OP1 = RF_DATA_R1;
                ALU_OP2 = sign_extend;
                ALU_OPRN = `ALU_OPRN_WIDTH'h2c; // alu operation code for multiplication
        end

        else if (opcode === 6'h0c) // and immediate (andi)
        begin
                ALU_OP1 = RF_DATA_R1;
                ALU_OP2 = zero_extend;
                ALU_OPRN = `ALU_OPRN_WIDTH'h24; // alu operation code for bitwise AND
        end

        else if (opcode === 6'h0d) // or immediate (ori)
        begin
                ALU_OP1 = RF_DATA_R1;
                ALU_OP2 = zero_extend;
                ALU_OPRN = `ALU_OPRN_WIDTH'h25; // alu operation code for bitwise OR
        end

        else if (opcode === 6'h0a) // set less than immediate (slti)
        begin
                ALU_OP1 = RF_DATA_R1;
                ALU_OP2 = sign_extend;
                ALU_OPRN = `ALU_OPRN_WIDTH'h2a; // alu operation code for set less than
        end

        // J-Types for execution state
        else if (opcode === 6'h1b) // push (push)
                RF_ADDR_R1 = 1'b0;
end
```

As noted in the first line's comment, instead of handling every single type of instruction in their own separate case, instructions that are handled the same are grouped together for easier readability and simplicity than a switch case with dozens of different options.

For R-Types, all instructions are handled in the execution state. Shift left logical (sll) and shift right logical (srl), since both are shifting, are handled together. Jump register (jr) is handled simply by changing the 'PC_REG' value. Otherwise, all other R-Type instructions for execution state follow the same process.

For I-Types, all but branch on equal (beq) and branch on not equal (bne) are handled in execution state. Add immediate (andi), load word (lw), and store word (sw) all use a sign-extended value and then are given the same 'ALU_OPRN' code for addition since addition, loading, and storing are essentially adding a value to a different location. Following that same line of thought, multiply immediate (muli), and immediate (andi), or immediate (ori), and set less than immediate (slti) are provided their respective ALU operation code.

Finally, for J-Types, only one is handled which is push (push). For push, the 'RF_ADDR_R1' file is set to 0 in preparation for future states where push will reappear.

4) Memory Access State

The memory access state will edit the memory, either reading or writing, if necessary. Since memory access is only handled when necessary, only instructions that require access to memory will be seen in this state.

```
if (proc_state === `PROC_MEM)
begin
        // I-types for memory access state
        if (opcode === 6'h23) // load word (lw)
        begin
                MEM_ADDR = ALU_RESULT;
                MEM_READ = 1'b1;
                MEM_WRITE = 1'b0;
        end

        else if (opcode === 6'h2b) // store word (sw)
        begin
                MEM_ADDR = ALU_RESULT;
                REG_MEM_DATA = RF_DATA_R2;
                MEM_READ = 1'b0;
                MEM_WRITE = 1'b1;
        end

        // J-types for memory access state
        else if (opcode === 6'h1b) // push (push)
        begin
                MEM_ADDR = SP_REF;
                SP_REF = SP_REF - 1;
                REG_MEM_DATA = RF_DATA_R1;
                MEM_READ = 1'b0;
                MEM_WRITE = 1'b1;
        end

        else if (opcode === 6'h1c) // pop (pop)
        begin
                SP_REF = SP_REF + 1;
                MEM_ADDR = SP_REF;
                MEM_READ = 1'b1;
                MEM_WRITE = 1'b0;
        end
end
```

In this state, no R-Types are handled. Only specific I-Types and J-Types are in this state. Since not all instructions require memory access, only the one that do require memory access will appear in this state, in which are only four.

For I-Types, load word (lw) and store word (sw) are handled. This is clear because loading and storing data requires memory access. 'MEM_READ' and 'MEM_WRITE' are set to 1 and 0 or 0 and 1 depending on whether data is being loaded or stored respectively.

For J-Types, push (push) and pop (pop) are handled because both are editing the stack, which accesses memory. In both cases, the stack pointer 'SP_REF' is edited to decrease or increase when an element is pushed or popped respectively. Otherwise, just like loading and storing, 'MEM_READ' and 'MEM_WRITE' are set inverse to one another depending on the operation.

5) Write-back State

The write-back state writes into the register file. Similar to the memory access state, not all instructions require writing-back. The instructions that do not require write-back are load word (lw), store word (sw), push (push), and pop (pop). The reason behind this can be understood because these instructions are the only ones found in the memory access state.

```
if (proc_state === `PROC_WB)
begin
        PC_REG = PC_REG + 1;
        MEM_READ = 1'b0;
        MEM_WRITE = 1'b0;

        // R-types for write-back state
        if (opcode === 6'h00) // R-type opcode
        begin
                if (funct === 6'h08) // jump register (jr)
                        PC_REG = RF_DATA_R1;
                else // rest of R-types are the same: add, sub, mul, and, or, nor, slt, sll, srl
                begin
                        RF_ADDR_W = rd;
                        RF_DATA_W = ALU_RESULT;
                        RF_READ = 1'b0;
                        RF_WRITE = 1'b1;
                end
        end

        // I-types
        // add immediate (addi), multiply immediate (muli), and immediate (andi),
        // or immediate (ori), load upper immediate (lui), or set less than immediate (slti)
        else if (opcode === 6'h08 || opcode === 6'h1d || opcode === 6'h0c ||
                 opcode === 6'h0d || opcode === 6'h0f || opcode === 6'h0a)
        begin
                RF_ADDR_W = rt;

                if (opcode === 6'h0f)
                        RF_DATA_W = lui;
                else
                        RF_DATA_W = ALU_RESULT;

                RF_READ = 1'b0;
                RF_WRITE = 1'b1;
        end

        // branch on equal (beq) or branch on not equal (bne)
        else if ((opcode === 6'h04 && RF_DATA_R1 === RF_DATA_R2) ||
                 (opcode === 6'h05 && RF_DATA_R1 !== RF_DATA_R2))
                PC_REG = PC_REG + sign_extend;


                        // J-types
                        else if (opcode === 6'h02) // jump (jmp)
                                PC_REG = jump_address;

                        else if (opcode === 6'h03) // jump and link (jal)
                        begin
                                RF_DATA_W = PC_REG;
                                PC_REG = jump_address;
                                RF_ADDR_W = `REG_INDEX_LIMIT;
                                RF_READ = 1'b0;
                                RF_WRITE = 1'b1;
                        end

                        else if (opcode === 6'h1c) // pop (pop)
                        begin
                                RF_DATA_W = MEM_DATA;
                                RF_ADDR_W = 1'b0;
                                RF_READ = 1'b0;
                                RF_WRITE = 1'b1;
                        end
                end
        end
```

At the start, the program counter value 'PC_REG' is incremented by 1, 'MEM_READ' is set to 0, and 'MEM_WRITE' is set to 0. Having read and write both set to 0 is known as 'no-op', or no operation. This is in preparation for the state cycle to continue onto the next state, which loops back to the instruction fetch state. The program counter is incremented to point to the next instruction to be fetched while read and write are set to 0 since no data needs to be read or written to in the instruction fetch state.

For R-Types, all instructions are handled, but only jump register (jr) is handled uniquely. The 'PC_REG' register will be set to the address that the jump address instruction states.

Otherwise, all other R-Types will write ‘ALU_RESULT’ to register file at location ‘RF_ADDR_W = rd’.

For I-Types, all instructions are handled. Add immediate (andi), multiply immedaite (muli), and immediate (andi), or immediate (ori), load upper immedaite (lui), and set less than immediate (slti) are handled together. The register file address to write to is ‘rt’. Depending on if the operation is lui or not, the data that will be written will either be ‘lui’ or ‘ALU_RESULT’. The branch on equals (beq) and branch on not equals (bne) are handled together, resulting in ‘PC_REG’ to be changed if the conditions for either branch is met.

Finally, for J-Types, all instructions but push (push) are handled. Jump (jmp) changes the value of ‘PC_REG’. Jump and link (jal) changes ‘PC_REG’ as well, but also sets its registers in preparation for linking and running a different execution. Lastly, for pop (pop), the popped value is written into the register file.

With the state machine, R-Type, I-Type, and J-Type instructions implemented, the control unit is now implemented. In order to test it, a testbench is used, just like the rest of the component thus far. For the control unit, ‘da_vinci_tb.v’ has been included with partially included code as a sample for testing. The rest of the file must be completed for each instruction type to be tested.

E. DaVinci Testbench

In ‘da_vinci_tb.v’, code has already been given for testing Fibonacci and Revfib. While these are good tests to check, the more important test to implement is the functionality of the control unit, specifically the instruction type parsing.

```
// DA_VINCI v1.0 instance
//defparam da_vinci_inst.mem_init_file = "fibonacci.dat";
//defparam da_vinci_inst.mem_init_file = "RevFib.dat";
//defparam da_vinci_inst.mem_init_file = "r-type_test.dat";
//defparam da_vinci_inst.mem_init_file = "i-type_test.dat";
//defparam da_vinci_inst.mem_init_file = "j-type_test.dat";
DA_VINCI da_vinci_inst(.DATA(DATA), .ADDR(ADDR), .READ(READ),
                       .WRITE(WRITE), .CLK(CLK), .RST(RST));

initial
begin
RST=1'b1;
#5 RST=1'b0;
#5 RST=1'b1;

//# 20 $stop;
#5000  $write("Testing complete. Check the waveform for results.");
       //$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;

end
endmodule;
```

After commenting out lines of code that are unnecessary, new ‘defparam’ lines are added to link different instruction type test files. To run them, one is uncommented and run, viewing the waveforms to ensure functionality. A write statement has been included at the end to have a flag of when the test is complete. The contents of the test files will be shown in the ‘Testing’ section.

All testbenches have been completed. The results of each respective testbench will be shown in the following section.

IV. Testing

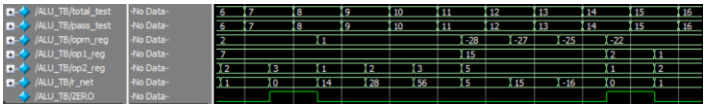
Finally, testing is a vital part of any project. To ensure the DaVinci v1.0 system works as intended, it is necessary to test each component to ensure functionality. Several components in this project requre testing:

A. Arithmathic Logic Unit (ALU)

Testing the ALU is the same as the ALU test for project one. Shown below are the results of the second project’s ALU:

```
# [TEST] 15 + 3 = 18 , got 18 ... [PASSED]
# [TEST] 15 - 5 = 10 , got 10 ... [PASSED]
# [TEST] 15 + 5 = 20 , got 20 ... [PASSED]
# [TEST] 2 * 5 = 10 , got 10 ... [PASSED]
# [TEST] 4 * 0 = 0 , got 0 ... [PASSED]
# [TEST] 7 >> 1 = 3 , got 3 ... [PASSED]
# [TEST] 7 >> 2 = 1 , got 1 ... [PASSED]
# [TEST] 7 >> 3 = 0 , got 0 ... [PASSED]
# [TEST] 7 << 1 = 14 , got 14 ... [PASSED]
# [TEST] 7 << 2 = 28 , got 28 ... [PASSED]
# [TEST] 7 << 3 = 56 , got 56 ... [PASSED]
# [TEST] 15 & 5 = 5 , got 5 ... [PASSED]
# [TEST] 15 | 5 = 15 , got 15 ... [PASSED]
# [TEST] 15 ~| 5 = 4294967280 , got 4294967280 ... [PASSED]
# [TEST] 2 < 1 = 0 , got 0 ... [PASSED]
# [TEST] 1 < 2 = 1 , got 1 ... [PASSED]
#
#       Total number of tests          16
#       Total number of pass           16
```

The same 16 cases tested from the first project are also successful for the second project. Now looking at the waveforms:



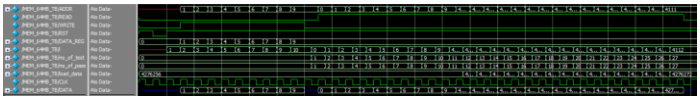
It can be seen that the values are changing appropriately and to the correct numbers, the same as project one’s waveforms. However, a new wave to take notice of is the ‘ZERO’ flag that was implemented in the testbench. Whenever the result ‘r_net’ is equal to 0, it can be seen that the value of ‘ZERO’ becomes 1. Whenever ‘r_net’ is not 0, ‘ZERO’ becomes 0. Since all test operations have passed alongside with the working inclusion of the ‘ZERO’ flag, ‘alu.v’ has been correctly implemented.

B. Memory

Testing the memory is the same as the ALU. Running it will provide a transcript that states number of test cases and number of passed test cases. Below are the results:

```
#
#       Total number of tests          27
#       Total number of pass           27
#
```

As shown, the 27 tests for memory have all passed. Additionally, it is important to look at the waveform diagram.



Though the image is small, by analyzing the waves, it can be seen that the clock is alternating from logic high and logic low, tests and their passes are counted, and the index ‘i’ is being incremented, as intended. These tests show that the memory is successful in writing to memory and reading it back, showing that the memory is performing correctly. The values that are written to memory are contained in ‘mem_dump_01.dat’. The contents are:

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

On the other hand, the contents of ‘mem_content_01.dat’ are read and written into the file ‘mem_dump_02.dat’. Below are the contents of ‘mem_content_01.dat’:

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

This same data can be seen in the file ‘mem_dump_02.dat’ as shown:

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

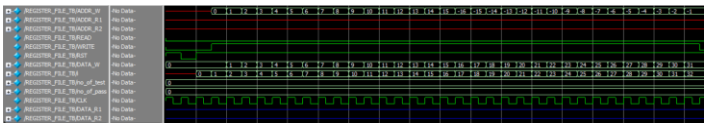
The memory file has been proven to work, succeeding on all 27 cases and having accurate waveform results. Similar testing process and test result behavior will be seen in the register file testing due to their very similar structure.

C. Register File

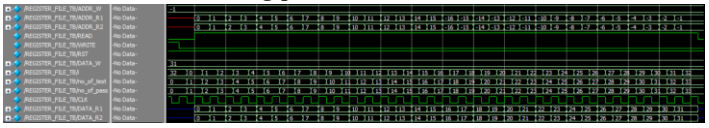
Following the same procedure as the ALU and memory testing, the following results for the register file have been output on the transcript:

```
#
#       Total number of tests          33
#       Total number of pass           33
#
```

All 33 test cases have passed, showing that the register file is successful in writing to all 32 registers and reading the same values back, along with the last case of no reading. Checking that the written and read values are the same means that the register file works correctly. Now looking at the waveforms:



The register file is seen initially resetting, which is what is wanted. The first half of the waveform is the writing phase. This phase shows that the register file is successful in writing to itself. Now in the reading phase:



Looking at the values being read from the register address, it can be seen that the values are the same as the values written in. This shows that the register file successfully implemented, being able to write and read from itself without problem.

D. Control Unit - State Machine

With the previously seen, very simple code, all that needs to be done is to simulate the state machine and analyze its waveforms cycle.

As shown, the states are cycling in the correct order. When the negative edge of the reset signal is met, ‘STATE’ is set back to an unknown value. Aside from immediately after a reset, it can be seen that ‘STATE’ and ‘next_state’ are always sequential and that both are cycling correctly. With the control unit’s state machine tested for functionality with success, the remainder of the control unit needs to be tested.

E. Control Unit - Instruction Parsing

To test the instructions, each instruction type is given a .dat file that contains functional ‘CS147’ Instruction set instructions. One of each instruction is tested to ensure that all instructions work. Each instruction type’s .dat file will then be run on ‘da_vinci_tb.v’ and the waveform will be checked to confirm the instructions have been implemented correctly.

1) R-Type

For R-Types, its .dat file contains:

```
@0001000
20420003 // addi r[2], r[2], 0x3; = 0010 0000 0100 0010 0000 0000 0000 0011
20840005 // addi r[4], r[4], 0x5; = 0010 0000 1000 0100 0000 0000 0000 0101
00441020 // add r[2], r[2], r[4]; = 0000 0000 0100 0100 0001 0000 0010 0000
00441022 // sub r[2], r[2], r[4]; = 0000 0000 0100 0100 0001 0000 0010 0010
0044102C // mul r[2], r[2], r[4]; = 0000 0000 0100 0100 0001 0000 0010 1100
00441024 // and r[2], r[2], r[4]; = 0000 0000 0100 0100 0001 0000 0010 0100
00441025 // or r[2], r[2], r[4]; = 0000 0000 0100 0100 0001 0000 0010 0101
00441027 // nor r[2], r[2], r[4]; = 0000 0000 0100 0100 0001 0000 0010 0110
0082102A // slt r[2], r[4], r[2]; = 0000 0000 1000 0010 0001 0000 0010 1010
00401141 // sll r[2], r[2], 0x5; = 0000 0000 0100 0000 0001 0001 0100 0001
00401142 // srl r[2], r[2], 0x5; = 0000 0000 0100 0000 0001 0001 0100 0002
00400008 // jr 0x0, r[2], 0x0; = 0000 0000 0100 0000 0000 0000 0000     1000
```

Running this file on ‘da_vinci_tb.v’, the waveform appears:

The waveform shows that all 12 of the R-Type instructions are functioning. In execution and write-back state, the correct registers were used. It can be seen in jump register (jr) that the program counter is correctly updated to the address dictated in the jump register instruction.

2) I-Type

For I-Types, its .dat file contains:

```
@0001000
20420003 // addi r[2], r[2], 0x3; = 0010 0000 0100 0010 0000 0000 0000 0011
74420005 // muli r[2], r[2], 0x5; = 0111 0100 0100 0010 0000 0000 0000 0101
30420004 // andi r[2], r[2], 0x4; = 0011 0000 0100 0010 0000 0000 0000 0100
3442000C // ori r[2], r[2], 0xC; = 0011 0100 0100 0010 0000 0000 0000 1100
3C020001 // lui r[2], 0x0, 0x1; = 0011 1100 0000 0010 0000 0000 0000 0001
28420002 // stli r[2], r[2], 0x2; = 0010 1000 0100 0010 0000 0000 0000 0010
10420000 // beq r[2], r[2], 0x1; = 0001 0000 0100 0010 0000 0000 0000 0000
20420003 // addi r[2], r[2], 0x3; = 0010 0000 0100 0010 0000 0000 0000 0011
14020000 // bne r[2], r[0], 0x2; = 0001 0100 0000 0010 0000 0000 0000 0000
8C020000 // lw r[2], r[0], 0x0; = 1000 1100 0000 0010 0000 0000 0000 0000
AC220000 // sw r[2], r[1], 0x0; = 1010 1100 0010 0010 0000 0000 0000 0000
```

Running this file on ‘da_vinci_tb.v’, the waveform appears:

The waveform shows that all 11 of the I-Type instructions are functioning. In execution, memory access, and write-back, the correct registers were used. It can be seen in branch instructions (beq & bne) that the program counter is correctly updated to the address dictated in the branch instructions.

3) J-Type

For J-Types, its .dat file contains:

```
@0001000
08001001 // jmp 0x0001001; = 0000 1000 0000 0000 0001 0000 0000 0001
20000003 // addi r[0], r[0], 0x3; = 0010 0000 0000 0000 0000 0000 0000 001
0C001003 // jal 0x0001003; = 0000 1100 0000 0000 0001 0000 0000 0011
6C000000 // push; = 0110 1100 0000 0000 0000 0000 0000 0000
70000000 // pop; = 0111 0000 0000 0000 0000 0000 0000 0000
```

Running this file on ‘da_vinci_tb.v’, the waveform appears:

The waveform shows that all 5 of the J-Type instructions are functioning. In execution, memory access, and write-back, the correct registers were used. It can be seen in push (push) and pop (pop) that the program counter and stack pointer are updated appropriately.

All components of the DaVinci v1.0 system has now been tested with success in all cases. Last of all is to test the entire system as a whole.

F. DaVinci v1.0

Provided alongside the code in ‘da_vinci_tb.v’ are two .dat files, ‘fibonacci.dat’ and ‘RevFib.dat’. By running each of these tests separately, the entire DaVinci v1.0 system can be tested. If the results written into ‘fibonacci_mem_dump.dat’ and ‘RevFib_mem_dump.dat’ match with the contents of ‘fibonacci_mem_dump.golden.dat’ and ‘RevFib_mem_dump.golden.dat’ respectively, then the tests are successful. It is important to note that ‘processor.v’ had to be corrected to have the correct port sizes in order to run properly. The comparisons of each pair of files are as shown:

fibonacci_mem_dump.dat

File Edit Format View He

```
// memory data file
// instance=/DA_VINC
// format=hex addres
00000000
00000001
00000001
00000002
00000003
00000005
00000008
0000000d
00000015
00000022
00000037
00000059
00000090
000000e9
00000179
00000262
```

fibonacci_mem_dump.golden.dat

File Edit Format View Help

```
// memory data file (do n
// instance=/DA_VINCI_TB/
// format=hex addressradi
00000000
00000001
00000001
00000002
00000003
00000005
00000008
0000000d
00000015
00000022
00000037
00000059
00000090
000000e9
00000179
00000262
```

RevFib_mem_dump.dat

File Edit Format View

```
// memory data file
// instance=/DA_VIN
// format=hex addre
ffffffc9
00000022
ffffffeb
0000000d
fffffff8
00000005
fffffffd
00000002
ffffffff
00000001
00000000
00000001
00000001
00000002
00000003
00000005
```

RevFib_mem_dump.golden.dat

File Edit Format View Help

```
// memory data file (do
// instance=/DA_VINCI_TB
// format=hex addressrad
ffffffc9
00000022
ffffffeb
0000000d
fffffff8
00000005
fffffffd
00000002
ffffffff
00000001
00000000
00000001
00000001
00000002
00000003
00000005
```

As shown by the two images, fibonacci and RevFib have both properly been read and written to file, showing that the DaVinci v1.0 system has been successfully implemented in all regards.

V. Conclusion

At the end of this project and report, I have learned how to implement the arithmetic logic unit, memory, register file, and control unit in Verilog. With a 100% success rate (at least on my generated test cases) on all components of, and including, the DaVinci v1.0 system, I am ready for future Verilog projects. With that, I have completed the second CS147 project in which I successfully implemented the DaVinci v1.0 computer system behaviorial model through Verilog in ModelSim.

References

[1] K. Patra, Class Lecture, Topic: "Computer System, Instruction Set, ALU," San José State University, San José, Jan., 28, 2020.

[2] K. Patra, Class Lab, Topic: "Behavioral Model II," San José State University, San José, Feb., 20, 2020.

[3] K. Patra, Class Lecture, Topic : "Clock, Memory, Controller, Von Neumann Arch, System SW," San José State University, San José, Feb., 30, 2020.