DaVinci 1.0: Simple Computer System Supporting CS147DV Instruction Set

Michelle Song
Department of Computer Science
San Jose State University
E-mail: michelle.song@sjsu.edu

***Abstract*— DaVinci v1.0 is a behavioral model of a simple computer system with the specifications of a 32-bit processor and 256MB memory. The system supports a special instruction set named CS147DV which is similar to MIPS instruction set with several modifications. The following report documents the process and explains the requirements of the implementation of DaVinci v1.0.**

I. Introduction

The everyday computer functions as a result of the conceptual and physical implementation of the computer system model. The computer system model consists of the memory, register file, ALU, and processor, with the control unit and clock connecting all of the parts together and synchronizing operations. DaVinci v1.0 features a functional computer system with a 32-bit processor and a minimal 256MB memory. The standard computer components in DaVinci v1.0 are implemented using HDL. The HDL, Verilog, is used to integrate the system and turn the digital design of the computer system into reality. The purpose of this project and report is to demonstrate how to install the simulation tool and execute the system, inform on the components of computer architecture, and successfully implement DaVinci v1.0.

II. Requirements

The following section states the software needed and minimal instructions for program execution and contains information on the concept of the computer system model that needs to be followed for accurate implementation.

A. *Software Requirements*

The digital simulation tool necessary for running the program is ModelSim. The installation process for the student edition will be specified, however, there are options for those who are not in academia. Additionally, the usage of ModelSim such as the creation and execution of the project will be briefly covered.

1) *Installation of ModelSim (Student Edition)*

To install ModelSim, visit the link to the student edition: http://www.mentor.com/company/higher_ed/modelsim-student-edition. Click on “Download Student Edition”.

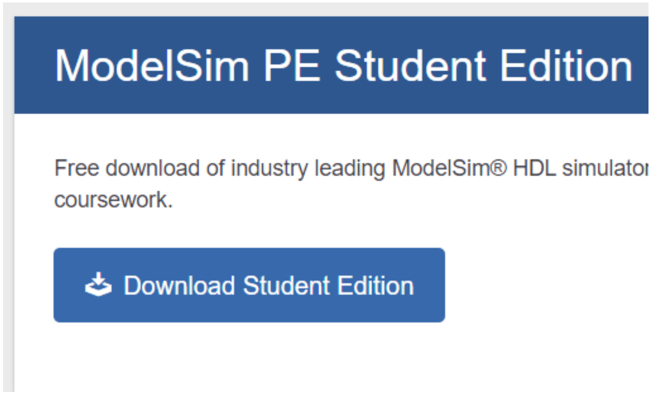
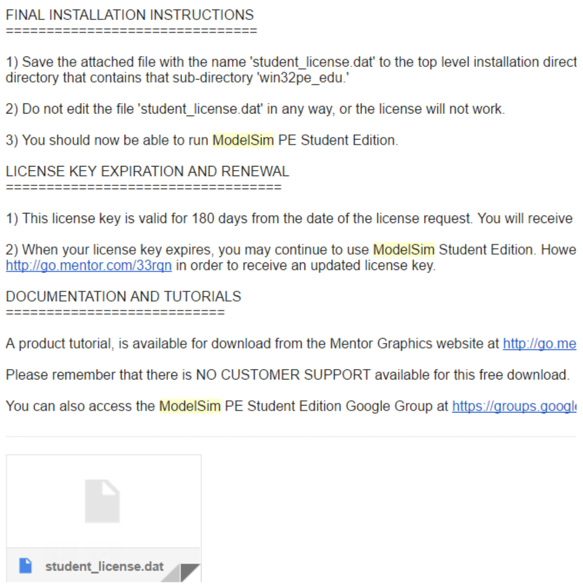

ModelSim PE Student Edition
Free download of industry leading ModelSim® HDL simulator
coursework.
Download Student Edition

Fig. 1. Downloading ModelSim – Download button

After installation, fill out the form to obtain the student license while ensuring that the email is correct. Next, check the email received from ModelSim and download the license attached to the email. There are additional instructions on the email for where to save the license file. As stated in the email, it is mandatory to keep the license file untouched for the license to work properly.



FINAL INSTALLATION INSTRUCTIONS
================================

1) Save the attached file with the name 'student_license.dat' to the top level installation direct directory that contains that sub-directory 'win32pe_edu.'

2) Do not edit the file 'student_license.dat' in any way, or the license will not work.

3) You should now be able to run ModelSim PE Student Edition.

LICENSE KEY EXPIRATION AND RENEWAL
==================================

1) This license key is valid for 180 days from the date of the license request. You will receive

2) When your license key expires, you may continue to use ModelSim Student Edition. Howe http://go.mentor.com/33rqn in order to receive an updated license key.

DOCUMENTATION AND TUTORIALS
===========================

A product tutorial, is available for download from the Mentor Graphics website at http://go.me

Please remember that there is NO CUSTOMER SUPPORT available for this free download.

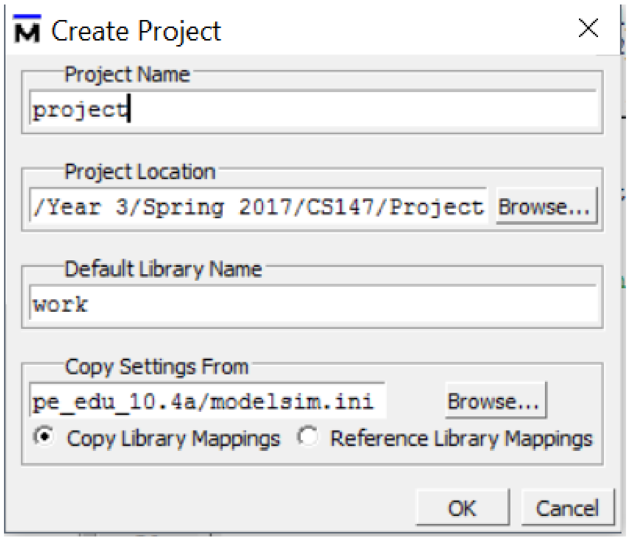
You can also access the ModelSim PE Student Edition Google Group at https://groups.googl

student_license.dat

Fig. 2. Downloading ModelSim – License

2) *Creating a simulation project*

After successful installation of ModelSim, open the workbench such that the menu and archives are displayed. After confirming that the project files (.v files) are downloaded, go to File -> New -> Project. Enter a name for the project and navigate to the directory where the project files were downloaded. Afterwards, press “OK”.



Create Project

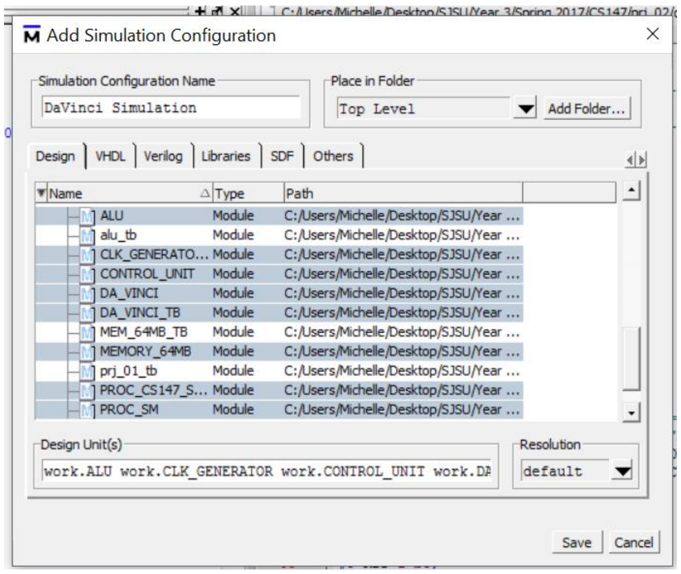
Project Name	project
Project Location	/Year 3/Spring 2017/CS147/Project
Default Library Name	work
Copy Settings From	pe_edu_10.4a/modelsim.ini

[✓ Copy Library Mappings] Reference Library Mappings

Fig. 3. Creating a project in ModelSim

3) *Creating a simulation*

Once the project has been created, select the project files excluding the test benches for ALU, memory, and register file which are for separate component testing. Then, right click and select Compile -> Compile All. Select Add to Project -> Simulation Configuration. On the design tab, expand the options for “work” and select the modules as shown in Figure 4. Once done, hit save.



Add Simulation Configuration

Simulation Configuration Name	DaVinci Simulation
Place in Folder	Top Level

Name	Type	Path
ALU	Module	C:/Users/Michelle/Desktop/SJSU/Year ...
alu_tb	Module	C:/Users/Michelle/Desktop/SJSU/Year ...
CLK_GENERATO...	Module	C:/Users/Michelle/Desktop/SJSU/Year ...
CONTROL_UNIT	Module	C:/Users/Michelle/Desktop/SJSU/Year ...
DA_VINCI	Module	C:/Users/Michelle/Desktop/SJSU/Year ...
DA_VINCI_TB	Module	C:/Users/Michelle/Desktop/SJSU/Year ...
MEM_64MB_TB	Module	C:/Users/Michelle/Desktop/SJSU/Year ...
MEMORY_64MB	Module	C:/Users/Michelle/Desktop/SJSU/Year ...
prj_01_tb	Module	C:/Users/Michelle/Desktop/SJSU/Year ...
PROC_CS147_S...	Module	C:/Users/Michelle/Desktop/SJSU/Year ...
PROC_SM	Module	C:/Users/Michelle/Desktop/SJSU/Year ...

Design Unit(s)	work.ALU work.CLK_GENERATOR work.CONTROL_UNIT work.DA
Resolution	default

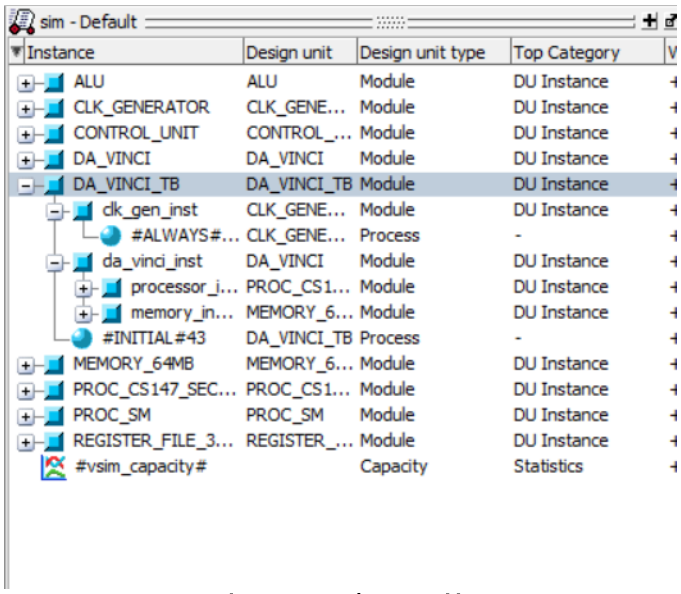
Fig. 4. Creating a simulation – Configuration window

4) *Running the simulation*

To run the simulation, right click on the name of the simulation created and click on execute. Then, on the toolbar at the top, go to Simulate -> Run - > Run –All. The memory data is dumped into a file depending on test settings. For example, if the Fibonacci program is selected, “fibonacci_mem_dump” would be created in the current directory. For more information on test cases, see section IV – Testing of this report.

5) *Observing waveforms*

To observe waveforms, go to the sim tab and double click DA_VINCI_TB. In the objects window that appears afterwards, select all objects and click on “Add wave”.

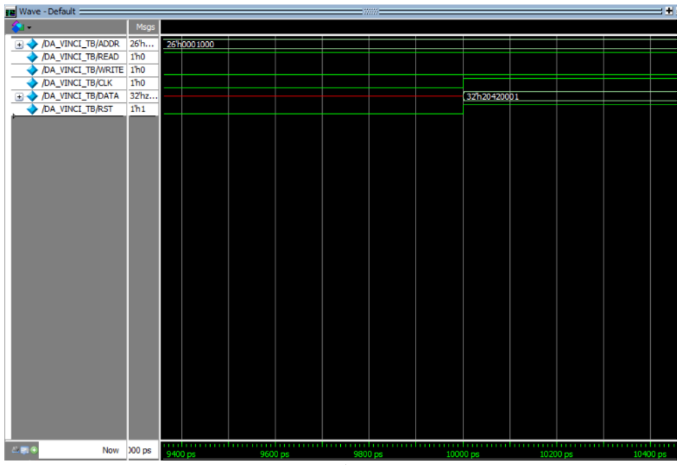


sim - Default

Instance	Design unit	Design unit type	Top Category
ALU	ALU	Module	DU Instance
CLK_GENERATOR	CLK_GENE...	Module	DU Instance
CONTROL_UNIT	CONTROL_...	Module	DU Instance
DA_VINCI	DA_VINCI	Module	DU Instance
DA_VINCI_TB	DA_VINCI_TB	Module	DU Instance
clk_gen_inst	CLK_GENE...	Module	DU Instance
#ALWAYS#...	CLK_GENE...	Process	-
da_vinci_inst	DA_VINCI	Module	DU Instance
processor_i...	PROC_CS1...	Module	DU Instance
memory_in...	MEMORY_6...	Module	DU Instance
#INITIAL#43	DA_VINCI_TB	Process	-
MEMORY_64MB	MEMORY_6...	Module	DU Instance
PROC_CS147_SEC...	PROC_CS1...	Module	DU Instance
PROC_SM	PROC_SM	Module	DU Instance
REGISTER_FILE_3...	REGISTER_...	Module	DU Instance
#vsim_capacity#		Capacity	Statistics

Fig. 5. Observing waveforms – Adding a wave

Next, run the simulation by clicking on “Simulate” on the top toolbar and selecting Run->Run –All. This enables navigation of the change in values occurring at specific time intervals in picoseconds.



Wave - Default

	Msgs
/DA_VINCI_TB/ADDR	26'h...
/DA_VINCI_TB/READ	1'h0
/DA_VINCI_TB/WRITE	1'h0
/DA_VINCI_TB/CLK	1'h0
/DA_VINCI_TB/DATA	32'hz...
/DA_VINCI_TB/RST	1'h1

Fig. 6. Observing waveforms – Viewing the wave

B. *Requirement for Computer System Model*

DaVinci v1.0 follows the computer system model consisting of the ALU, memory, register file, control unit, and clock. To understand the process of implementing DaVinci v1.0, it is important to understand the responsibility and requirements for each component of the computer system model. The following section states descriptions for each component.

1) ALU

The arithmetic and logic unit (ALU) is responsible for the mathematical and logical operations happening in a computer, providing the foundation for the functionality of a computer whose tasks are broken down into many arithmetic operations.

The structure of the ALU comprises of: two operand ports, one operation port, and a port for the output of the computation. The number bits for every port depends on the operation width of the computer. In DaVinci v1.0, the operation width of the computer is 32-bit, and thus, the number of bits for op1, op2, and the result is 32. In DaVinci v.1.0, there is also a zero flag which is turned on if the result from the ALU is 0. The zero flag is used for the instructions branch if equal and branch not equal.

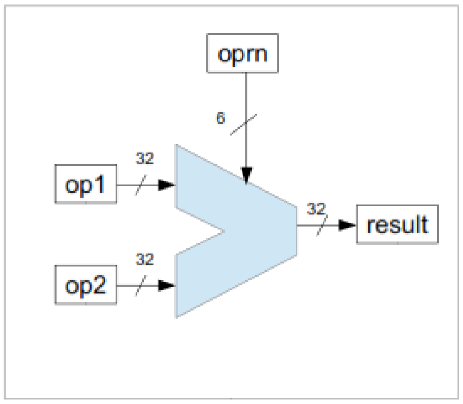
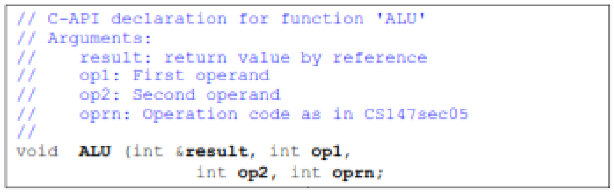
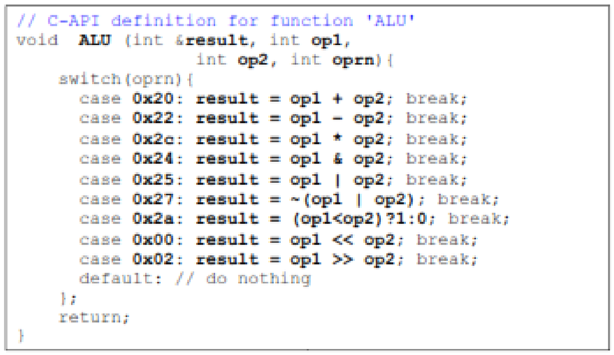


Fig.7. Schematic diagram representation of ALU [1]

The ALU is responsible for handling basic arithmetic operations such as addition, subtraction, multiplication, and division as well as logical operations such as AND, OR, NOT, and XOR. The correct operation is selected by the operation code passed from the control unit and applied to the two operands. The functionality of the ALU can be represented in a switch-case statement in the C code shown in Figure 8. Depending on the operation code given, an operation is selected to be used on the two operands.



```
// C-API declaration for function 'ALU'
// Arguments:
//    result: return value by reference
//    op1: First operand
//    op2: Second operand
//    oprn: Operation code as in CS147sec05
//
void  ALU (int &result, int op1,
                int op2, int oprn;
```



```
// C-API definition for function 'ALU'
void  ALU (int &result, int op1,
                int op2, int oprn){
    switch(oprn){
      case 0x20: result = op1 + op2; break;
      case 0x22: result = op1 - op2; break;
      case 0x2c: result = op1 * op2; break;
      case 0x24: result = op1 & op2; break;
      case 0x25: result = op1 | op2; break;
      case 0x27: result = ~(op1 | op2); break;
      case 0x2a: result = (op1<op2)?1:0; break;
      case 0x00: result = op1 << op2; break;
      case 0x02: result = op1 >> op2; break;
      default: // do nothing
    };
    return;
}
```

Fig.8. Corresponding C code for ALU [1]

2) Memory

The part of the computer that stores information such as instructions and data is the memory. The memory is essential for program execution since holds instructions and stores variables used by the program. The memory can be written into or read from by turning on/off the correct signals. To write to the memory, the read signal must be turned off and the write signal must be turned on. The opposite holds for reading to the memory. By inputting an address or data with the desired signal, the memory can be read from or written to. When the reset signal is turned on, all the values in the memory are set to 0. The address width depends on the size of the memory. In 1K memory, the address width is $\log_2 1K = 10$. In DaVinci v1.0, the size of the memory is 256MB so the address width is 28.

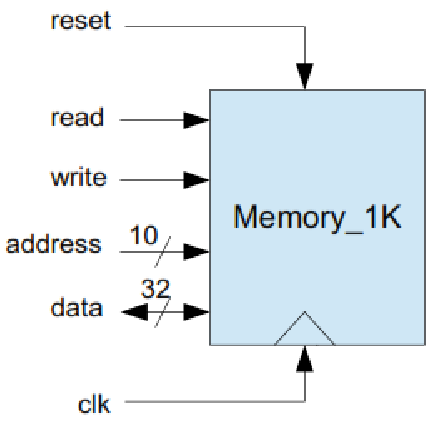


Fig.9. Schematic diagram of the memory [2]

3) Register File

The register file is a group of temporary registers located inside the processor, acting like a memory with data in and out ports. Similar to the memory, it stores information needed for a running program. The difference is that the register file acts like a cache memory and allows faster access to information. Register file sizes are limited, so for parallelism, two addresses can be inputted at the same time to be read from the register file.

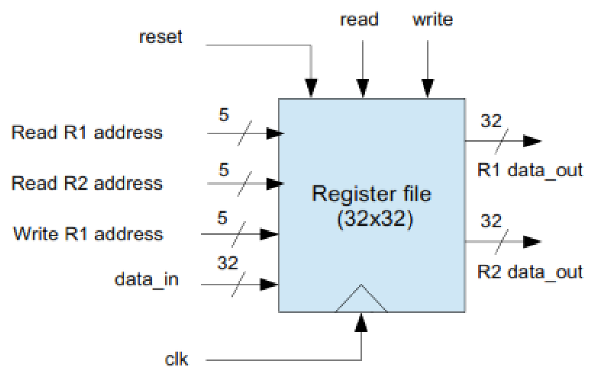


Fig.10. Schematic diagram of the register file [2]

4) Control Unit and Processor

The control unit issues the signals and manages data in the computer system. The stages of the control unit are shown in Figure 11. As an example, to execute a general R-Type instruction, the control unit fetches the operands from register file and issues the retrieved data as op1 and op2 to obtain the result from the ALU. The control unit controls the ALU result going back into the register file by issuing a write signal to the register file. All in all, the control unit is a necessity for the functionality of the processor.

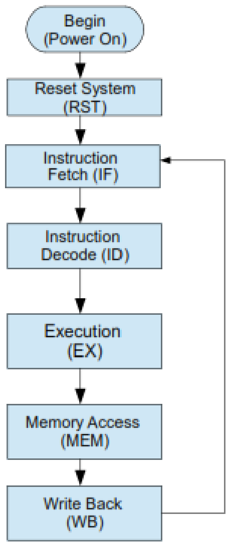


Fig.11. Control System Stages [2]

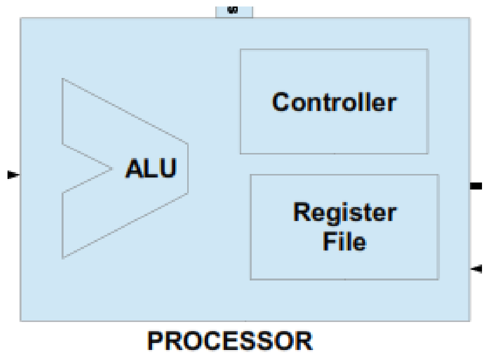


Fig.13. Processor [2]

In addition to the control unit's role in the processor, the control unit also manages the data flow from and into the memory. As shown in Figure 12, the control unit has signals for reset, read, and write into the memory and a data in/out as well as address input into the memory. The control unit issues signals to read/write from the memory for certain instructions such as store word, load word, push, and pop.

There are special registers in the control unit. One of them is the program counter which holds the address of the memory of the next instruction. The instruction memory is a register that holds the data of the current instruction which is fetched from the memory at the address of the previous program counter value.

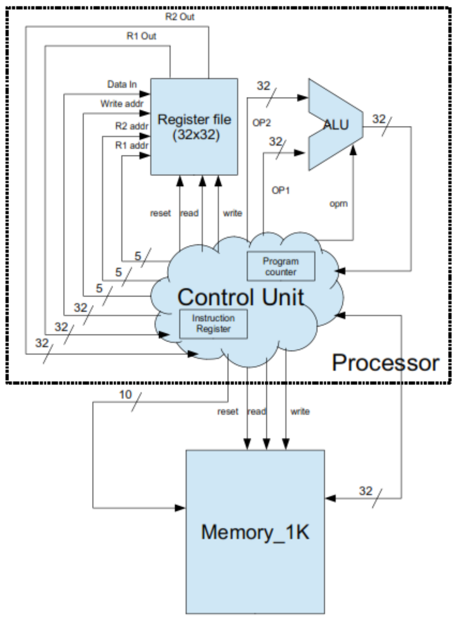


Fig.12. Control unit [2]

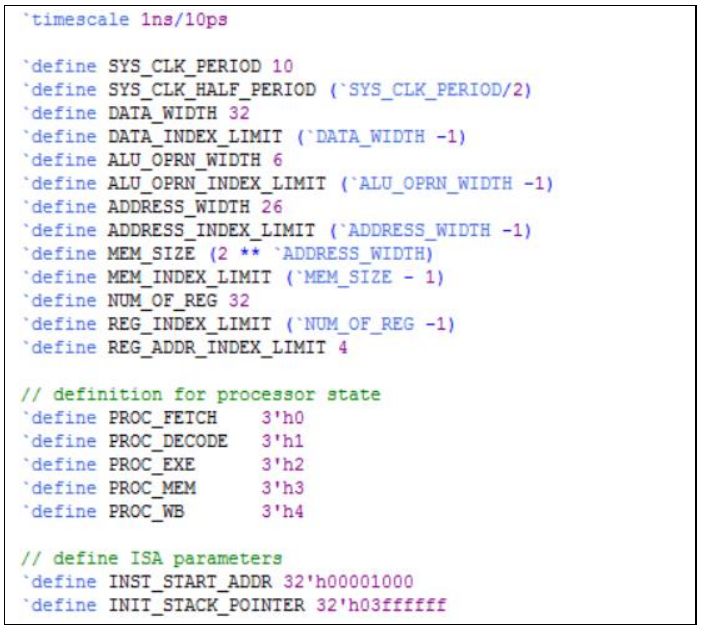
5) Clock

Along with the control unit issuing signals and data flow, the operations need to be synchronized so that they are performed in a desired timely manner. The clock switches between logic 0 and logic 1 depending on the clock period. The clock period is typically denoted as *T*, representing the time between clock ticks. In DaVinci v1.0, the clock period is defined as 10ns. Therefore, a half period = 5ns. In another words, the time the clock is at high is 5 ns and low at 5 ns, since the duty ratio is 50%.

III. IMPLEMENTATION

1) *prj_definition.v*

The project definition file “prj_definition.v” defines the clock periods, number of bits for all of the ports, values for the procedural state machine, and the ISA parameters for important addresses in the memory. The ALU, memory, and register file implemented utilizes these definitions to follow their requirements using the statement “include prj_definition.v” at the beginning of the file. The timescale is defined here indicates that the unit of time is 1 ns with the precision of 10ps. Since the clock period is defined as 10, the value in nanoseconds is 1ns * 10 = 10 ns. The widths of the ports are defined as stated in the requirements for each component. The stages of the control unit (fetch, decode, execution, memory, and write back) are assigned to values 0, 1, 2, 3, 4, respectively.



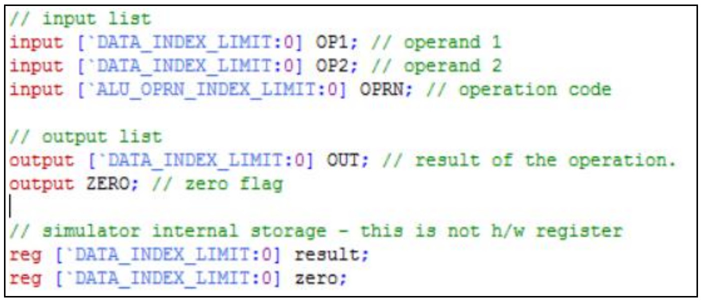
```
`timescale 1ns/10ps

`define SYS_CLK_PERIOD 10
`define SYS_CLK_HALF_PERIOD (`SYS_CLK_PERIOD/2)
`define DATA_WIDTH 32
`define DATA_INDEX_LIMIT (`DATA_WIDTH -1)
`define ALU_OPRN_WIDTH 6
`define ALU_OPRN_INDEX_LIMIT (`ALU_OPRN_WIDTH -1)
`define ADDRESS_WIDTH 26
`define ADDRESS_INDEX_LIMIT (`ADDRESS_WIDTH -1)
`define MEM_SIZE (2 ** `ADDRESS_WIDTH)
`define MEM_INDEX_LIMIT (`MEM_SIZE - 1)
`define NUM_OF_REG 32
`define REG_INDEX_LIMIT (`NUM_OF_REG -1)
`define REG_ADDR_INDEX_LIMIT 4

// definition for processor state
`define PROC_FETCH    3'h0
`define PROC_DECODE   3'h1
`define PROC_EXE      3'h2
`define PROC_MEM      3'h3
`define PROC_WB       3'h4

// define ISA parameters
`define INST_START_ADDR 32'h00001000
`define INIT_STACK_POINTER 32'h03ffffff
```

2) *alu.v*

The “alu.v” file creates a module or a design of the ALU providing a way of communication between ports. The module is declared with the keyword “module” followed by the name of the module, “alu”, and the name of the ports passed in as parameters: out (result), zero, op1, op2, and oprn.

i. Initializing the ports

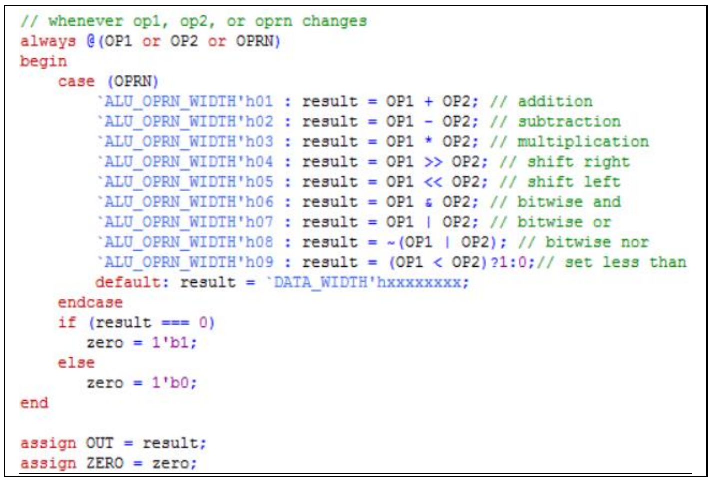
In the lines of code proceeding the declaration, whether the port is input or output is specified along with the port width. In this case, the ports op1, op2, and oprn are wire. Only the result and zero port needs to be specified as reg.



```
// 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; // zero flag

// simulator internal storage - this is not h/w register
reg [`DATA_INDEX_LIMIT:0] result;
reg [`DATA_INDEX_LIMIT:0] zero;
```

ii. Statements for basic and logic operations

The “always” block ensures that the ALU will perform as long as op1, op2, or oprn changes. Inside the always block, there is a case statement similar to the case-switch function as used in a higher level language like C or Java. Depending on the operation, the result is computed in a different way. For a simple example, if given the operands op1 = 5, op2 = 3, and operation code = 1 (addition), the result obtained will be 5 + 3 = 8. The 9 supported operations for the declared ALU are: addition, subtraction, multiplication, shift right, shift left, bitwise and, bitwise or, and set less than. Each operation has a corresponding opcode that will allow a different computation on the operands.



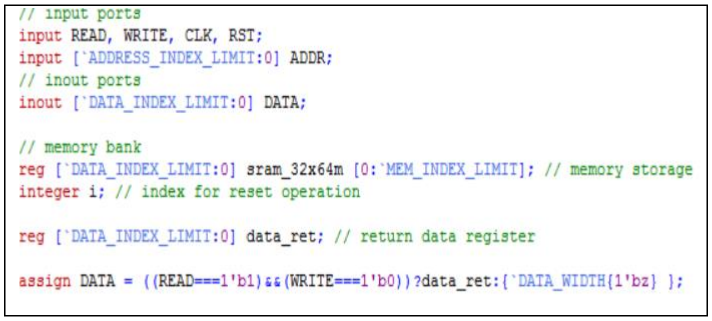
```
// whenever op1, op2, or oprn changes
always @(OP1 or OP2 or OPRN)
begin
    case (OPRN)
        `ALU_OPRN_WIDTH'h01 : result = OP1 + OP2; // addition
        `ALU_OPRN_WIDTH'h02 : result = OP1 - OP2; // subtraction
        `ALU_OPRN_WIDTH'h03 : result = OP1 * OP2; // multiplication
        `ALU_OPRN_WIDTH'h04 : result = OP1 >> OP2; // shift right
        `ALU_OPRN_WIDTH'h05 : result = OP1 << OP2; // shift left
        `ALU_OPRN_WIDTH'h06 : result = OP1 & OP2; // bitwise and
        `ALU_OPRN_WIDTH'h07 : result = OP1 | OP2; // bitwise or
        `ALU_OPRN_WIDTH'h08 : result = ~(OP1 | OP2); // bitwise nor
        `ALU_OPRN_WIDTH'h09 : result = (OP1 < OP2)?1:0;// set less than
       default: result = `DATA_WIDTH'hxxxxxxxx;
    endcase
    if (result === 0)
       zero = 1'b1;
    else
       zero = 1'b0;
end

assign OUT = result;
assign ZERO = zero;
```

3) *memory.v*

The file “memory.v” defines the memory module with read/write/reset signals, address port, and data in/out ports. It implements the functionality of the memory depending on reset, read, and write signals.

i. Initialization

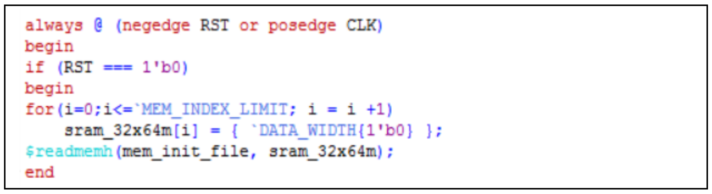
The ports for signals are initialized with 1 bit width each. For the memory, the data port is for both input and output it is specified as “inout”. The memory storage defined as a register with the size specified in the project definition file. There is a register that keeps the returned data from read operation. That data is only returned if the control is on read.



```
// 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} };
```

ii. Resetting the Memory Content

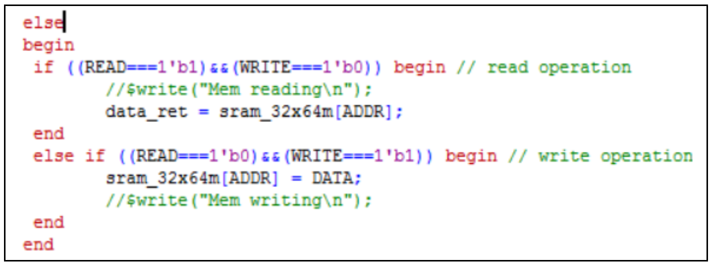
In the memory, there is an option to reset the content in the memory. The following implements this, setting all of the content in the memory to 0 and then initializing the rest of the memory according to the file used to initialize the memory.



```
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
```

iii. Data reading and writing

The following if/else statements check for the read or write operation. Again, read occurs when read is 1 and write is 0. Write occurs when read is 0 and write is 1. In read phase, set the data return register to the data contained in the memory at the input address. In the write phase, write the data as input and set the memory at the address location to that data.



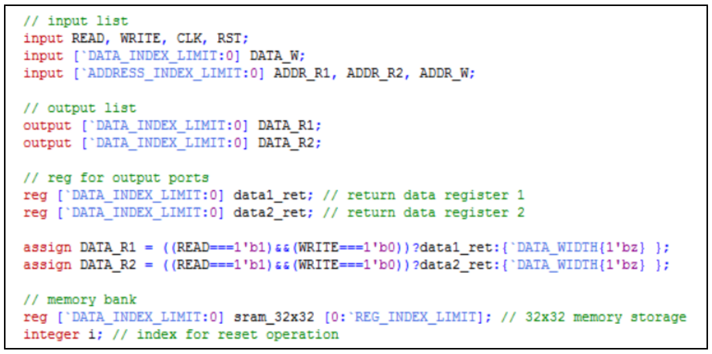
```
else
begin
 if ((READ===1'b1)&&(WRITE===1'b0)) begin // read operation
        //$write("Mem reading\n");
        data_ret = sram_32x64m[ADDR];
 end
 else if ((READ===1'b0)&&(WRITE===1'b1)) begin // write operation
        sram_32x64m[ADDR] = DATA;
        //$write("Mem writing\n");
 end
end
```

4) *register_file.v*

The register file of DaVinci v1.0 (32x32) is defined in this file. The register file is similar to memory in terms of port initialization and functionality. The main difference is the parallelism for reading from the register file.

i. Initialization

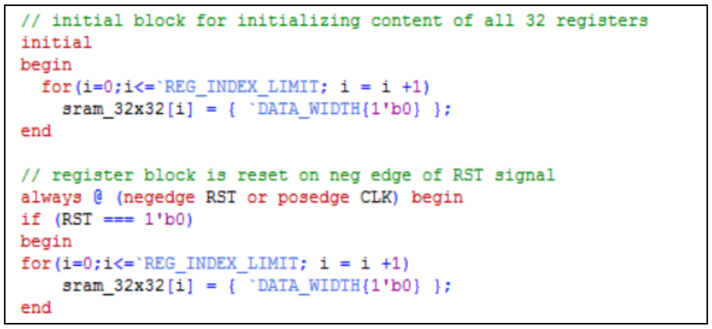
Similar to the memory, there are 1 bit input ports for the signals read, write, clock, and reset. Because of the parallelism for read, there are 2 read addresses inputs and 2 data return outputs. Another difference is the size of the memory storage, 32x32, following the specification of DaVinci v1.0.



```
// input list
input READ, WRITE, CLK, RST;
input [`DATA_INDEX_LIMIT:0] DATA_W;
input [`ADDRESS_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 for output ports
reg [`DATA_INDEX_LIMIT:0] data1_ret; // return data register 1
reg [`DATA_INDEX_LIMIT:0] data2_ret; // return data register 2

assign DATA_R1 = ((READ===1'b1)&&(WRITE===1'b0))?data1_ret:{`DATA_WIDTH{1'bz} };
assign DATA_R2 = ((READ===1'b1)&&(WRITE===1'b0))?data2_ret:{`DATA_WIDTH{1'bz} };

// memory bank
reg [`DATA_INDEX_LIMIT:0] sram_32x32 [0:`REG_INDEX_LIMIT]; // 32x32 memory storage
integer i; // index for reset operation
```

ii. Resetting the Register File Content

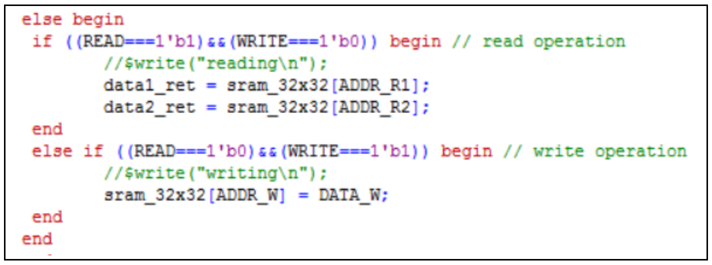
As in the memory, if the reset condition is selected, all of the content of the register file is set to 0. Since there does not exist an initialization from file option, there is nothing to load into the register file.



```
// initial block for initializing content of all 32 registers
initial
begin
  for(i=0;i<=`REG_INDEX_LIMIT; i = i +1)
    sram_32x32[i] = { `DATA_WIDTH{1'b0} };
end

// register block is reset on neg edge of RST signal
always @ (negedge RST or posedge CLK) begin
if (RST === 1'b0)
begin
for(i=0;i<=`REG_INDEX_LIMIT; i = i +1)
    sram_32x32[i] = { `DATA_WIDTH{1'b0} };
end
```

iii. Reading and Writing to Register File

The register file is able to read in parallel, and therefore, has two data outputs for read. When read is turned on, the data is read from the memory at both read addresses. When the write signal is on, the data from data write is written into the register file at the write address location.



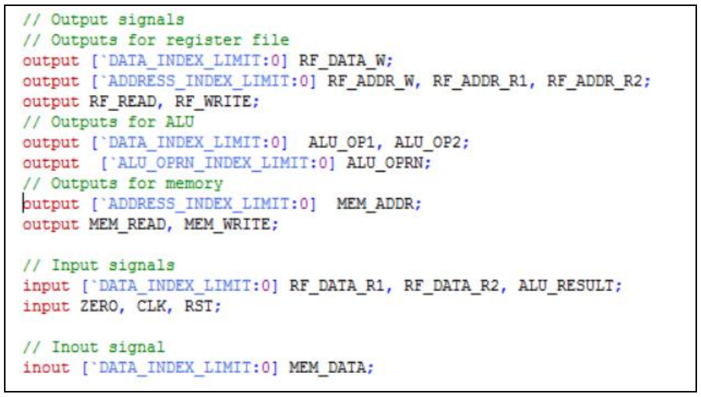
```
else begin
 if ((READ===1'b1)&&(WRITE===1'b0)) begin // read operation
        //$write("reading\n");
        data1_ret = sram_32x32[ADDR_R1];
        data2_ret = sram_32x32[ADDR_R2];
 end
 else if ((READ===1'b0)&&(WRITE===1'b1)) begin // write operation
        //$write("writing\n");
        sram_32x32[ADDR_W] = DATA_W;
 end
end
```

5) *control_unit.v*

The control unit module is responsible for the data flow in and out of ALU, register file, and memory, controlling the changing of states from one to the next by implementing a state machine and designating what happens at each stage.

i. Initializing the ports

In the control unit, all of the ports from ALU, memory, and register file are combined. Here, they are defined similar to how they were defined in the ALU, memory, and register file. However, in the control unit, the inputs are outputs and vice versa.

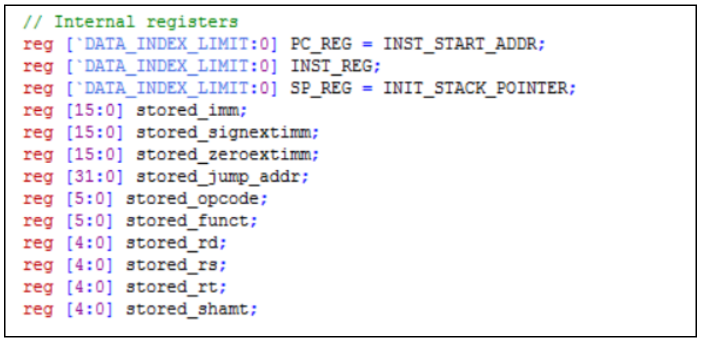


```
// Output signals
// Outputs for register file
output [`DATA_INDEX_LIMIT:0] RF_DATA_W;
output [`ADDRESS_INDEX_LIMIT:0] RF_ADDR_W, RF_ADDR_R1, RF_ADDR_R2;
output RF_READ, RF_WRITE;
// Outputs for ALU
output [`DATA_INDEX_LIMIT:0]  ALU_OP1, ALU_OP2;
output  [`ALU_OPRN_INDEX_LIMIT:0] ALU_OPRN;
// Outputs for memory
output [`ADDRESS_INDEX_LIMIT:0]  MEM_ADDR;
output MEM_READ, MEM_WRITE;

// Input signals
input [`DATA_INDEX_LIMIT:0] RF_DATA_R1, RF_DATA_R2, ALU_RESULT;
input ZERO, CLK, RST;

// Inout signal
inout [`DATA_INDEX_LIMIT:0] MEM_DATA;
```

ii. *Registers*

The special registers for the control unit are the program counter and instruction register. The ISA specification is defined in the project definition file the definition INST_START_ADDR is used to initialize the program counter register value to 32'h00001000.

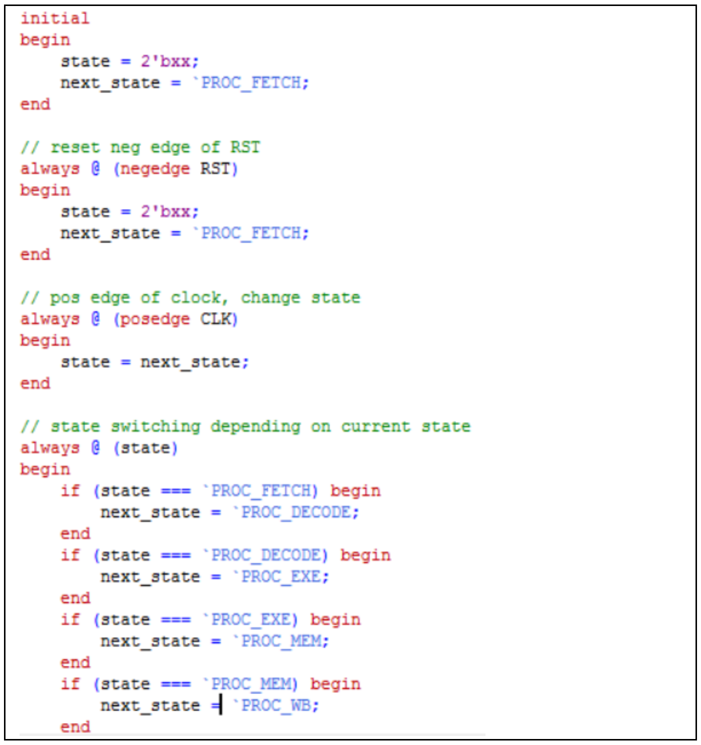
Other values need to be temporarily stored for each instruction. In the instruction decode stage, the values are parsed depending on instruction and stored into the corresponding registers to be used for future stages such as execution, memory access, and write back.



```
// Internal registers
reg [`DATA_INDEX_LIMIT:0] PC_REG = INST_START_ADDR;
reg [`DATA_INDEX_LIMIT:0] INST_REG;
reg [`DATA_INDEX_LIMIT:0] SP_REG = INIT_STACK_POINTER;
reg [15:0] stored_imm;
reg [15:0] stored_signextimm;
reg [15:0] stored_zeroextimm;
reg [31:0] stored_jump_addr;
reg [5:0] stored_opcode;
reg [5:0] stored_funct;
reg [4:0] stored_rd;
reg [4:0] stored_rs;
reg [4:0] stored_rt;
reg [4:0] stored_shamt;
```

iii. *State machine*

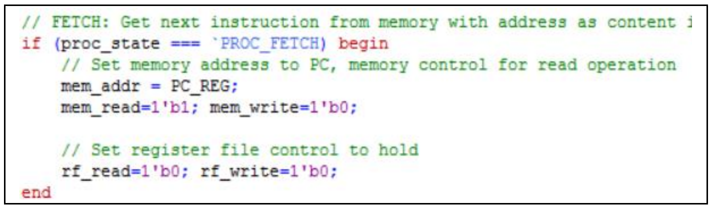
The control system model is essentially a state machine which is initially at 2’bxx state (unknown state). At every positive edge of the clock, the state switches to the next state as defined in the always block. The states switch from instruction fetch -> instruction decode -> execution -> memory -> write back and loops around as described in the control system model.



```
initial
begin
    state = 2'bxx;
    next_state = `PROC_FETCH;
end

// reset neg edge of RST
always @ (negedge RST)
begin
    state = 2'bxx;
    next_state = `PROC_FETCH;
end

// pos edge of clock, change state
always @ (posedge CLK)
begin
    state = next_state;
end

// state switching depending on current state
always @ (state)
begin
    if (state === `PROC_FETCH) begin
        next_state = `PROC_DECODE;
    end
    if (state === `PROC_DECODE) begin
        next_state = `PROC_EXE;
    end
    if (state === `PROC_EXE) begin
        next_state = `PROC_MEM;
    end
    if (state === `PROC_MEM) begin
        next_state = `PROC_WB;
    end
```

iv. *Instruction Fetch*

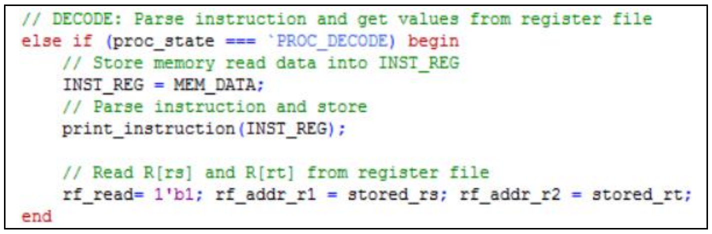
In the instruction fetch phase, the instruction at the address of program counter is fetched and stored in the instruction register. This stage consists of the set up by assigning the PC_REG value to mem_addr and turning the memory signal on. The register file control is set to hold since only the memory is being accessed.



```
// FETCH: Get next instruction from memory with address as content i
if (proc_state === `PROC_FETCH) begin
    // Set memory address to PC, memory control for read operation
    mem_addr = PC_REG;
    mem_read=1'b1; mem_write=1'b0;

    // Set register file control to hold
    rf_read=1'b0; rf_write=1'b0;
end
```

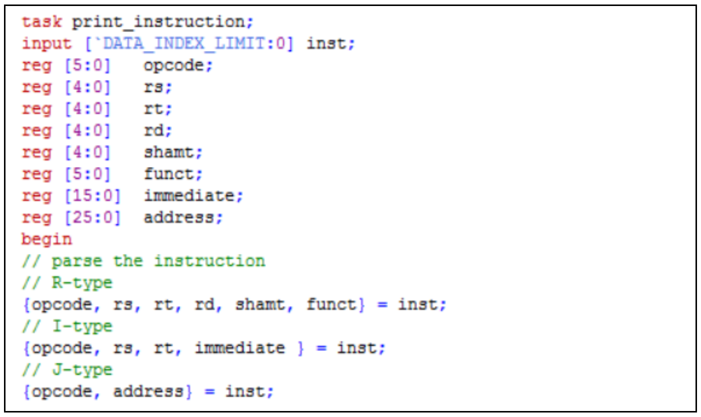
v. *Instruction Decode*

In the instruction decode phase, the instruction in INST_REG is parsed using the print instruction task. The retrieved values rs and rt are then used to read the values of R[rs] and R[rt] from the register file to prepare for the execution phase.



```
// DECODE: Parse instruction and get values from register file
else if (proc_state === `PROC_DECODE) begin
    // Store memory read data into INST_REG
    INST_REG = MEM_DATA;
    // Parse instruction and store
    print_instruction(INST_REG);

    // Read R[rs] and R[rt] from register file
    rf_read= 1'b1; rf_addr_r1 = stored_rs; rf_addr_r2 = stored_rt;
end
```

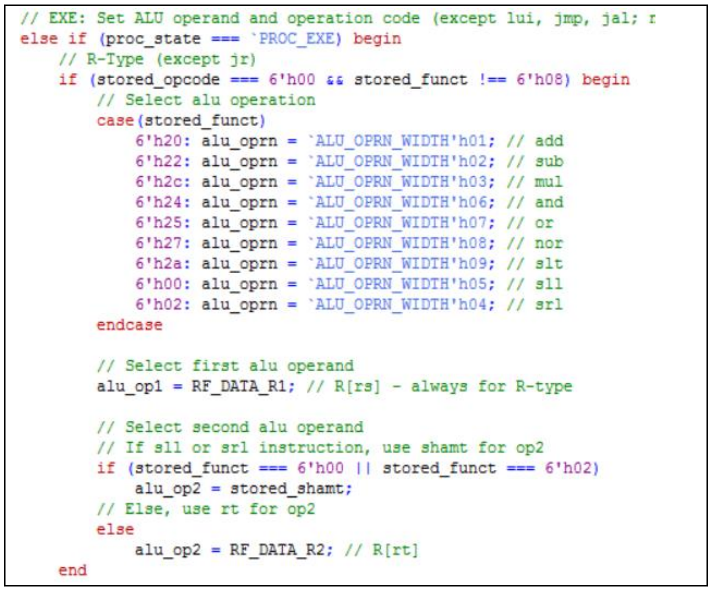
In the print instruction task, the instruction is parsed into a 6-bit opcode, 5-bit rs, rt, rd, shamt, and 6 bit function code for a R-type instruction. For I-Type, the instruction is parsed into a 16-bit immediate instead of rd, shamt, and funct. For J-Type, only the opcode and a 26-bit address is obtained. At the end of the task, the registers stored_rs, stored_rt, stored_rd, etc, will be assigned to rs, rt, rd, etc. to be used in future phases.



```
task print_instruction;
input [`DATA_INDEX_LIMIT:0] inst;
reg [5:0]   opcode;
reg [4:0]   rs;
reg [4:0]   rt;
reg [4:0]   rd;
reg [4:0]   shamt;
reg [5:0]   funct;
reg [15:0]  immediate;
reg [25:0]  address;
begin
// parse the instruction
// R-type
{opcode, rs, rt, rd, shamt, funct} = inst;
// I-type
{opcode, rs, rt, immediate } = inst;
// J-type
{opcode, address} = inst;
```

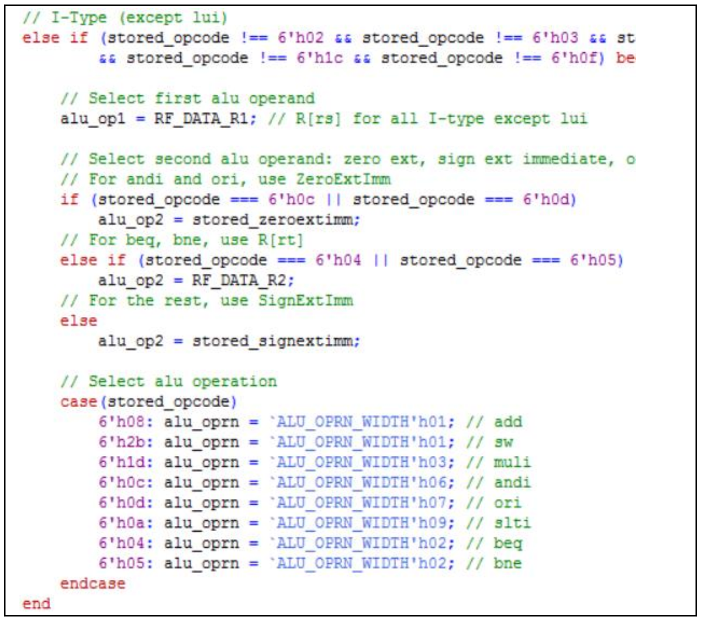
vi. *Execution*

In the execution phase, the majority of the instructions need to perform computations using the ALU. For R-type instructions excluding jump register, the operands are rs and rt/shamt. Depending on the function code, the correct ALU operation code is selected. The first ALU operand is always rs and the second one is rt or shamt for only the sll and srl instructions. In the next clock cycle (memory access), the result will appear in the alu_result register.



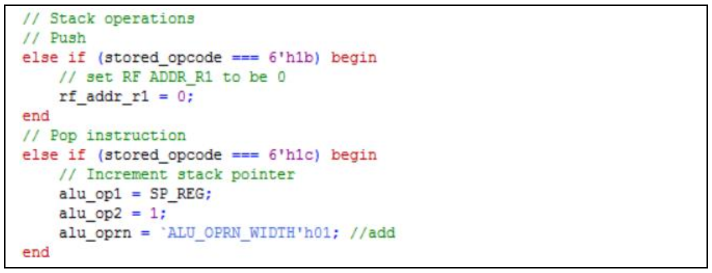
```
// EXE: Set ALU operand and operation code (except lui, jmp, jal; r
else if (proc_state === `PROC_EXE) begin
    // R-Type (except jr)
    if (stored_opcode === 6'h00 && stored_funct !== 6'h08) begin
        // Select alu operation
        case(stored_funct)
            6'h20: alu_oprn = `ALU_OPRN_WIDTH'h01; // add
            6'h22: alu_oprn = `ALU_OPRN_WIDTH'h02; // sub
            6'h2c: alu_oprn = `ALU_OPRN_WIDTH'h03; // mul
            6'h24: alu_oprn = `ALU_OPRN_WIDTH'h06; // and
            6'h25: alu_oprn = `ALU_OPRN_WIDTH'h07; // or
            6'h27: alu_oprn = `ALU_OPRN_WIDTH'h08; // nor
            6'h2a: alu_oprn = `ALU_OPRN_WIDTH'h09; // slt
            6'h00: alu_oprn = `ALU_OPRN_WIDTH'h05; // sll
            6'h02: alu_oprn = `ALU_OPRN_WIDTH'h04; // srl
        endcase

        // Select first alu operand
        alu_op1 = RF_DATA_R1; // R[rs] - always for R-type

        // Select second alu operand
        // If sll or srl instruction, use shamt for op2
        if (stored_funct === 6'h00 || stored_funct === 6'h02)
            alu_op2 = stored_shamt;
        // Else, use rt for op2
        else
            alu_op2 = RF_DATA_R2; // R[rt]
    end
```

The same concept of selecting the ALU operands and operation applies for the I-Type instruction but the difference is that there are more special cases. Rs is selected for the first operand in the ALU for all instructions but the lui instruction. Then, the second ALU operand is signed extended or zero extended. Zero extension happens for the andi and ori instruction. The ALU operation is selected similar to the R-Type instruction but the I-Type depends on the operation code instead of the function code.



```
// I-Type (except lui)
else if (stored_opcode !== 6'h02 && stored_opcode !== 6'h03 && st
        && stored_opcode !== 6'h1c && stored_opcode !== 6'h0f) be

    // Select first alu operand
    alu_op1 = RF_DATA_R1; // R[rs] for all I-type except lui

    // Select second alu operand: zero ext, sign ext immediate, o
    // For andi and ori, use ZeroExtImm
    if (stored_opcode === 6'h0c || stored_opcode === 6'h0d)
        alu_op2 = stored_zeroextimm;
    // For beq, bne, use R[rt]
    else if (stored_opcode === 6'h04 || stored_opcode === 6'h05)
        alu_op2 = RF_DATA_R2;
    // For the rest, use SignExtImm
    else
        alu_op2 = stored_signextimm;

    // Select alu operation
    case(stored_opcode)
        6'h08: alu_oprn = `ALU_OPRN_WIDTH'h01; // add
        6'h2b: alu_oprn = `ALU_OPRN_WIDTH'h01; // sw
        6'h1d: alu_oprn = `ALU_OPRN_WIDTH'h03; // muli
        6'h0c: alu_oprn = `ALU_OPRN_WIDTH'h06; // andi
        6'h0d: alu_oprn = `ALU_OPRN_WIDTH'h07; // ori
        6'h0a: alu_oprn = `ALU_OPRN_WIDTH'h09; // slti
        6'h04: alu_oprn = `ALU_OPRN_WIDTH'h02; // beq
        6'h05: alu_oprn = `ALU_OPRN_WIDTH'h02; // bne
    endcase
end
```

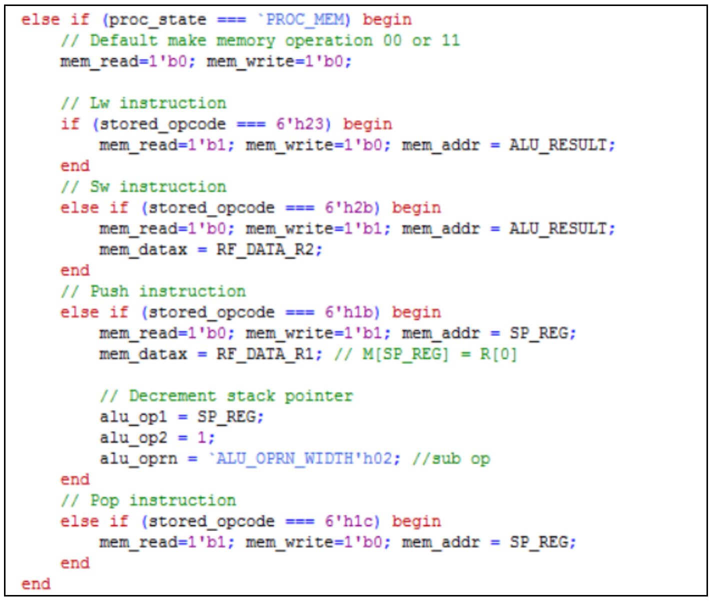
Only the push and pop instructions of the J-Type are configured at this stage. For push, set the register file read address to be 0 to prepare to write the result R[0] into the memory since the data result from the register file takes one clock cycle to obtain. For pop, increment the stack pointer by selecting the operand as the stack pointer register, the second operand to 1, and the ALU operation to the add instruction.



```
// Stack operations
// Push
else if (stored_opcode === 6'h1b) begin
    // set RF ADDR_R1 to be 0
    rf_addr_r1 = 0;
end
// Pop instruction
else if (stored_opcode === 6'h1c) begin
    // Increment stack pointer
    alu_op1 = SP_REG;
    alu_op2 = 1;
    alu_oprn = `ALU_OPRN_WIDTH'h01; //add
end
```

vii. *Memory Access*

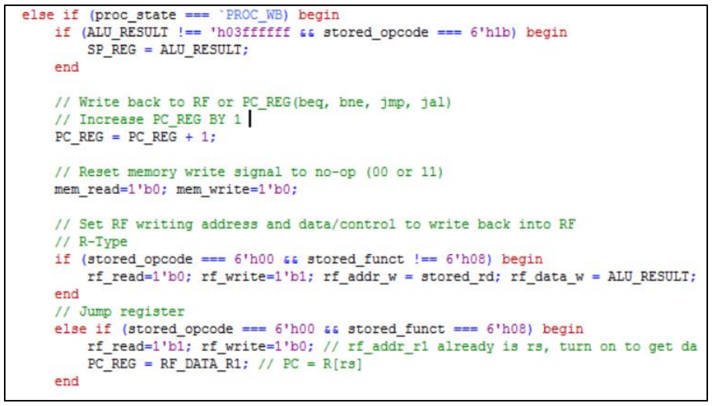
The memory write back phase is only applicable for lw, sw, push, and pop instructions. By default, the memory read and write is set to hold. For the case of load word, the address to read from in the memory is the address computed by the ALU. For store word, the memory at the computed address location is set to the data from R[rt].



```
else if (proc_state === `PROC_MEM) begin
    // Default make memory operation 00 or 11
    mem_read=1'b0; mem_write=1'b0;

    // Lw instruction
    if (stored_opcode === 6'h23) begin
        mem_read=1'b1; mem_write=1'b0; mem_addr = ALU_RESULT;
    end
    // Sw instruction
    else if (stored_opcode === 6'h2b) begin
        mem_read=1'b0; mem_write=1'b1; mem_addr = ALU_RESULT;
        mem_datax = RF_DATA_R2;
    end
    // Push instruction
    else if (stored_opcode === 6'h1b) begin
        mem_read=1'b0; mem_write=1'b1; mem_addr = SP_REG;
        mem_datax = RF_DATA_R1; // M[SP_REG] = R[0]

        // Decrement stack pointer
        alu_op1 = SP_REG;
        alu_op2 = 1;
        alu_oprn = `ALU_OPRN_WIDTH'h02; //sub op
    end
    // Pop instruction
    else if (stored_opcode === 6'h1c) begin
        mem_read=1'b1; mem_write=1'b0; mem_addr = SP_REG;
    end
end
```

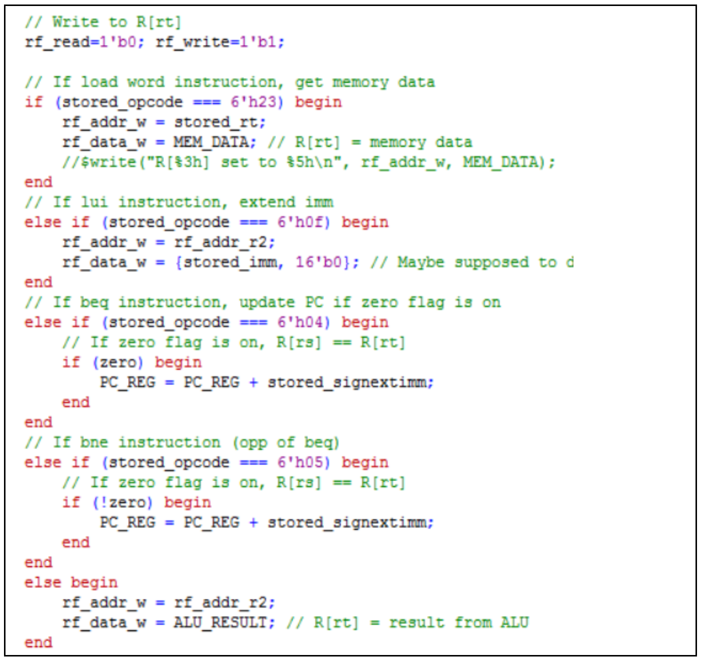
viii. Write Back

By default, in the write back stage, the program counter is incremented by 1. Since any writing happens to the register file and not the memory, the memory read and write is set to 0. Certain data is written into a certain address in the register file depending on the type of instruction. For most R-Type instructions, the ALU result is written into the destination of R[rd]. To do so, the register file write address is set to rd and the data to write to is set to ALU_RESULT. There is an exception for jump register, which simply sets the PC value to the value of R[rs].



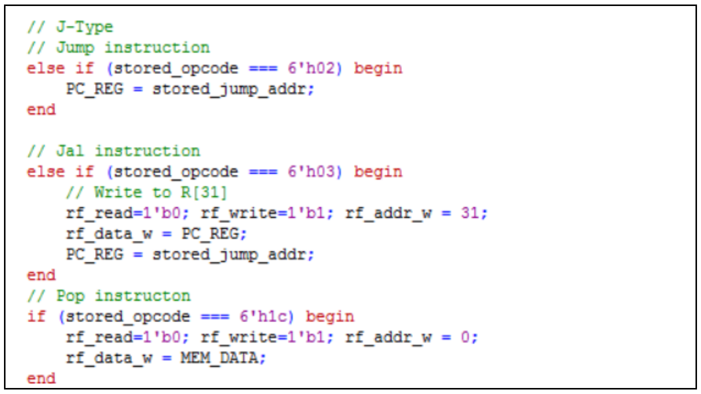
```
else if (proc_state === `PROC_WB) begin
    if (ALU_RESULT !== 'h03ffffff && stored_opcode === 6'h1b) begin
        SP_REG = ALU_RESULT;
    end

    // Write back to RF or PC_REG(beq, bne, jmp, jal)
    // Increase PC_REG BY 1
    PC_REG = PC_REG + 1;

    // Reset memory write signal to no-op (00 or 11)
    mem_read=1'b0; mem_write=1'b0;

    // Set RF writing address and data/control to write back into RF
    // R-Type
    if (stored_opcode === 6'h00 && stored_funct !== 6'h08) begin
        rf_read=1'b0; rf_write=1'b1; rf_addr_w = stored_rd; rf_data_w = ALU_RESULT;
    end
    // Jump register
    else if (stored_opcode === 6'h00 && stored_funct === 6'h08) begin
        rf_read=1'b1; rf_write=1'b0; // rf_addr_r1 already is rs, turn on to get da
        PC_REG = RF_DATA_R1; // PC = R[rs]
    end
```

For I-Type instructions, ALU_RESULT is written back into the destination of R[rt] by selecting the address of rt for the register file. However, there is an exception for lui and branch instructions. For lui, the lower half of rt is set 16 bits of 0 while keeping the upper half. For branch instructions, the zero flag is checked, and if the condition is satisfied, the PC updates by adding the stored sign immediate value to itself.



```
// Write to R[rt]
rf_read=1'b0; rf_write=1'b1;

// If load word instruction, get memory data
if (stored_opcode === 6'h23) begin
    rf_addr_w = stored_rt;
    rf_data_w = MEM_DATA; // R[rt] = memory data
    //$write("R[%3h] set to %5h\n", rf_addr_w, MEM_DATA);
end
// If lui instruction, extend imm
else if (stored_opcode === 6'h0f) begin
    rf_addr_w = rf_addr_r2;
    rf_data_w = {stored_imm, 16'b0}; // Maybe supposed to d
end
// If beq instruction, update PC if zero flag is on
else if (stored_opcode === 6'h04) begin
    // If zero flag is on, R[rs] == R[rt]
    if (zero) begin
        PC_REG = PC_REG + stored_signextimm;
    end
end
// If bne instruction (opp of beq)
else if (stored_opcode === 6'h05) begin
    // If zero flag is on, R[rs] == R[rt]
    if (!zero) begin
        PC_REG = PC_REG + stored_signextimm;
    end
end
else begin
    rf_addr_w = rf_addr_r2;
    rf_data_w = ALU_RESULT; // R[rt] = result from ALU
end
```

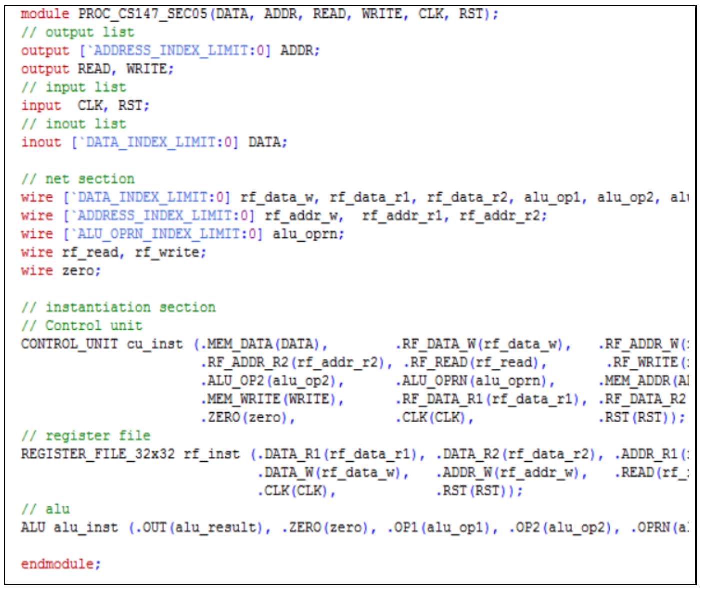
The jump instruction sets the program counter to the jump address. The jump and link instruction stores the current value in the PC register while updating the PC. The pop instruction writes to R[0] and the data written is the resulting memory data value from the memory access stage.



```
// J-Type
// Jump instruction
else if (stored_opcode === 6'h02) begin
    PC_REG = stored_jump_addr;
end

// Jal instruction
else if (stored_opcode === 6'h03) begin
    // Write to R[31]
    rf_read=1'b0; rf_write=1'b1; rf_addr_w = 31;
    rf_data_w = PC_REG;
    PC_REG = stored_jump_addr;
end
// Pop instructon
if (stored_opcode === 6'h1c) begin
    rf_read=1'b0; rf_write=1'b1; rf_addr_w = 0;
    rf_data_w = MEM_DATA;
end
```

6) processor.v

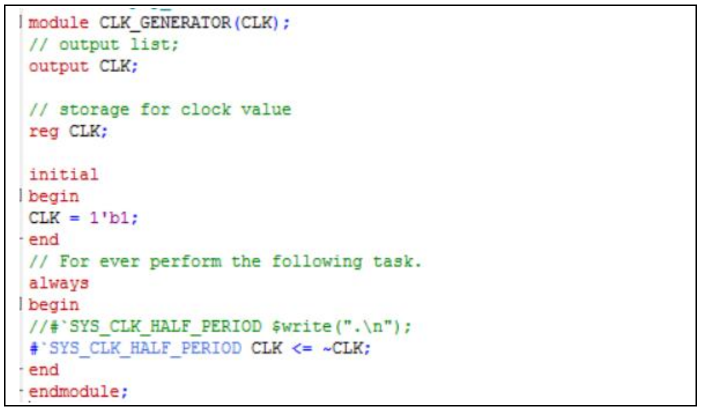
Since the processor contains the ALU, register file, and control unit, the input/output ports of the ALU, register file, and control unit are initialized. Additionally, the components are instantiated using their module definitions and the ports are passed in as parameters to connect them together.



```
module PROC_CS147_SEC05(DATA, ADDR, READ, WRITE, CLK, RST);
// output list
output [`ADDRESS_INDEX_LIMIT:0] ADDR;
output READ, WRITE;
// input list
input  CLK, RST;
// inout list
inout [`DATA_INDEX_LIMIT:0] DATA;

// net section
wire [`DATA_INDEX_LIMIT:0] rf_data_w, rf_data_r1, rf_data_r2, alu_op1, alu_op2, al
wire [`ADDRESS_INDEX_LIMIT:0] rf_addr_w,  rf_addr_r1, rf_addr_r2;
wire [`ALU_OPRN_INDEX_LIMIT:0] alu_oprn;
wire rf_read, rf_write;
wire zero;

// instantiation section
// Control unit
CONTROL_UNIT cu_inst (.MEM_DATA(DATA),        .RF_DATA_W(rf_data_w),    .RF_ADDR_W(
                      .RF_ADDR_R2(rf_addr_r2), .RF_READ(rf_read),         .RF_WRITE(
                      .ALU_OP2(alu_op2),       .ALU_OPRN(alu_oprn),       .MEM_ADDR(A
                      .MEM_WRITE(WRITE),       .RF_DATA_R1(rf_data_r1), .RF_DATA_R2
                      .ZERO(zero),             .CLK(CLK),                 .RST(RST));
// register file
REGISTER_FILE_32x32 rf_inst (.DATA_R1(rf_data_r1), .DATA_R2(rf_data_r2), .ADDR_R1(
                             .DATA_W(rf_data_w),   .ADDR_W(rf_addr_w),   .READ(rf_
                             .CLK(CLK),            .RST(RST));
// alu
ALU alu_inst (.OUT(alu_result), .ZERO(zero), .OP1(alu_op1), .OP2(alu_op2), .OPRN(a

endmodule;
```

7) *clk_gen.v*

In the clock module, the output clock signal and register is defined to be able to change the output of the signal. Initially, the clock is set to high. For every half period or 5ns, the clock inverts, turning the signal off if on and on if off.



```
module CLK_GENERATOR(CLK);
// output list;
output CLK;

// storage for clock value
reg CLK;

initial
begin
CLK = 1'b1;
end
// For ever perform the following task.
always
begin
//#`SYS_CLK_HALF_PERIOD $write(".\n");
#`SYS_CLK_HALF_PERIOD CLK <= ~CLK;
end
endmodule;
```

IV. TESTING

After installation of ModelSim and ensuring that the project is properly loaded, select the preferred configuration (program) by commenting/uncommenting the correct memory initialization file. Then, start the simulation and run all. The results will be dumped into a .dat file. To check the correctness of the program, compare the dumped memory file to the golden .dat file corresponding to the program selected.

1) *Testing procedure*

Test benches are provided for the entire behavioral computer system along with the following architectural components: ALU, memory, register file. The following explains the test bench code and procedure for ensuring that the implementation of each component is correct.

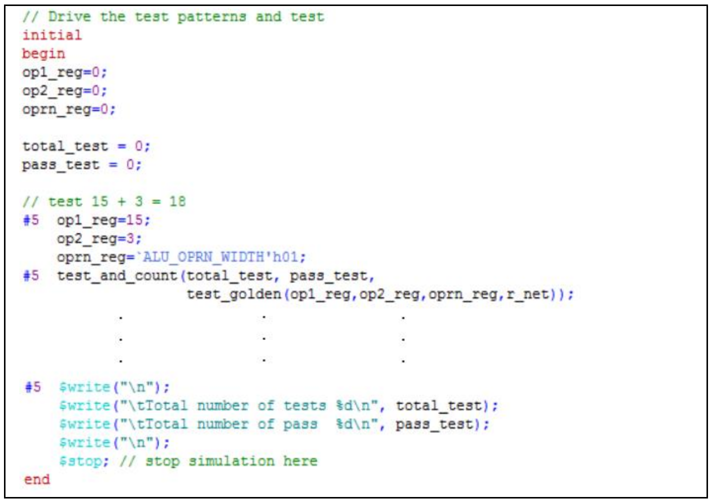
i. *Timing control*

The first line of code in "prj_01_tb.v" specifies the time unit for delays that occur during simulation. The statement "`timescale 1ns/10ps" indicates that the timing delays are multiplied by 1ns. The compiler rounds the resulting delay by the closest integer multiple of 10ps.

ii. *ALU Testing*

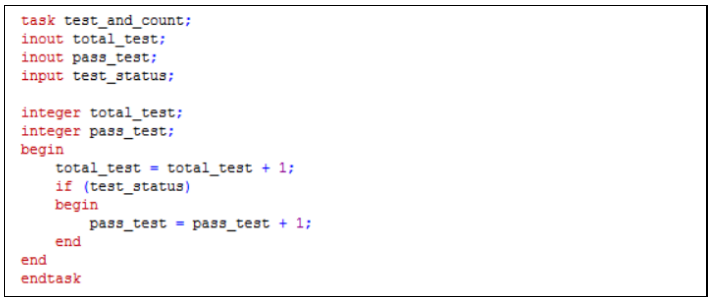
The testing of the implemented ALU is done in the project file named "alu_tb.v". To verify that the ALU is implemented correctly, the result from the ALU is compared to a golden result by calling a function in the test program.

After initializing the integer representing the total number of tests cases, passed tests, and registers to a value of 0 for normalization purposes, test cases were created for each operation. Note that the majority of the test cases in the following code snippet were removed for readability purposes. The testing code in the testing file, however, includes all of the test cases.



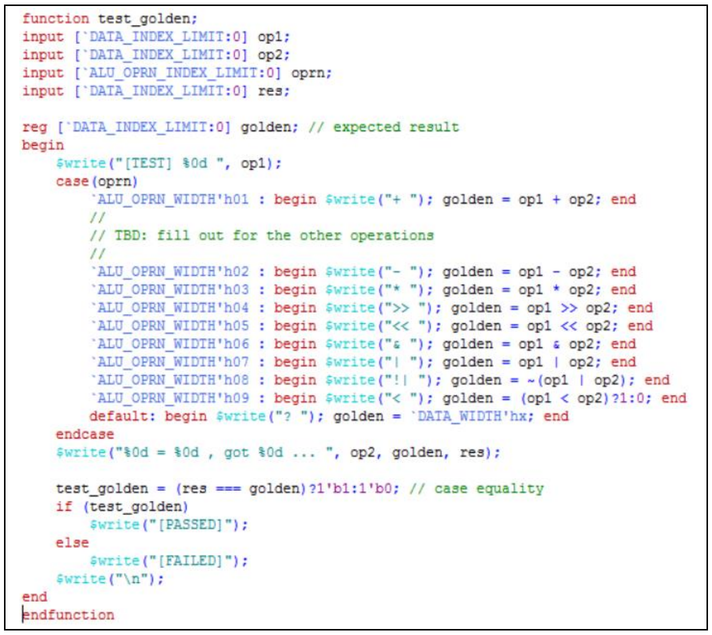
```
// Drive the test patterns and test
initial
begin
op1_reg=0;
op2_reg=0;
oprn_reg=0;

total_test = 0;
pass_test = 0;

// test 15 + 3 = 18
#5  op1_reg=15;
    op2_reg=3;
    oprn_reg=`ALU_OPRN_WIDTH'h01;
#5  test_and_count(total_test, pass_test,
                   test_golden(op1_reg,op2_reg,oprn_reg,r_net));
           .                .                .
           .                .                .
           .                .                .

#5  $write("\n");
    $write("\tTotal number of tests %d\n", total_test);
    $write("\tTotal number of pass  %d\n", pass_test);
    $write("\n");
    $stop; // stop simulation here
end
```

Each test case calls the task "test_and_count" as defined in the same file. The task runs the test and increments the total tests by 1 and increments the number of passed tests by 1 if the test outcome was successful, meaning, the golden result matched the result from the ALU.



```
task test_and_count;
inout total_test;
inout pass_test;
input test_status;

integer total_test;
integer pass_test;
begin
    total_test = total_test + 1;
    if (test_status)
    begin
        pass_test = pass_test + 1;
    end
end
endtask
```

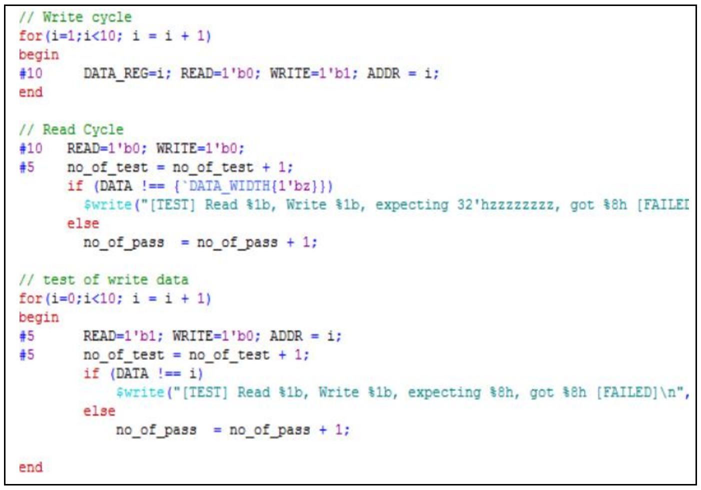
The "test_golden" task tests the ALU result to a golden result depending on the operation code. Following the case block is the comparison of the golden result to the ALU result written as output. This task tests for case equality and writes "PASSED" or "FAILED" depending on the outcome of the test.



```
function test_golden;
input [`DATA_INDEX_LIMIT:0] op1;
input [`DATA_INDEX_LIMIT:0] op2;
input [`ALU_OPRN_INDEX_LIMIT:0] oprn;
input [`DATA_INDEX_LIMIT:0] res;

reg [`DATA_INDEX_LIMIT:0] golden; // expected result
begin
    $write("[TEST] %0d ", op1);
    case(oprn)
        `ALU_OPRN_WIDTH'h01 : begin $write("+ "); golden = op1 + op2; end
        //
        // TBD: fill out for the other operations
        //
        `ALU_OPRN_WIDTH'h02 : begin $write("- "); golden = op1 - op2; end
        `ALU_OPRN_WIDTH'h03 : begin $write("* "); golden = op1 * op2; end
        `ALU_OPRN_WIDTH'h04 : begin $write(">> "); golden = op1 >> op2; end
        `ALU_OPRN_WIDTH'h05 : begin $write("<< "); golden = op1 << op2; end
        `ALU_OPRN_WIDTH'h06 : begin $write("& "); golden = op1 & op2; end
        `ALU_OPRN_WIDTH'h07 : begin $write("| "); golden = op1 | op2; end
        `ALU_OPRN_WIDTH'h08 : begin $write("!| "); golden = ~(op1 | op2); end
        `ALU_OPRN_WIDTH'h09 : begin $write("< "); golden = (op1 < op2)?1:0; end
        default: begin $write("? "); golden = `DATA_WIDTH'hx; end
    endcase
    $write("%0d = %0d , got %0d ... ", op2, golden, res);

    test_golden = (res === golden)?1'b1:1'b0; // case equality
    if (test_golden)
        $write("[PASSED]");
    else
        $write("[FAILED]");
    $write("\n");
end
endfunction
```

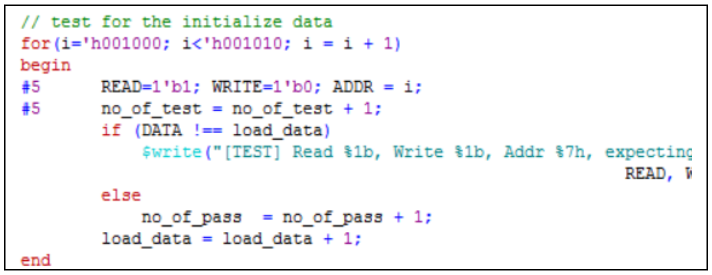
iii. *Memory testing*

The test bench of the memory is named "mem_64MB_tb.v". In this test bench, the testing is done by writing values into the memory and reading them to see if they are the equal. There is also a test for checking the Hi-Z state of the memory by setting the read signal to 0 and the write signal to 0.



```
// Write cycle
for(i=1;i<10; i = i + 1)
begin
#10     DATA_REG=i; READ=1'b0; WRITE=1'b1; ADDR = i;
end

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

// test of write data
for(i=0;i<10; i = i + 1)
begin
#5      READ=1'b1; WRITE=1'b0; ADDR = i;
#5      no_of_test = no_of_test + 1;
        if (DATA !== i)
            $write("[TEST] Read %1b, Write %1b, expecting %8h, got %8h [FAILED]\n",
        else
            no_of_pass  = no_of_pass + 1;

end
```

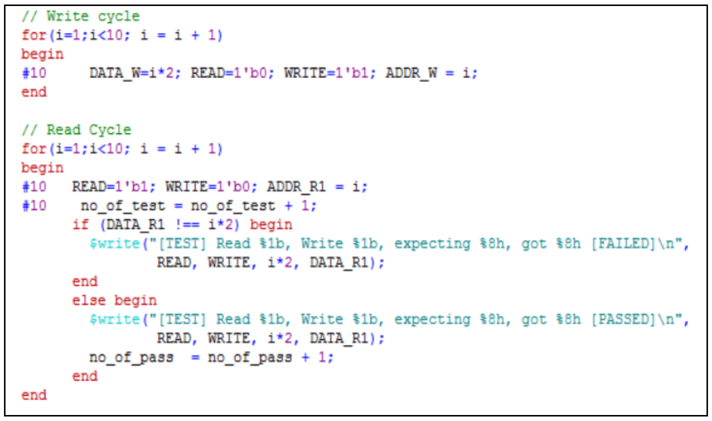
The memory also has the option to initialize data from a data file and that must be tested as well. In the beginning of the test bench, the initialization file is defined as "mem_content_01.dat". The following snippet of code tests for the initialization of data, checking if the load_data variable is equal to data in the memory.



```
// test for the initialize data
for(i='h001000; i<'h001010; i = i + 1)
begin
#5      READ=1'b1; WRITE=1'b0; ADDR = i;
#5      no_of_test = no_of_test + 1;
        if (DATA !== load_data)
            $write("[TEST] Read %1b, Write %1b, Addr %7h, expecting
                                                        READ, W
        else
            no_of_pass  = no_of_pass + 1;
        load_data = load_data + 1;
end
```

iv. *Register file testing*

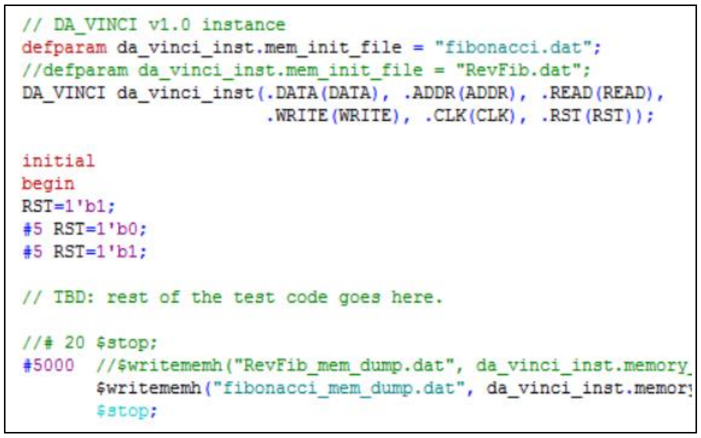
The testing of the register file is done in "reg_32x32_tb.v". The testing procedure is similar to that of the memory without the test for initialization since data cannot be loaded into the register file from the reset signal. To test the register file, data is written into the register file and checked during the read cycle. If the data from read cycle is equal to the data supposedly written during the write cycle, the test will pass with 10/10.



```
// Write cycle
for(i=1;i<10; i = i + 1)
begin
#10     DATA_W=i*2; READ=1'b0; WRITE=1'b1; ADDR_W = i;
end

// Read Cycle
for(i=1;i<10; i = i + 1)
begin
#10   READ=1'b1; WRITE=1'b0; ADDR_R1 = i;
#10    no_of_test = no_of_test + 1;
      if (DATA_R1 !== i*2) begin
        $write("[TEST] Read %1b, Write %1b, expecting %8h, got %8h [FAILED]\n",
                READ, WRITE, i*2, DATA_R1);
      end
      else begin
        $write("[TEST] Read %1b, Write %1b, expecting %8h, got %8h [PASSED]\n",
                READ, WRITE, i*2, DATA_R1);
        no_of_pass  = no_of_pass + 1;
      end
end
```

v. *Testing for entire system*

To test the entire system, use the "da_vinci_tb.v" and select the memory initialization file and the corresponding memory dump file. In the following code snippet, the fibonacci.dat file is selected so the corresponding memory dump file is "fibonacci_mem_dump.dat". Start the simulation and run the test bench. In the directory of the project files, the memory dump file will be updated. Comparing the dumped memory file the golden file will allow checking of the correctness of DaVinci v1.0. It is also possible to add other configurations to further test DaVinci v1.0 with other programs.



```
// DA_VINCI v1.0 instance
defparam da_vinci_inst.mem_init_file = "fibonacci.dat";
//defparam da_vinci_inst.mem_init_file = "RevFib.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;

// TBD: rest of the test code goes here.

//# 20 $stop;
#5000   //$writememh("RevFib_mem_dump.dat", da_vinci_inst.memory_
        $writememh("fibonacci_mem_dump.dat", da_vinci_inst.memory
        $stop;
```

V. Conclusion

This project heavily focused on individual parts of computer architecture: the ALU, register file, memory, and control unit and their integration in order to successfully execute a program. In CS147 lectures, the concept of the computer system and data flow was taught without implementation. With a hands-on approach as done in this project, the concept of the computer system became more of a reality.

Additionally, the project required diving deeper into the logic level of the computer system and knowing how to issue the correct signals to obtain the correct results. Many of the problems encountered during implementation were a result of not fully understanding concepts related to this. For example, a problem encountered was due to overseeing that the result from register file read or ALU took an extra clock cycle. Overall, this project contributed to an understanding of the hardware design process, brought the concept of the computer system into reality, and being entirely conceptual, brought attention to the logical details of the computer system.

References

[1] K. Patra. CS 147. Class Lecture, Lecture 01. San Jose State University, San Jose, CA, March 31, 2017.
[2] K. Patra. CS 147. Class Lecture, Lecture 02. San Jose State University, San Jose, CA, April 4, 2017.