DaVinci v1.0m: Behavioral and Gate Level Model for Computer System Supporting CS147DV Instruction Set

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

***Abstract*— DaVinci v1.0m is a mixed 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. Unlike its predecessor DaVinci v1.0, this version implements a data path with special control signals has components such as register file and ALU implemented at the gate level. The following report documents the process and explains the requirements of the implementation of DaVinci v1.0m.**

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 v1.0m features a functional computer system with a 32-bit processor and a minimal 256MB memory. The standard computer components in DaVinci v1.0m 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 simulate the system, inform on the components of computer architecture, and successfully implement DaVinci v1.0m by implementation from small components to standard computer components.

II. REQUIREMENTS

The following section states the software needed and minimal instructions for system 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".

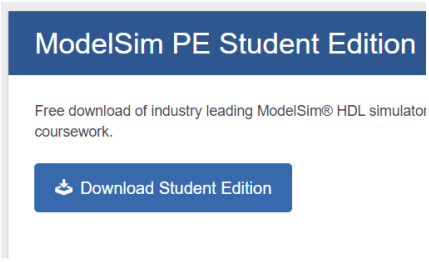

Fig. 1. Downloading ModelSim – Download button

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.

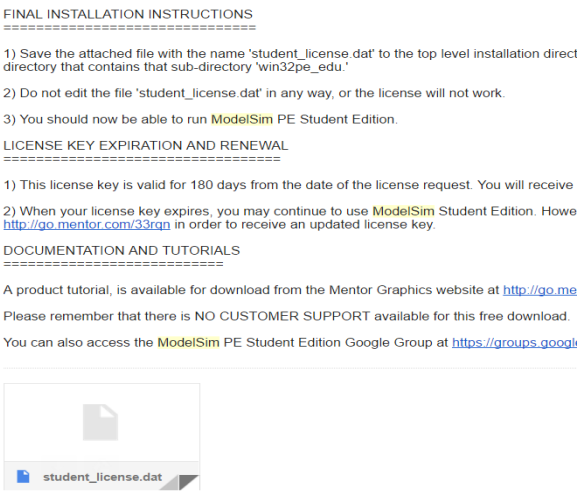

Fig. 2. Downloading ModelSim – License

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".

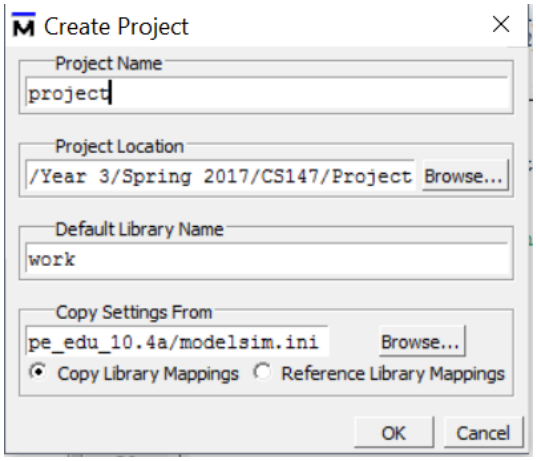

Fig. 3. Creating a project in ModelSim

3) Creating a simulation

Once the project has been created, select the project files excluding the test benches for the rest of the components such as the adders, multipliers, decoders, flip flops, etc. which are used for individual 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 two modules "DA_VINCI" and "DA_VINCI_TB" as shown in Figure 4. Once done, hit save.

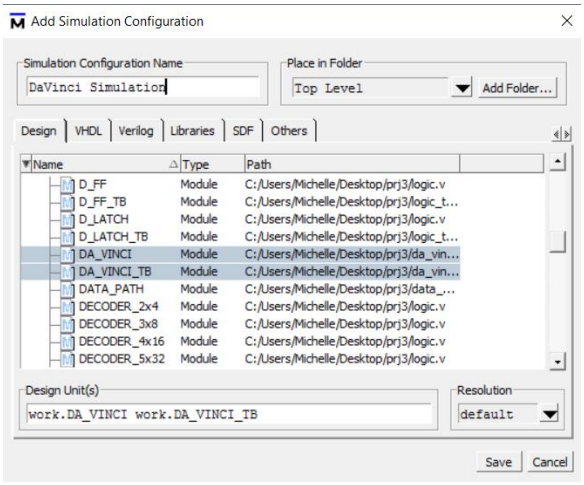

Fig. 4. Creating a simulation – Configuration window

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 in the test bench, "fibonacci_mem_dump" would be created with memory data 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 right click DA_VINCI_TB. In this window, go to Add to -> Wave -> All items in region.

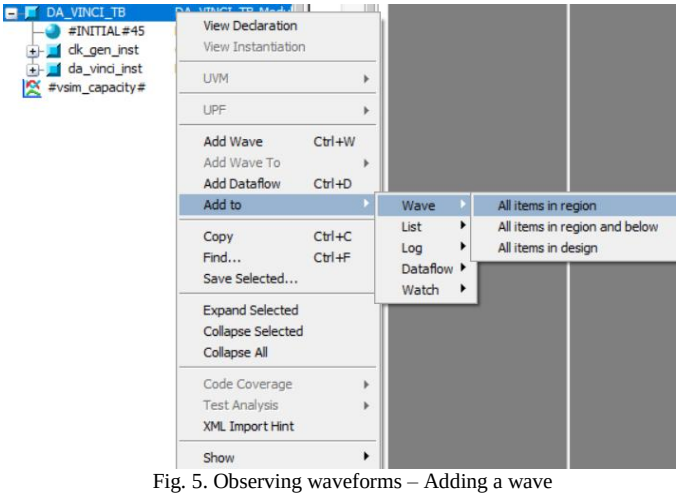

Fig. 5. Observing waveforms – Adding a wave

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 for memory data, memory address and read/write signals occurring at specific time intervals in picoseconds.

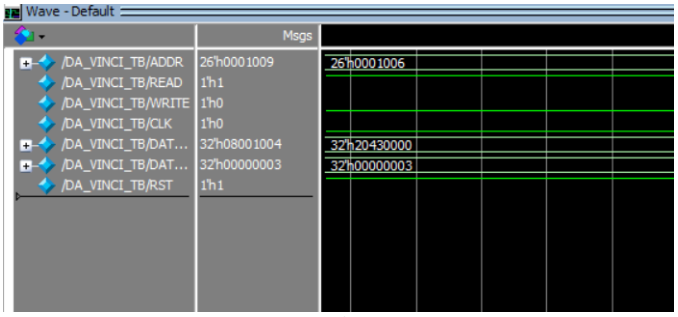

Fig. 6. Observing waveforms – Viewing the wave

B. Requirement for Computer System Model

DaVinci v1.0m follows the computer system model consisting of the ALU, memory, register file, control unit, and clock. The ALU and register file is assembled with basic logic gates such as AND, OR, XOR, NAND, NOT, etc. These simple logic gates build into larger components such as half adders, multipliers, multiplexers, decoders, etc. which are then used to build the computer components. Therefore, to understand the process of implementation of DaVinci v1.0m, it is important to understand the responsibility and requirements for every single component. The following section states descriptions for each component.

1) Binary Adder-Subtractor

The ALU handles many operations including the add and subtract operation. Within the ALU, the add and subtract

operations are done by the adder subtractor circuit. The 32-bit binary adder-subtractor is comprised of full adders connected together with each carry rippling to the next adder. The full adders consist of two half adders combined with the half adders having their own circuit from XOR and AND gate as shown in the following section.

i. Half Adder

When two bits are added and the result is a value greater than the max bit value of 1, the sum of the two bits is stored while a carry is sent to the next bit position. Table I shows the result when adding two bits A and B.

TABLE I
Truth table for half adder

A	B	SUM	CARRY
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Note that the truth table for the sum is identical to a truth table for XOR logic and the truth table for the carry is identical to the truth table for AND logic. Figure 7 shows the digital circuit for adding two bits A and B.

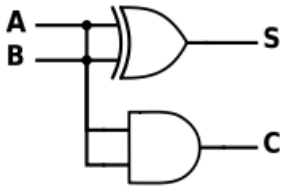

Fig. 7. Half adder circuit

ii. Full Adder

The half adder yields only the temporary result because it does not consider a carry-in. Combining two half adders result in a full adder. In another words, the first half-adder's carry-out result is the carry-in for the second half-adder. This carry-in is then added to the result of the current addition operation. The resulting carry-in becomes the carry out. Additionally, the OR operation is added, attaching the carry operations from the first half adder and the second half adder.

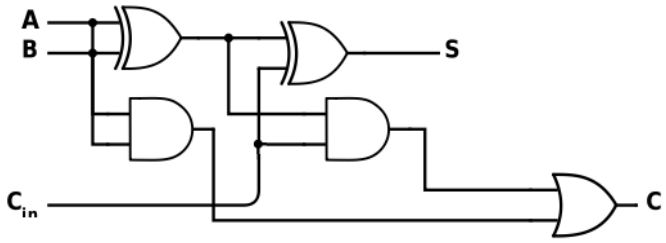

Fig. 8. Full adder circuit

By combining 32 full adders together and connecting the carry out from the previous adder to the carry in of the next, the result is the binary carry ripple adder as shown in Figure 9. For subtraction, the same circuit can be used. If given two values A and B to subtract, and the result wanted is A – B. The equivalent addition operation is A + ~B. Therefore, for the subtraction circuit, every single bit in B needs to be XOR'ed with the SnA signal which is set to 1 for the subtraction operation. The following binary carry ripple adder-subtractor shown in Figure 9 is the representation of all of the concepts combined.

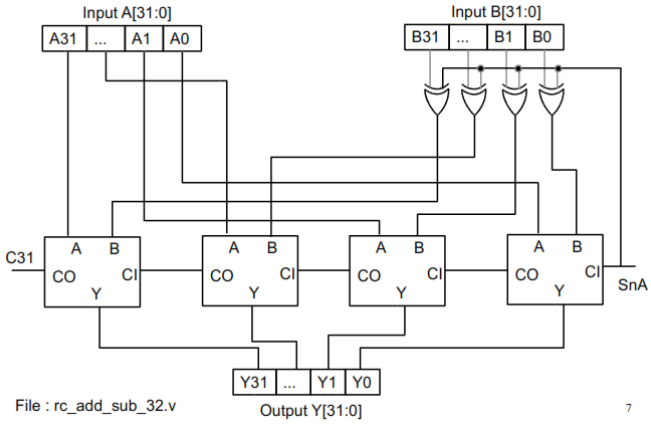

Fig. 9. Binary carry ripple adder-subtractor [1]

2) Decoder

The decoder is a basic digital component that gathers all outputs and sets one of the outputs to 1 based on the given input. As shown later, the decoder is necessary for building the multiplexer which is a decision making circuit. In brief, the multiplexer allows selection of a certain amount of inputs based on the control signal issued. For instance, in the ALU, the selection of the output from which operation circuit needs to occur since there is only one output result.

In the 2x4 decoder as shown in Figure 10, there are 2 inputs and 4 outputs. The decoder is designed such that only one of the 4 outputs will be turned on (set to logic 1) from the inputs A0 and A1.

TABLE II
Truth table for 2x4 Decoder

A1	A0	m0	m1	m2	m3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

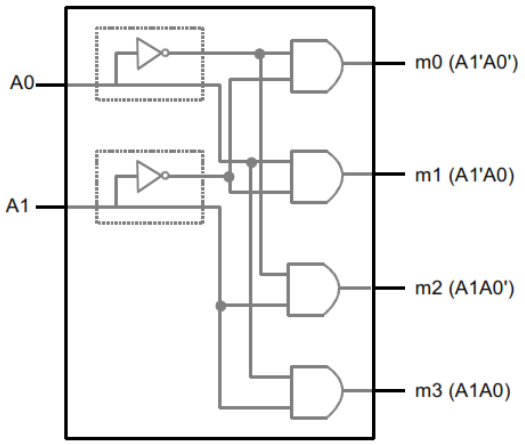

Fig. 10. 2x4 decoder circuit [2]

3) Multiplexer

The multiplexer selects only one output Y from the given inputs (I). The selection is determined by the selection bits (S). The bits for the signal depends on the number of inputs, that is, log base 2 of the given inputs. For a 4x1 multiplexer, there are 4 inputs and 1 output. Therefore, the number of bits for the selection bits is $\log_2 4 = 2$.

TABLE III
Truth table for 4x1 Multiplexer

S1	S0	Y
0	0	I0
0	1	I1
1	0	I2
1	1	I3

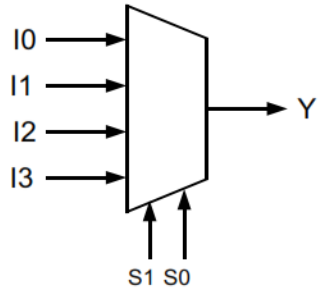

Fig. 11. General 4x1 multiplexer [2]

The multiplexer is built from the decoder, AND gates for each output from the decoder, and an OR gate to combine the outputs from the AND gates together to obtain a single final output.

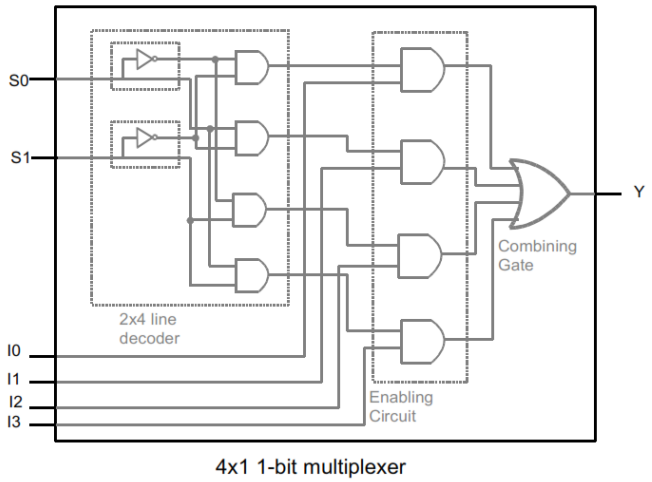

Fig. 12. 4x1 multiplexer circuit from decoder [2]

4) Multiplier

DaVinci v1.0m supports multiplication as a part of CS147DV instruction set which is implemented by the 32-bit multiplier. The multiplication circuit shown in Figure 13 is a combinatorial circuit that uses loop unrolling technique to save clock cycles unlike the sequential multiplier circuit. For multiplication, the AND result from a multiplier bit and multiplicand bit is computed to determine if the multiplicand is added to the final result. In the faster combinational multiplier, the result from the previous multiplication stage is input to the next adder. The carry out from the adder is inputted to the next adder along with 31 bits of the adder output. To produce a 64 bit result (32 bit in hi and 32 bit in lo register), the final adder output is replicated to produce the higher 32 bits.

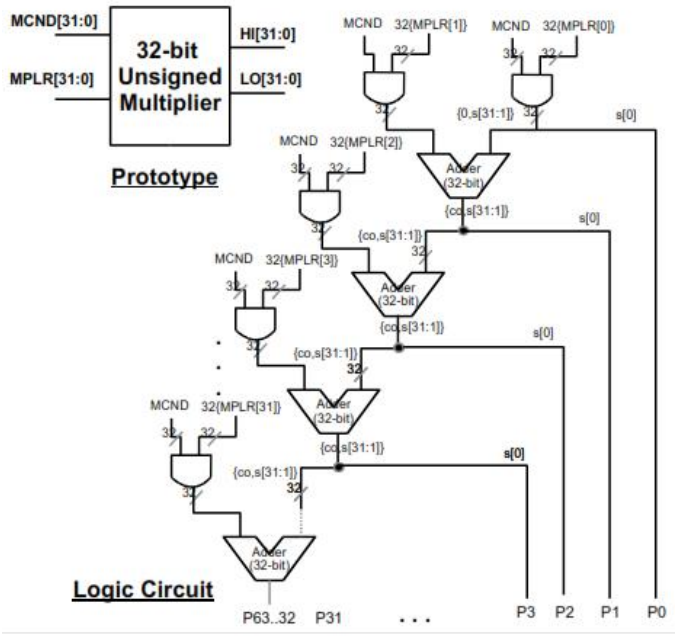

Fig. 13. Unsigned multiplier circuit [3]

There is also a need to take into account the resulting sign bit of the multiplication operation for an accurate signed result. The same multiplier circuit can be reused but the sign

of the multiplier and multiplicand must be converted into two's complement if negative before multiplying since the multiplier does not support signed multiplication. Whether the multiplicand or multiplier will be converted into two's complement depends on the MSB of their values. If the MSB is 1, the value is negative and must be converted into two's complement before multiplying. The output or product from the unsigned multiplier must be converted back to negative only if the either the multiplicand or multiplier were negative from the start, but not both. Therefore, the signed bit at the end is determined by XOR operation between the MSB of the multiplicand and the MSB of the multiplier.

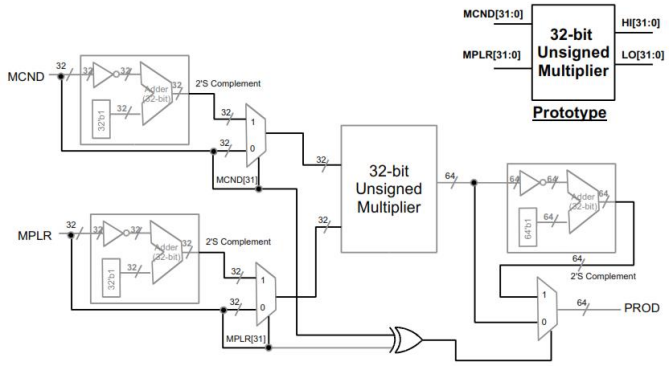

Fig. 14. 32-bit signed multiplier circuit [3]

5) Barrel Shifter

The shift left logical and shift right logical operations are also a part of the CS147DV instruction set is a part of the implementation of the ALU. The barrel shifter takes a shift amount as input and shifts the input several times depending on the bit positions of the binary form of the shift amount.

For example, a shift amount of 11 in binary is equivalent to 2+1 in binary. The left shifting operation can be split into 2 separate shift operations. Since the first bit is 1, a left shift of 1 is done first. Since the second bit position is 1, a left shift of 2 is done afterwards. The similar procedure applies for right shift as well. It is only the direction of shifting that differs.

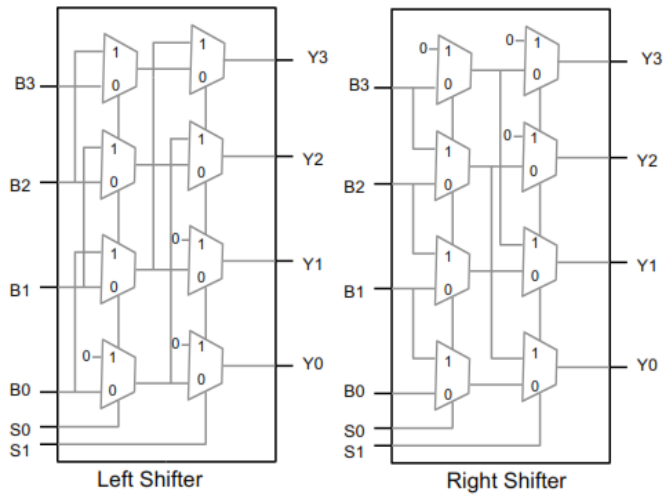

Fig. 15. Left and right shifters [4]

The left shifter and right shifter are combined together and selection is done using a multiplexer and a special LnR bit that determines whether the shift operation is a shift left or shift right. By combining the two shifters together and allowing selection, the circuit for universal barrel shifter is as follows.

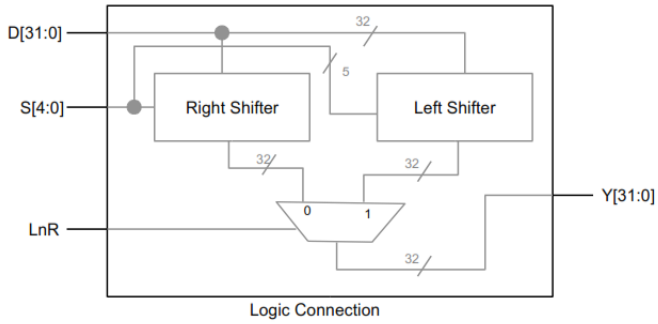

Fig. 16. Barrel shifter with left and right control [4]

In this implementation of a computer system, the shift amount inputted has a bit width of 32. Any shift amount that exceeds 5 bits automatically results in an output of zero since the maximum value of the input would be shifted out. If there is content in the rest of the shift amount, that is, if the value from 5th bit to 31st bit is greater than zero, the output will automatically be set to 32 bits of zero.

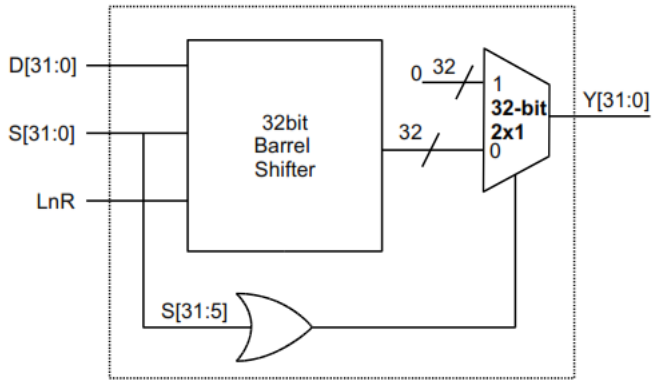

Fig. 17. Barrel shifter with 32-bit shift amount [4]

6) 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 v1.0m, the operation width of the computer is 32-bit, and thus, the number of bits for op1, op2, and the result is 32. There is also a zero flag which is turned on if the result from the ALU is equal to 0. The zero flag is used for the instructions branch if equal and branch not equal.

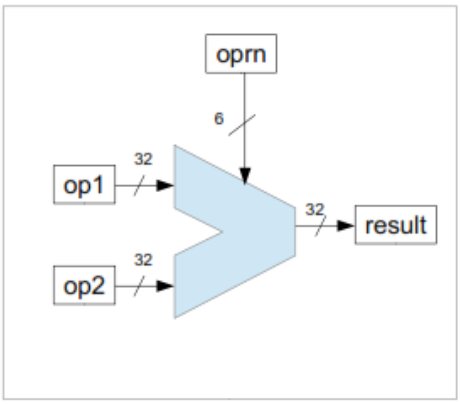

Fig.18. Schematic diagram representation of ALU [5]

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.19. Corresponding C code for ALU [5]

The operands are inputted to every component to perform multiplication, shift, add/sub, and, or, and nor operations. The ALU result is determined by the operation code signal which is used as the signal for the multiplexer that outputs the final result along with the zero flag result. The SnA input for the adder/subtractor and the LnR input for the shifter is also determined by the operation code.

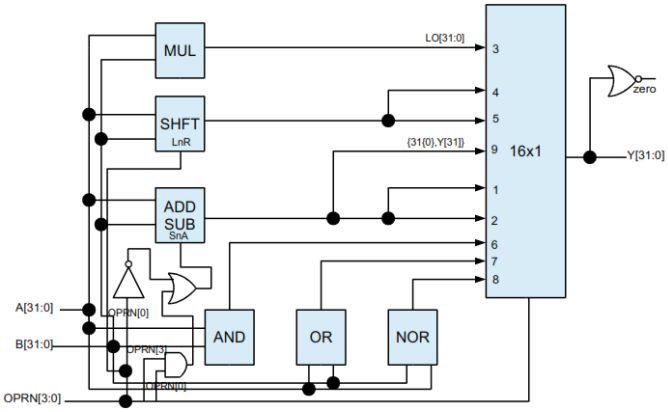

Fig.20. Gate level representation of the ALU [6]

7) SR Latch

The simplest storage element circuit is the set-reset latch or the SR latch. The SR latch can be implemented with either NOR gates or NAND gates but in this implementation, the NAND gate option is used.

The purpose of the SR latch is to provide stable storage for one bit, namely, Q_T. The next output Q_{T+1} can change or hold depending on the set and reset bits as shown in the following truth table for SR latch.

However, there is a problem in which the input values can change during computation time which will cause the output to become indeterministic. To resolve that issue, a control bit is introduced that holds on to the previous output if C=0. Otherwise, if C=1, operations remain as normal. Another issue with the SR latch is shown in the last line of the truth table where $Q_{T+1} = Q'_{T+1}$ which is undefined since it contradicts circuit operations. Fortunately, the D latch resolves this issue.

C	S	R	Q_T	Q_{T+1}	Q'_{T+1}
0	x	x	0	0	1
0	x	x	1	1	0
1	1	0	X	1	0
1	0	1	X	0	1
1	0	0	0	0	1
1	0	0	1	1	0
1	1	1	X	1	1

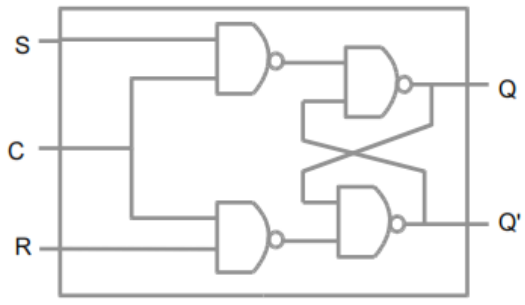

Fig.21. Schematic diagram of SR Latch [7]

8) D Latch

To avoid the undefined state that causes indeterministic circuit behavior, the D latch can be used in place of the SR

latch. Instead of having two bits set and reset that are complimentary to each other, a single bit can be used as an input. The other bit is produced by inverting that bit. With the D latch, the configurations of set = 1 and reset = 1 and set = 0 and reset = 0 are eliminated. The result is an eliminated possible indeterministic state and unnecessary hold configurations.

C	D	Q_T	Q_{T+1}	Q'_{T+1}
0	x	0	0	1
0	x	1	1	0
1	1	X	1	0
1	0	X	0	1

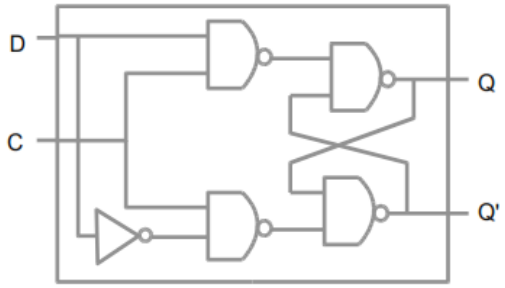

Fig.22. Schematic diagram of D Latch [7]

9) D Flip Flop

With the D latch, there is still a problem with the final output being indeterministic due to changes during computation. There is also no longer a hold for the previous output option for when C = 1 to store a stable storage element. The solution to the problem with D latch is the D flip flop which connects the D latch with the SR latch. With the D flip flop, the storage data is stable and the next computation of the storage element will not be in an indeterministic state.

Preset and reset signals for the D Flip Flop are used to set and reset the output which is similar to the preset and reset inputs for the latches. If preset is on and reset is off, the outputs for both latches will be 1. If preset is off and reset is on, the output for both will be 0. If they are both 1, operation is normal. It is important to note that with both signals are 0, there will be a race condition.

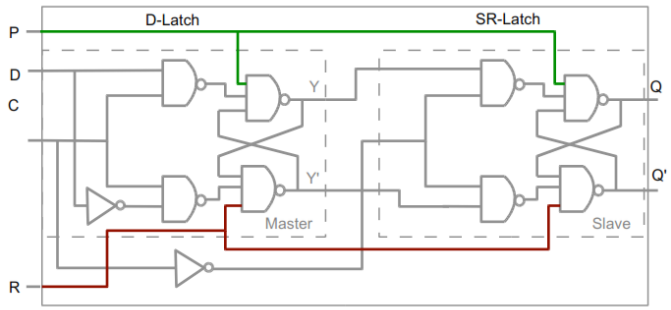

Fig.23. Schematic diagram of D Flip Flop [7]

The control signal of the D Flip Flop is the clock signal issued at intervals determined by the time period of the computer system. The two options for D Flip Flop triggered by the clock signal are negative edge and positive edge.

For the negative edge triggered D Flip Flop, the value is held when the clock is negative. When the clock is positive, the value changes depending on the input. On the other hand, for the positive edge triggered D Flip Flop, the clock is inverted at the beginning. Therefore, in this case, the opposite of the negative edge triggered D Flip Flop occurs. The value is held if the clock is positive and the value changes depending on input if the clock is negative. The difference of implementation is the extra addition of an inverter gate for the clock signal in the beginning for a positive edge triggered D Flip Flop.

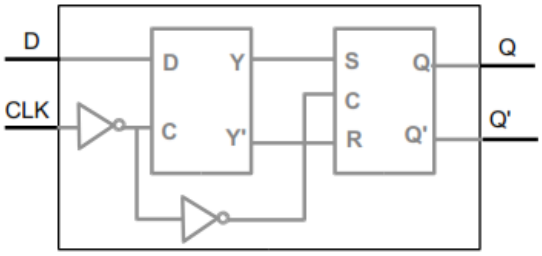

Fig.24. Positive edge triggered D Flip Flop [8]

10) Register

A one bit storage is a register and can be implemented using a single D Flip Flop. The input into the D latch of the D Flip Flop is either the previous output or the current input which is determined by a load signal using a multiplexer. If load is 0, Q is still stored in the register. However, if load is 1, the register value will change to the input and a new value will be loaded into the register.

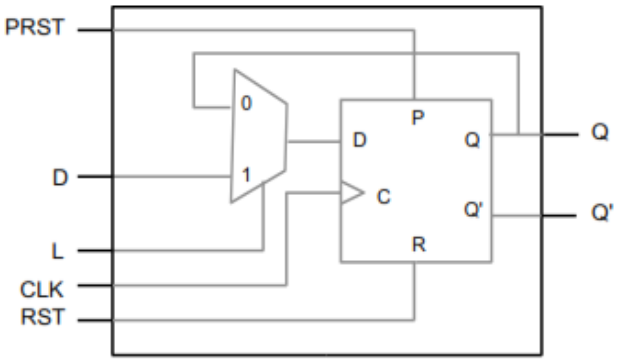

Fig.25. Diagram of a single bit register [7]

A 32-bit register can be implemented by replicating 32 of these 1-bit registers and using the same input for PRST, RST, D, and L as shown in Figure 26.

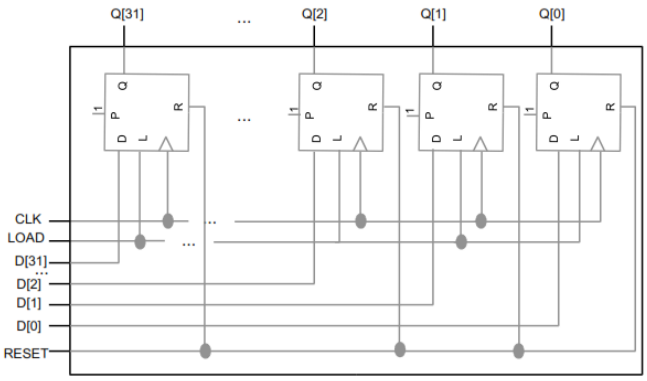

Fig.26. Diagram of a 32-bit register [7]

11) 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 access is frequently needed, so for parallelism, two addresses can be inputted at the same time to read from two addresses in the register file.

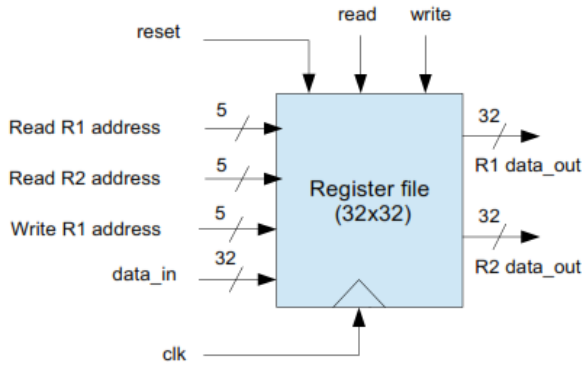

Fig.27. Schematic diagram of the register file [9]

The inputs into the 32 32-bit register files for writing to the register file are determined by input data and the write addresses. A 5x32 line decoder is used to determine the outputs from the write address to r1 and the load signal is generated from the AND operation between the write address and the write signal. If the write signal is on, the chosen register will have the load signal of 1, meaning, the data will be written into that register. The inputs for data in (W_DATA), clock (CLK), and reset (RST) remain the same for every register.

All of the outputs from the flops in the individual registers are input for two multiplexers. The output from the two multiplexers are selected by the address of R1 (R1_ADDR) and R2 (R2_ADDR) so that the register needed to be read from is selected as output from the register file. That result goes into separate multiplexers that go directly to the data out ports depending if the read signal is on. Otherwise, if the read signal is off, data out is in a hi-Z state which is our desired result for a non-read operation.

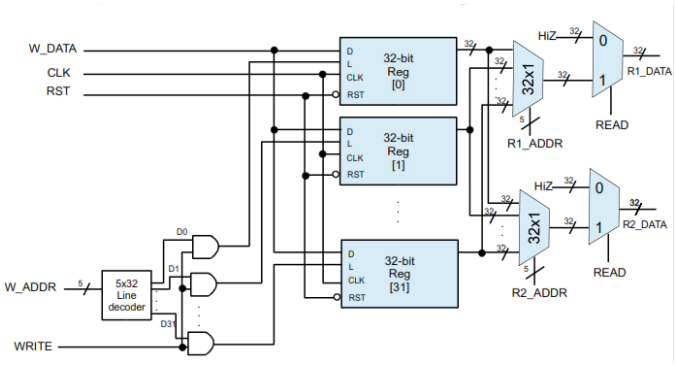

Fig.28. Gate level representation of register file [10]

12) 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 addressable memory. In DaVinci v1.0m, the size of the memory is 256MB and the address width is 26.

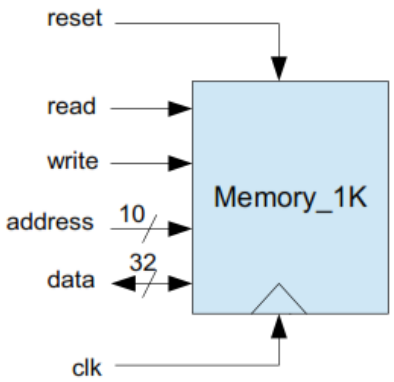

Fig.29. Schematic diagram of the memory [9]

13) Data Path

To execute any of the instructions from CS147DV instruction set, the instruction from the PC must flow through the circuit to retrieve data from the register file needed for the instruction. The data path shows the potential paths from each component such as the PC register, SP register, instruction memory, register file, ALU, and data memory. Data retrieval or distribution is determined by the control unit which issues the signals to the multiplexers depending on the stage of the instruction.

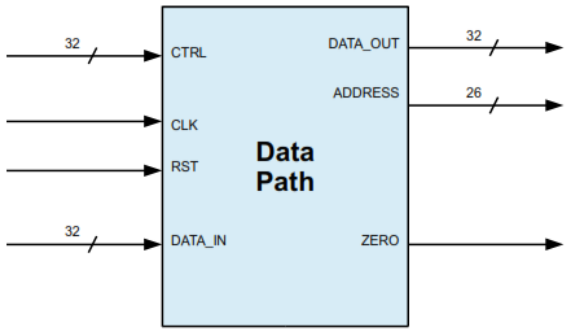

Fig.30. Schematic diagram of the data path [11]

For most operations, the register file and ALU is needed. Instructions such as jump (to an address), lui, etc. do not require the same data retrieval and flow as the rest of the instructions. Therefore, the data path must be implemented to account for the differences in data path for each type of instruction by using a multiplexer for data path conflicts.

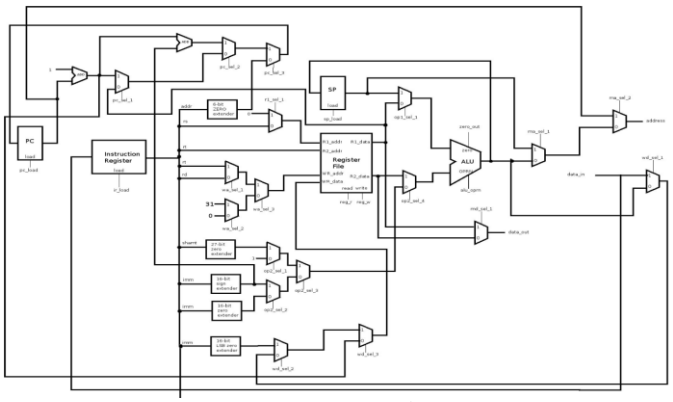

Fig.31. Gate level representation of the data path [11]

i. R-Type Instruction Data Path

A general R-Type instruction in the CS147DV instruction set reads from the address of rs and rt from the register file so the parsed instruction is connected to the read and write addresses of the register file. The resulting data outputs from the register file are connected to the input operands of the ALU which performs an operation on the two operands. The ALU output is connected to the write data in the register file.

An exception to the R-Type instruction is the shift left logical and shift right logical operations. Instead of accessing R[rt], the second operand input for the ALU is determined by the shift amount. The shift amount as a part of the instruction is extended to a 32-bit shift amount and used instead of R[rt] as input to the ALU.

Instead of being connected to the input of the ALU, for the jump register instruction of the same type, the value of R[rs] is used as input to the program counter register. Aside from the differences for shift and jump register instructions, the R-type instructions share almost the same data paths with each other.

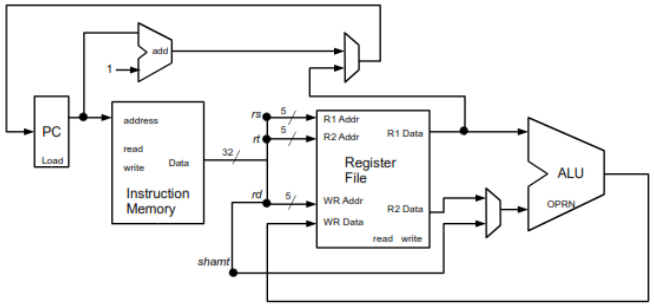

Fig.32. Data path for R-type instructions [12]

ii. I-Type Instruction Data Path

The I-Type instructions can be broken into five categories: logical, arithmetic, memory involvement, LUI, and branch. For logical and arithmetic I-Type instructions, only the extension of the immediate is different. As with the R-Type instruction, these two categories of I-Type read from the register file and use the retrieved data as input into the ALU. Then, the ALU output is used as the input for the write data for the register file. Instead of reading from two addresses, however, one of the operands for the ALU is directly from the immediate from the instruction. For the arithmetic I-Type, the immediate is sign-extended whereas for the logical I-Type, the immediate is zero-extended instead. The LUI instruction does not use the ALU at all and only uses the same zero-extender to extend the immediate as direct input into the write data input of the register file.

The load word and store word instructions follow the same concept as the arithmetic I-Type of adding a value in the register, namely R[rs] to a signed extended immediate. Instead of writing back the ALU output to the register file, however, the ALU output is connected to the read address for the load word instruction and to the write address for the store word instruction. Additionally, in the store word instruction, the data output from reading from address of rt is connected to the write data input of the memory. In the load word instruction, the memory data output is written back into the register file so it is connected to the write data input of the register file.

After adding the PC value to 1, the branch instructions add the immediate value obtained from the instruction on top of that. As with the majority of the I-Type instructions, the immediate added to the PC is sign extended.

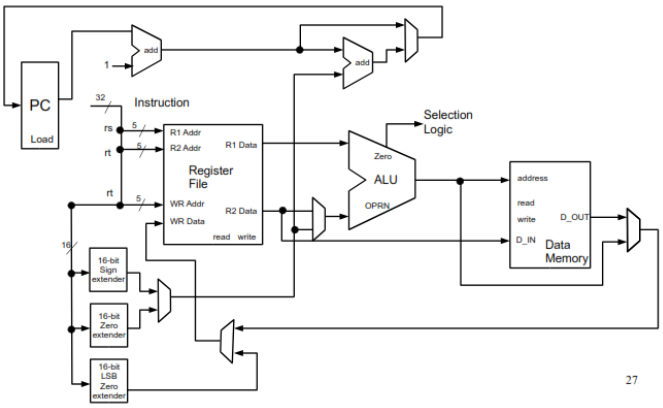

Fig.33. Data path for I-type instructions [12]

iii. J-Type Instruction Data Path

J-Type instructions typically involve updating the PC or SP by adding to them. Some of the J-Type instructions such as jump and link and the stack operations write back into the register file and/or update the memory.

As with the branch instruction (I-Type), the jump updates the program counter by adding directly the address provided in the instruction. For the jump instruction, the address is zero-extended. The jump and link instruction performs the same way but stores the previous PC value into the register file at the address of 31.

The stack operations push and pop behave similar to store word and load word, respectively. The differences are that a different section of memory (stack) is accessed and the stack pointer which contains the address of the top of the stack is updated after push and pop instructions. The data to be read/write into the stack is the value of R[0]. Thus, the register file must be accessed accordingly. After reading or writing to the stack, the SP must be incremented or decremented using the ALU in reverse of each other depending on the implementation.

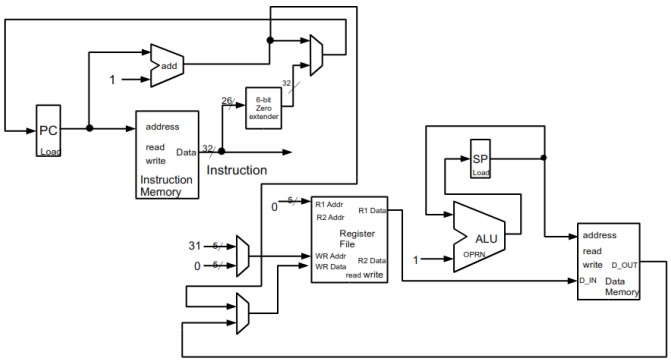

Fig.34. Data path for J-Type instructions [12]

14) Control Unit and Processor

The control unit issues the signals for the mentioned data path. The stages of the control unit are shown in Figure 35. 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.

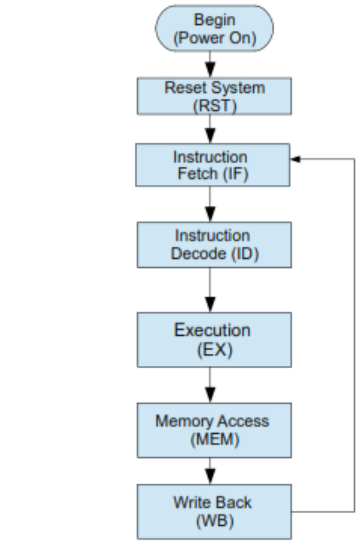

Fig.35. Control System Stages [9]

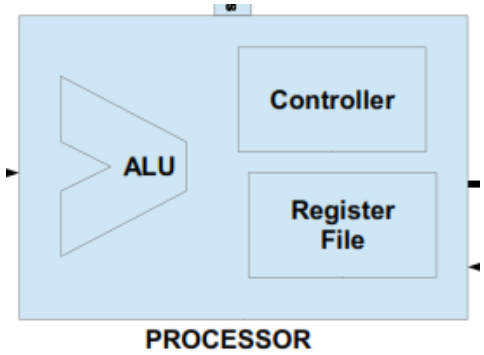

Fig.36. Processor [9]

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.

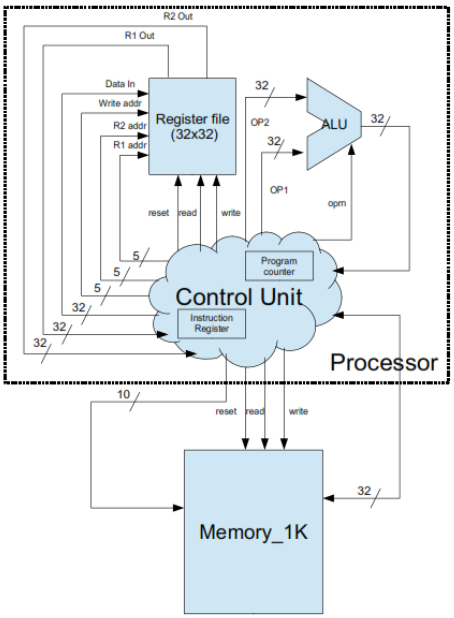

Fig.37. Control unit [9]

15) 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 v1.0m, 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) half_adder.v

The "half_adder.v" file declares single bit width ports for outputs Y and C and inputs A and B. Following the stated truth table and Boolean expression for half adder, the XOR is instantiated to set Y, single bit addition result to A ⊕ B and the A.B is computed using an AND gate for the carry bit result in C.

```
module HALF_ADDER(Y,C,A,B);
output Y,C;
input A,B;

// TBD
xor inst1(Y, A, B);
and inst2(C, A, B);

endmodule;
```

3) full_adder.v

The full adder implementation in "full_adder.v" declares single bit width ports for the result output bit Y, carry out output bit CO, and input bits A, B, CI that are added together for the result.

For the full adder, the Boolean expression to implement by instantiating the corresponding logic gates for Y is $Y = A \oplus B \oplus C$ and the expression for the carry out is $CO = CI.(A \oplus B) + A.B$.

```
`include "prj_definition.v"

module FULL_ADDER(Y,CO,A,B, CI);
output Y,CO;
input A,B, CI;

xor a_xor_b(C, A, B);
xor inst1(Y, CI, C);
and first_term(FT, CI, C);
and second_term(ST, A, B);
or inst2(CO, FT, ST);

endmodule;
```

4) rc_add_sub_32.v

In the RC_ADD_SUB_32 module, the output list and input list ports are declared in the beginning of the file. Since the adder-subtractor is 32-bit, the output Y and inputs A and B are 32-bit or the data index wide. The output CO is a single bit and the SnA signal is a single bit signal.

The carry out from each of the 32 full adders are stored into individual wires. To account for the subtraction operation for this circuit, the XOR result from ith bit of the second input B and SnA is stored into another wire. Each of the full adders are connected to the ith bit of A, stored result of the XOR operation, and the carry out of the previous full adder is connected to the carry in of the next full adder.

```
module RC_ADD_SUB_32(Y, CO, A, B, SnA);
// output list
output [`DATA_INDEX_LIMIT:0] Y;
output CO;
// input list
input [`DATA_INDEX_LIMIT:0] A;
input [`DATA_INDEX_LIMIT:0] B;
input SnA;

// TBD
wire CO_0b;
xor add_sub_0(as0, B[0], SnA);
FULL_ADDER full_adder_0(.Y(Y[0]), .CO(CO_0b), .A(A[0]), .B(as0), .

wire CO_1b;
xor add_sub_1(as1, B[1], SnA);
FULL_ADDER full_adder_1(.Y(Y[1]), .CO(CO_1b), .A(A[1]), .B(as1), .

wire CO_2b;
xor add_sub_2(as2, B[2], SnA);
FULL_ADDER full_adder_2(.Y(Y[2]), .CO(CO_2b), .A(A[2]), .B(as2), .

wire CO_3b;
xor add_sub_3(as3, B[3], SnA);
FULL_ADDER full_adder_3(.Y(Y[3]), .CO(CO_3b), .A(A[3]), .B(as3), .

wire CO_4b;
xor add_sub_4(as4, B[4], SnA);
FULL_ADDER full_adder_4(.Y(Y[4]), .CO(CO_4b), .A(A[4]), .B(as4), .

wire CO_5b;
xor add_sub_5(as5, B[5], SnA);
FULL_ADDER full_adder_5(.Y(Y[5]), .CO(CO_5b), .A(A[5]), .B(as5), .

wire CO_6b;
```

5) mux.v

There are several multiplexers of different input and output widths such as 2x1, 4x1, 8x1, 16x1, and 32x1. All of the multiplexers are created by instantiating many 1-bit 2x1 multiplexers as follows. The Boolean expression used to implement the 1-bit 2x1 multiplexer is Y = S'I0 + SI1. Instantiations for not, and, and occurred to achieve the result for the output bit Y.

```
// 1-bit mux
module MUX1_2x1(Y,I0, I1, S);
//output list
output Y;
//input list
input I0, I1, S;

// TBD
not not_s(NS, S);
and and_inst1(Y1, NS, I0);
and and_inst2(Y2, S, I1);
or or_inst(Y, Y1, Y2);

endmodule
```

To obtain a 32-bit 2x1 multiplexer, the same 1-bit 2x1 multiplexer must be generated for every single bit. Instead of instantiating 32 individual multiplexers, it is more efficient to use a generate statement since the wiring for every multiplexer is similar, using the ith bit of the input and output wires of this multiplexer.

```
// 32-bit mux
module MUX32_2x1(Y, I0, I1, S);
// output list
output [`DATA_INDEX_LIMIT:0] Y;
//input list
input [`DATA_INDEX_LIMIT:0] I0;
input [`DATA_INDEX_LIMIT:0] I1;
input S;

// TBD
genvar i;
generate
        for(i=0; i<32; i=i+1)
        begin: mux32_gen_loop
                MUX1_2x1 mux2x1_inst(Y[i], I0[i], I1[i], S);
        end
endgenerate
endmodule
```

To obtain a 32-bit 4x1 multiplexer which is a 32-bit 2x1 multiplexer with twice the amount of input, combine three multiplexers together. Two of the multiplexers use the same selection bits to select between I0/I1 and I2/I3 input. From those selections or the output from the two multiplexers, a final selection is determined by the last bit of the selection input and using another multiplexer.

The 32-bit 8x1 multiplexer was implemented using the same mechanism of combining multiplexers to obtain a final selection output in Y. In the same way, the 8x1 multiplexer was used to build the 16x1, and the 16x1 multiplexer was used to build the 32x1 multiplexer.

```
// 32-bit 4x1 mux
module MUX32_4x1(Y, I0, I1, I2, I3, S);
// output list
output [`DATA_INDEX_LIMIT:0] Y;
//input list
input [`DATA_INDEX_LIMIT:0] I0;
input [`DATA_INDEX_LIMIT:0] I1;
input [`DATA_INDEX_LIMIT:0] I2;
input [`DATA_INDEX_LIMIT:0] I3;
input [1:0] S;

// TBD
wire [`DATA_INDEX_LIMIT:0] mux_1a_out1;
wire [`DATA_INDEX_LIMIT:0] mux_1b_out1;
MUX32_2x1 mux32_inst_1a(mux_1a_out1, I0, I1, S[0]);
MUX32_2x1 mux32_inst_1b(mux_1b_out1, I2, I3, S[0]);

MUX32_2x1 mux32_inst_1(Y, mux_1a_out1, mux_1b_out1, S[1]);

endmodule
```

6) mult.v

In the "mult.v" file, the input and output ports are declared for HI and LO outputs and A and B (multiplicand, multiplier) inputs. Because in multiplication, the product result is constantly being added to by new input, the SnA register is also initialized to 0 to set the adder subtractor to add mode.

Following the circuit as discussed, every bit of the multiplier is AND'ed with the multiplicand and stored into a named wire to figure out if there is a value to add. The wire from the AND result is then used as input for the next adder. The other input for the adder is comprised of the carry out from the previous addition as the MSB and the addition result.

```
module MULT32_U(HI, LO, A, B);
// output list
output [`DATA_INDEX_LIMIT:0] HI;
output [`DATA_INDEX_LIMIT:0] LO;
// input list
input [`DATA_INDEX_LIMIT:0] A; // MCND
input [`DATA_INDEX_LIMIT:0] B; // MPLR

// TBD
reg SnA = 1'b0;

wire [`DATA_INDEX_LIMIT:0] and_1a;
wire [`DATA_INDEX_LIMIT:0] and_1b;
AND32_2x1 and_inst_1a(and_1a, A, {32{B[1]}});
AND32_2x1 and_inst_1b(and_1b, A, {32{B[0]}});
wire CO_1;
wire [`DATA_INDEX_LIMIT:0] Y1;
RC_ADD_SUB_32 adder1(.Y(Y1), .CO(CO_1), .A(and_1a), .B({1'b0, and_1b[31:1]}), .SnA

wire [`DATA_INDEX_LIMIT:0] and_2;
AND32_2x1 and_inst_2(and_2, A, {32{B[2]}});
wire CO_2;
wire [`DATA_INDEX_LIMIT:0] Y2;
RC_ADD_SUB_32 adder2(.Y(Y2), .CO(CO_2), .A(and_2), .B({CO_1,Y1[31:1]}), .SnA(SnA))

wire [`DATA_INDEX_LIMIT:0] and_3;
AND32_2x1 and_inst_3(and_3, A, {32{B[3]}});
wire CO_3;
```

The least significant bits of all of the addition results are finalized as the bits in the LO result of multiplication. The entire 32-bit result from the final addition becomes the entire HI result of multiplication.

```
RC_ADD_SUB_32 adder30(.Y(Y30), .CO(CO_30), .A(and_30), .B({CO_29,Y29[31:1]}), .S

wire [`DATA_INDEX_LIMIT:0] and_31;
AND32_2x1 and_inst_31(and_31, A, {32{B[31]}});
wire CO_31;
wire [`DATA_INDEX_LIMIT:0] Y31;
RC_ADD_SUB_32 adder31(.Y(Y31), .CO(CO_31), .A(and_31), .B({CO_30,Y30[31:1]}), .S


assign LO = {Y31[0], Y30[0], Y29[0], Y28[0], Y27[0], Y26[0], Y25[0], Y24[0], Y23
             Y20[0], Y19[0], Y18[0], Y17[0], Y16[0], Y15[0], Y14[0], Y13[0],
             Y10[0], Y9[0], Y8[0], Y7[0], Y6[0], Y5[0], Y4[0], Y3[0], Y2[0],

assign HI = Y31;

endmodule
```

The two's complement module needs to be implemented for the purpose of signed multiplication. After initializing the output port for Y and the input port for A, the input A is inverted by instantiating an inverter. Finally, the inverted result ~A must be added by 1. To do that, an adder-subtractor is instantiated and set to addition mode by initializing SnA to 0.

```
// 32-bit two's complement
module TWOSCOMP32(Y,A);
//output list
output [`DATA_INDEX_LIMIT:0] Y;
//input list
input [`DATA_INDEX_LIMIT:0] A;

// TBD
wire [`DATA_INDEX_LIMIT:0] a_inv;
INV32_1x1 inv_32_inst(a_inv, A);
wire CO;
reg SnA = 1'b0;
RC_ADD_SUB_32 adder1(.Y(Y), .CO(CO), .A(a_inv), .B(32'b1), .SnA(SnA));

endmodule
```

To implement the signed multiplier, two twos complement instances are created to compute the twos complement for both A and B. The two's complement version is selected for A and B only if the value is negative. The selection is done by a multiplexer and the selection signal is determined by the most significant bit (sign bit) of the value. The multiplexer outputs are input for the unsigned multiplier.

To ensure that the unsigned multiplier product's sign is correct, the sign of the product is determined using XOR on the most significant bits of the A and B. The two's complement of the result HI and LO registers is computed using the exact same method of selection by inputting the sign bit into a multiplexer. If the resulting sign bit is negative, the output for both HI and LO is in two's complement form.

```
wire [`DATA_INDEX_LIMIT:0] twos_mcnd;
wire [`DATA_INDEX_LIMIT:0] twos_mplr;

TWOSCOMP32 inst_twos_mcnd(.Y(twos_mcnd), .A(A));
TWOSCOMP32 inst_twos_mplr(.Y(twos_mplr), .A(B));

wire [`DATA_INDEX_LIMIT:0] MCND;
wire [`DATA_INDEX_LIMIT:0] MPLR;

MUX32_2x1 inst_mux_1(.Y(MCND), .I0(A), .I1(twos_mcnd), .S(A[31]));
MUX32_2x1 inst_mux_2(.Y(MPLR), .I0(B), .I1(twos_mplr), .S(B[31]));

wire [`DATA_INDEX_LIMIT:0] HI_RES;
wire [`DATA_INDEX_LIMIT:0] LO_RES;

MULT32_U inst_uns_mult(.HI(HI_RES), .LO(LO_RES), .A(MCND), .B(MPLR));

wire [`DATA_INDEX_LIMIT:0] inv_hi;
wire [`DATA_INDEX_LIMIT:0] twos_lo;

assign HI_RES = (HI_RES === {`DATA_WIDTH{1'bz}})?{32'b0}:HI_RES;

INV32_1x1 inst_inv_hi(.Y(inv_hi), .A(HI_RES));
TWOSCOMP32 inst_twos_lo(.Y(twos_lo), .A(LO_RES));

wire result_sign;

xor xor_result_sign(result_sign, A[31], B[31]);

MUX32_2x1 inst_mux_prod1(.Y(HI), .I0(HI_RES), .I1(inv_hi), .S(result_sign))
MUX32_2x1 inst_mux_prod2(.Y(LO), .I0(LO_RES), .I1(twos_lo), .S(result_sign)
```

7) barrel_shifter.v

The individual shifters (left and right) are implemented in the "barrel_shifter.v" file. The ports for the left shifter for the input value D, input shift amount D, and output Y are declared. 32 multiplexers are generated for every bit of the shift amount. The code is grouped into columns or the bit position of the shift amount. In the following code snippet, 32 multiplexers are generated for the first bit position (0) to shift the input by 1 if the selection bit S[0] is 1. The second column will perform in a similar way but the bits will shift by $2^1 = 2$ and so on.

```
// Left shifter
module SHIFT32_L(Y,D,S);
// output list
output [`DATA_INDEX_LIMIT:0] Y;
// input list
input [`DATA_INDEX_LIMIT:0] D; // B = D
input [4:0] S; //S0 TO S4

// TBD
// Ea column will have 32 multiplexer
// 5 such columns

wire [`DATA_INDEX_LIMIT:0] out1;

// Column 1 (shift 1)
genvar i1;
generate
    for(i1=0; i1<32; i1=i1+1)
    begin: barrelshift1_gen_loop
        if (i1 == 0) begin
            MUX1_2x1 mux2x1_inst(out1[i1], D[i1], 1'b0, S[0]);
        end
        else begin
            MUX1_2x1 mux2x1_inst(out1[i1], D[i1], D[i1-1], S[0]);
        end
    end
endgenerate

wire [`DATA_INDEX_LIMIT:0] out2;

// Column 2 (shift 2)
```

The barrel shifter implemented allows selection of the output of the left and right shifters based on the LnR input bit. If LnR = 0, the right shift output is selected. Else, the left shift output is selected as the final result.

```
// Shift with control L or R shift
module BARREL_SHIFTER32(Y,D,S, LnR);
// output list
output [`DATA_INDEX_LIMIT:0] Y;
// input list
input [`DATA_INDEX_LIMIT:0] D;
input [4:0] S;
input LnR;

// TBD

wire [`DATA_INDEX_LIMIT:0] outR;
wire [`DATA_INDEX_LIMIT:0] outL;

SHIFT32_R inst_rshift(.Y(outR), .D(D), .S(S));
SHIFT32_L inst_lshift(.Y(outL), .D(D), .S(S));

// Right shift if LnR = 0
MUX32_2x1 inst_mux_32bit(.Y(Y),.I0(outR), .I1(outL), .S(LnR));

endmodule
```

To extend the shift amount to 32-bit instead of 5-bit, the barrel shifter instantiation is as normal with the first 5 bits of the shift amount used as input. With any shift amount that uses the rest of the bits, the result will be zero. That can be implemented using a multiplexer with the selection bit being 0 or 1 depending if the higher bits of the shift amount is equal to 0.

```
// 32-bit shift amount shifter
module SHIFT32(Y,D,S, LnR);
// output list
output [`DATA_INDEX_LIMIT:0] Y;
// input list
input [`DATA_INDEX_LIMIT:0] D;
input [`DATA_INDEX_LIMIT:0] S;
input LnR;

// TBD
wire [`DATA_INDEX_LIMIT:0] shift_out;
BARREL_SHIFTER32 inst_barrel_shifter32(.Y(shift_out), .D(D), .S(S[4:0]), .LnR(Ln

MUX32_2x1 inst_mux_32bit(.Y(Y),.I0(32'b0), .I1(shift_out), .S((S[31:5] == 0)));

endmodule
```

8) alu.v

The "alu.v" file creates a module or a design of the ALU providing a way of communication between 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 A, B, and OPRN are input. Only the result Y and zero port need to be specified as output.

```
`include "prj_definition.v"
module ALU(A, B, OPRN, Y, ZERO);
// output list
output [`DATA_INDEX_LIMIT:0] Y;
output [`DATA_INDEX_LIMIT:0] ZERO;
// input list
input [`DATA_INDEX_LIMIT:0] A;
input [`DATA_INDEX_LIMIT:0] B;
input [5:0] OPRN;

wire CO;
wire SnA;
```

In the ALU, the following components are instantiated: adder/subtractor, multiplier, barrel shifter with left and right

shift control, and, or, and not. All components must support 32-bit width of data and a 32 bit wire is created for every operation result and assigned to the results of the components. For the adder/subtractor and the barrel shifter, the controls SnA and LnR are determined by the operation codes given as input.

```
// Set SnA to OPRN[0]' + OPRN[3]OPRN[0]
not not_oprn0 (not_res, OPRN[0]);
and and_oprn13 (and_res, OPRN[0], OPRN[3]);
or or_res(SnA, not_res, and_res);

// Add and Sub
wire [`DATA_INDEX_LIMIT:0] add_sub_out;
RC_ADD_SUB_32 inst_add_32(.Y(add_sub_out), .CO(CO), .A(A), .B(B), .SnA(SnA)

// Mul
wire [`DATA_INDEX_LIMIT:0] mul_hi_out;
wire [`DATA_INDEX_LIMIT:0] mul_lo_out;
MULT32 inst_mult_32(.HI(mul_hi_out), .LO(mul_lo_out), .A(A), .B(B));

// Shifter
wire [`DATA_INDEX_LIMIT:0] shift_out;
SHIFT32 inst_shiftLR_32bit(.Y(shift_out), .D(A), .S(B), .LnR(OPRN[0]));

// And
wire [`DATA_INDEX_LIMIT:0] and_out;
AND32_2x1 and_32_bit_inst(.Y(and_out),.A(A),.B(B));

// Or
wire [`DATA_INDEX_LIMIT:0] or_out;
OR32_2x1 or_32_bit_inst(.Y(or_out),.A(A),.B(B));

// Nor
```

All of the results are inputted to the multiplexer whose result is selected by the operation code and stored in Y. Another result is the zero flag, which is determined by using a NOR gate for the result and 32 bit width of zero.

```
MUX32_16x1 oprn_select (Y, {`DATA_WIDTH{1'bx} }, add_sub_out, add_sub_out, mul_lo_out, sl
                        or_out, nor_out, {31'b0, add_sub_out[31]}, {`DATA_WIDTH{1'bx} },
                        {`DATA_WIDTH{1'bx} }, {`DATA_WIDTH{1'bx} }, {`DATA_WIDTH{1'bx} }

wire [`DATA_INDEX_LIMIT:0] zero_flag_result;

NOR32_2x1 zero_flag(.Y(zero_flag_result), .A(Y), .B({`DATA_WIDTH{1'b0} }));

assign ZERO = (zero_flag_result == {`DATA_WIDTH{1'b1} });

endmodule
```

9) logic.v

i. Latches and D Flip Flop

The SR_LATCH module begins with declaring the inputs set, reset, control, preset, reset and the outputs Q and Q'. Four NAND gates are instantiated to compute the NAND result for inputs as shown in the Figure 21 in the requirements section. The order of declaring the ports and instantiating logic gates that follow the circuit shown in the requirements section of this report apply to D Latch and D Flip Flop in the following code snippet.

```
// 1 bit SR latch
// Preset on nP=0, nR=1, reset on nP=1, nR=0;
// Undefined nP=0, nR=0
// normal operation nP=1, nR=1
module SR_LATCH(Q,Qbar, S, R, C, nP, nR);
input S, R, C;
input nP, nR;
output Q,Qbar;

wire sr_out_1, sr_out_2;

// TBD
nand nand_sr_1(sr_out_1, S, C);
nand nand_sr_2(sr_out_2, R, C);
nand nand_sr_3(Q, sr_out_1, Qbar, nP);
nand nand_sr_4(Qbar, sr_out_2, Q, nR);

endmodule

// 1 bit D latch
// Preset on nP=0, nR=1, reset on nP=1, nR=0;
// Undefined nP=0, nR=0
// normal operation nP=1, nR=1
module D_LATCH(Q, Qbar, D, C, nP, nR);
input D, C;
input nP, nR;
output Q,Qbar;

// TBD
wire Dbar;
not not_d(Dbar, D);
SR_LATCH inst_dlatch_sr(.Q(Q), .Qbar(Qbar), .S(D), .R(Dbar), .C(C), .nP(nP), .nR(nR)

endmodule

// 1 bit flipflop +ve edge,
// Preset on nP=0, nR=1, reset on nP=1, nR=0;
// Undefined nP=0, nR=0
// normal operation nP=1, nR=1
module D_FF(Q, Qbar, D, C, nP, nR);
input D, C;
input nP, nR;
output Q,Qbar;

not inv_c(nC, C);

// TBD
wire Y, Ybar;
D_LATCH inst_d_latch(.Q(Y), .Qbar(Ybar), .D(D), .C(nC), .nP(nP), .nR(nR));

//not inv_inv_c(nnC, nC);

SR_LATCH inst_sr_latch(.Q(Q), .Qbar(Qbar), .S(Y), .R(Ybar), .C(C), .nP(nP), .nR(nR)

endmodule
```

ii. Register (1-bit)

By using the D Flip Flop as in instantiating it in REG1 module, a 1 bit register can be implemented. The input into the D Flip Flop can be either the current input or the previous output depending on the load signal into the register. To implement the selection, a 2x1 multiplexer is instantiated to select between Q and D using the register input load signal.

```
// 1 bit register +ve edge
// Preset on nP=0, nR=1, reset on nP=1, nR=0;
// Undefined nP=0, nR=0
// normal operation nP=1, nR=1
module REG1(Q, Qbar, D, L, C, nP, nR);
input D, C, L;
input nP, nR;
output Q,Qbar;

// TBD
wire mux2x1_d_res;
MUX1_2x1 mux2x1_d(mux2x1_d_res, Q, D, L);
D_FF inst_dff(.Q(Q), .Qbar(Qbar), .D(mux2x1_d_res), .C(C), .nP(nP), .nR(nR))

endmodule
```

iii. Register (32-bit)

Since all of the single bit register in the 32-bit register share the same properties, a generate statement was used to create

32 single bit registers instead of writing the instantiation statements one by one. For the ith register, the ith bit of Q, Q' and D are used as input for the single bit registers. The preset signal is set to 1 for every register.

```
// 32-bit register +ve edge, Reset on RESET=0
module REG32(Q, D, LOAD, CLK, RESET);
output [`DATA_INDEX_LIMIT:0] Q;

input CLK, LOAD;
input [`DATA_INDEX_LIMIT:0] D;
input RESET;

wire [`DATA_INDEX_LIMIT:0] Qbar;

// TBD
genvar i;
generate
        for(i=0; i<32; i=i+1)
        begin: reg32_gen_loop
                REG1 reg1_inst(Q[i], Qbar[i], D[i], LOAD, CLK, 1'b1, RESET);
        end
endgenerate
endmodule
```

iv. Decoder (2x4)

The DECODER_2x4 module follows the same steps of declaring the output and input ports followed by instantiating the logic gates to achieve the circuit as shown in the requirements section for a 2x4 line decoder in Figure 10.

```
// 2x4 Line decoder
module DECODER_2x4(D,I);
// output
output [3:0] D;
// input
input [1:0] I;

// TBD
not inst_inv_i0(inv_I0, I[0]);
not inst_inv_i1(inv_I1, I[1]);
and inst_and1(D[0], inv_I1, inv_I0);
and inst_and2(D[1], inv_I1, I[0]);
and inst_and3(D[2], I[1], inv_I0);
and inst_and4(D[3], I[0], I[1]);

endmodule
```

10) register_file.v

The register file of DaVinci v1.0m (32x32) is defined in this file. The ports for register file inputs and outputs are initialized with bit widths defined in the project definition file. A 5x32 decoder is instantiated to determine the address needed to write to if the write signal is on and to retrieve the load value to input into the 32 32-bit registers.

```
// This is going to be +ve edge clock triggered register file.
// Reset on RST=0
module REGISTER_FILE_32x32(DATA_R1, DATA_R2, ADDR_R1, ADDR_R2,
                           DATA_W, ADDR_W, READ, WRITE, CLK, RST);

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

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

wire [31:0] decoder_5x32_out;
wire [31:0] L;
DECODER_5x32 inst_decoder_5x32(.D(decoder_5x32_out), .I(ADDR_W));
```

32 32-bit registers are instantiated with the ith load bit into the ith register. The result or content of each register is stored into individual wires that will be inputted into multiplexers.

```
wire [31:0] reg0_res;
REG32 inst_reg32_0(reg0_res, DATA_W, L[0], CLK, RST);

wire [31:0] reg1_res;
REG32 inst_reg32_1(reg1_res, DATA_W, L[1], CLK, RST);

wire [31:0] reg2_res;
REG32 inst_reg32_2(reg2_res, DATA_W, L[2], CLK, RST);

wire [31:0] reg3_res;
REG32 inst_reg32_3(reg3_res, DATA_W, L[3], CLK, RST);

wire [31:0] reg4_res;
REG32 inst_reg32_4(reg4_res, DATA_W, L[4], CLK, RST);

wire [31:0] reg5_res;
REG32 inst_reg32_5(reg5_res, DATA_W, L[5], CLK, RST);

wire [31:0] reg6_res;
REG32 inst_reg32_6(reg6_res, DATA_W, L[6], CLK, RST);

wire [31:0] reg7_res;
REG32 inst_reg32_7(reg7_res, DATA_W, L[7], CLK, RST);

wire [31:0] reg8_res;
REG32 inst_reg32_8(reg8_res, DATA_W, L[8], CLK, RST);

wire [31:0] reg9_res;
REG32 inst_reg32_9(reg9_res, DATA_W, L[9], CLK, RST);

wire [31:0] reg10_res;
REG32 inst_reg32_10(reg10_res, DATA_W, L[10], CLK, RST);

wire [31:0] reg11_res;
REG32 inst_reg32_11(reg11_res, DATA_W, L[11], CLK, RST);
```

For register file read implementation, all of the register results are inputted into two multiplexers. The two multiplexers have differing signals ADDR_R1 and ADDR_R2 so that the read result from the multiplexer can be in parallel. If READ is 1, the resulting output from the read operation will be the data read. If READ IS 0, the output is a data width or 32-bit of z.

```
wire [31:0] mux_32x1_res1;
wire [31:0] mux_32x1_res2;

MUX32_32x1 inst_mux_32x1_1(mux_32x1_res1, reg0_res, reg1_res, reg2_res, reg3_res,
                           reg4_res, reg5_res, reg6_res, reg7_res, reg8_res,
                           reg9_res, reg10_res, reg11_res, reg12_res, reg13_res,
                           reg14_res, reg15_res, reg16_res, reg17_res, reg18_res,
                           reg19_res, reg20_res, reg21_res, reg22_res, reg23_res,
                           reg24_res, reg25_res, reg26_res, reg27_res, reg28_res,
                           reg29_res, reg30_res, reg31_res, ADDR_R1);

MUX32_32x1 inst_mux_32x1_2(mux_32x1_res2, reg0_res, reg1_res, reg2_res, reg3_res,
                           reg4_res, reg5_res, reg6_res, reg7_res, reg8_res,
                           reg9_res, reg10_res, reg11_res, reg12_res, reg13_res,
                           reg14_res, reg15_res, reg16_res, reg17_res, reg18_res,
                           reg19_res, reg20_res, reg21_res, reg22_res, reg23_res,
                           reg24_res, reg25_res, reg26_res, reg27_res, reg28_res,
                           reg29_res, reg30_res, reg31_res, ADDR_R2);

MUX32_2x1 inst_mux_2x1_1(DATA_R1, {`DATA_WIDTH{1'bz} }, mux_32x1_res1, READ);

MUX32_2x1 inst_mux_2x1_2(DATA_R2, {`DATA_WIDTH{1'bz} }, mux_32x1_res2, READ);
```

11) 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.

The ports for signals are declared 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} };
```

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
```

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 the data in.

```
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
```

12) data_path.v

In the "data_path.v" file, the module DATA_PATH is defined with the input and output signals in its definition. The data path takes in a 32-bit control signal from the control unit, clock and reset signals, and data in from the memory. It outputs data that flows into the memory, address for the memory, zero result, and instruction to send to the control unit. The bit width for these inputs and outputs are defined in the project definition file.

```
`include "prj_definition.v"
module DATA_PATH(DATA_OUT, ADDR, ZERO, INSTRUCTION, DATA_IN, CTRL, CLK, RST)

// output list
output [`ADDRESS_INDEX_LIMIT:0] ADDR;
output ZERO;
output [`DATA_INDEX_LIMIT:0] DATA_OUT, INSTRUCTION;

// input list
input [`CTRL_WIDTH_INDEX_LIMIT:0]  CTRL;
input CLK, RST;
input [`DATA_INDEX_LIMIT:0] DATA_IN;
```

After obtaining the instruction from the memory in DATA_IN, this data must be parsed and stored into several registers so that they can be used by selection later. For instance, the information for rs, rt for general R-Type instruction need to be used as input addresses for the register file.

```
always @(DATA_IN) begin
/* ================== Parse data (instruction) ====================== */
// parse the instruction
// R-type
{opcode, rs, rt, rd, shamt, funct} = DATA_IN;
// I-type
{opcode, rs, rt, immediate } = DATA_IN;
// J-type
{opcode, address} = DATA_IN;

instr_reg <= DATA_IN;
//$write("In data path, instr = %8h\n", DATA_IN);
end
```

The special registers for the data path are the program counter and stack pointer 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. To initialize that pattern, a special register REG_32_PP needs to be instantiated.

```
defparam sp_inst.PATTERN = `INIT_STACK_POINTER;
REG32_PP sp_inst(.Q(sp), .D(alu_out), .LOAD(CTRL[15]), .CLK(CLK), .RESET(RST));

defparam pc_inst.PATTERN = `INST_START_ADDR;
REG32_PP pc_inst(.Q(pc), .D(jump_or_res), .LOAD(CTRL[0]), .CLK(CLK), .RESET(RST));

/* ================ PC selection ================== */

RC_ADD_SUB_32 inst_pcadd_1(.Y(pc_adder_out1), .CO(CO), .A(pc), .B({31'b0, 1'b1}), .SnA

MUX32_2x1 inst_mux_jr(.Y(jump_reg_or_increment), .I0(r1_data), .I1(pc_adder_out1), .S(

// Add with sign extended immediate
RC_ADD_SUB_32 inst_pcadd_2(.Y(pc_adder_out2), .CO(CO), .A(pc_adder_out1), .B(sign_exte

MUX32_2x1 inst_mux_pcadd(.Y(pc_or_add), .I0(jump_reg_or_increment), .I1(pc_adder_out2)

MUX32_2x1 inst_mux_j(.Y(jump_or_res), .I0({6'b0, address}), .I1(pc_or_add), .S(CTRL[3]
```

The register file is instantiated with stable r1 and r2 data store. To achieve the stability, 32 bit registers were instantiated for r1 data and r2 data. As a result, the control from the control unit can issue when r1 data and r2 data changes based on if the bits CTRL[30] and CTRL[27] are 0 or 1. The address for r1 and write address is assigned to the result from a multiplexer that selects between a result or 0 depending on whether the instruction currently in the data path is a stack instruction or not.

```
/* ================= init REG FILE ================== */

MUX32_2x1 inst_mux_32bit(.Y(rs_or_zero), .I0({27'b0, rs}), .I1(32'b0), .S(CTRL[6]))

wire [31:0] r1_data_store;
REG32 r1_data_store_inst(.Q(r1_data_store), .D(r1_data), .LOAD(CTRL[30]), .CLK(CLK)

wire [31:0] r2_data_store;
REG32 r2_data_store_inst(.Q(r2_data_store), .D(r2_data), .LOAD(CTRL[27]), .CLK(CLK)

REGISTER_FILE_32x32 inst_reg_32x32(.DATA_R1(r1_data), .DATA_R2(r2_data),
                                   .ADDR_R1(rs_or_zero[4:0]), .ADDR_R2(rt),
                                   .DATA_W(is_pc_add1), .ADDR_W(is_stack_op[4:0]),
                                   .READ(CTRL[7]), .WRITE(CTRL[8]), .CLK(CLK), .RST
```

The data path consists of many multiplexers due to conflicting data paths and below is an example of instantiation of a multiplexer for selecting operands for the ALU. In general, two options for the multiplexer are given in I0 and I1. The result is stored into a 32-bit wire and the selection signal is a designated bit position from the CTRL input.

```
/* ================ OPERAND select for ALU ================== */

MUX32_2x1 inst_mux_op1(.Y(shamt_or_1), .I0({31'b0, 1'b1}), .I1({27'b0, shamt}), .S(CTRL

MUX32_2x1 inst_mux_op2(.Y(zero_or_sign_ext), .I0({16'b0, immediate}), .I1(sign_extended

MUX32_2x1 inst_mux_op3(.Y(shamt_or_imm), .I0(zero_or_sign_ext), .I1(shamt_or_1), .S(CTR

// Alu select
//MUX32_2x1 inst_mux_alu1(.Y(rf_or_sp), .I0(r1_data), .I1(sp), .S(CTRL[16]));
MUX32_2x1 inst_mux_alu1(.Y(rf_or_sp), .I0(r1_data_store), .I1(sp), .S(CTRL[16]));


always @(posedge CLK) begin
$write("RF or SP = %h : %h\n", CTRL[16], rf_or_sp);
end


MUX32_2x1 inst_mux_alu2(.Y(is_r_type), .I0(shamt_or_imm), .I1(r2_data_store), .S(CTRL[2


// Alu instantiation
ALU inst_ALU(.A(rf_or_sp), .B(is_r_type), .OPRN({1'bx, CTRL[25:21]}), .Y(alu_out), .ZER
```

13) 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.

In the control unit, all of the input ports are the output ports from the data path. The control unit output ports include the 32-bit control signal and read/write signals as input into the memory.

```
module CONTROL_UNIT(CTRL, READ, WRITE, ZERO, INSTRUCTION, CLK, RST);
// Output signals
output [`CTRL_WIDTH_INDEX_LIMIT:0]  CTRL;
output READ, WRITE;

// input signals
input ZERO, CLK, RST;
input [`DATA_INDEX_LIMIT:0] INSTRUCTION;

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;
reg [`CTRL_WIDTH_INDEX_LIMIT:0] C;

assign CTRL = C;
assign READ = C[4];
assign WRITE = C[5];
```

i. 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
```

ii. Instruction Fetch

In the instruction fetch phase, the instruction at the address of program counter is fetched and stored in the instruction register. To read the instruction at the PC address from the memory, the memory read and write signals are set to 1 and 0, respectively.

```
// FETCH: Get next instruction from memory with address as content in PC register
if (proc_state === `PROC_FETCH) begin
    $write("\n======================\n");
    $write("Instruction Fetch: %h\n", INST_REG);
    // mem_r = CTRL[4] mem_w = CTRL[5]
    C[4]=1'b1; C[5]=1'b0;
    // pc_load = CTRL[0]
    C[0]= 1'b0;
    // r1 load
    C[30] = 1'b1;
    // r2 load
    C[27] = 1'b1;
    // sp load
    C[15] = 1'b0;
    // ma_sel_2
    C[31] = 1'b1;

    // reg_r = CTRL[7] reg_w = CTRL[8]
    C[7] = 1'b1; C[8] = 1'b0;
end
```

iii. 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. To do so, the control signals C[7] (read) and C[8] (write) are set to read mode or 1 and 0, respectively.

```
// DECODE: Parse instruction and get values from register file
else if (proc_state === `PROC_DECODE) begin
    // mem_r = CTRL[4] mem_w = CTRL[5]
    C[4]=1'b1; C[5]=1'b0;
    // ma_sel_2
    C[31] = 1'b1;
    //pc_load = CTRL[0];
    C[0] = 1'b0;
    // CTRL[30]; // r1 load
    C[30] = 1'b1;
    // r2 load
    C[27] = 1'b1;
    // SP LOAD
    C[15] = 1'b0;

    $write("\n=====================\n");
    $write("Instruction Decode\n");

    // Set register file control to read
    // reg_r = CTRL[7] reg_w = CTRL[8]
    C[7] = 1'b1; C[8] = 1'b0;

    // Read R[rs] and R[rt] from register file (rt automatically selected)
    C[27] = 1'b1;
    // SP LOAD
    C[15] = 1'b0;
```

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 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;
```

iv. 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.

```
// R-Type (except jr)
if (opcode === 6'h00 && funct !== 6'h08) begin
    // Select alu operation, alu_oprn = CTRL[21];
    case(funct)
        6'h20: C[25:21] = `ALU_OPRN_WIDTH'h01; // add
        6'h22: C[25:21] = `ALU_OPRN_WIDTH'h02; // sub
        6'h2c: C[25:21] = `ALU_OPRN_WIDTH'h03; // mul
        6'h24: C[25:21] = `ALU_OPRN_WIDTH'h06; // and
        6'h25: C[25:21] = `ALU_OPRN_WIDTH'h07; // or
        6'h27: C[25:21] = `ALU_OPRN_WIDTH'h08; // nor
        6'h2a: C[25:21] = `ALU_OPRN_WIDTH'h09; // slt
        6'h01: C[25:21] = `ALU_OPRN_WIDTH'h05; // sll
        6'h02: C[25:21] = `ALU_OPRN_WIDTH'h04; // srl
    endcase

    // Select first alu operand
    C[16] = 1'b0; //R[rs] - always for R-type

    // Select second alu operand
    // If sll or srl instruction, use shamt for op2
    if (funct === 6'h01 || funct === 6'h02) begin
        C[17] = 1'b1;
        C[19] = 1'b1;
        C[20] = 1'b0;
    end
    // Else, use rt for op2
    else
        C[20] = 1'b1; // 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 (opcode !== 6'h02 && opcode !== 6'h03 && opcode !== 6'h1b
        && opcode !== 6'h1c && opcode !== 6'h0f) begin

    // Select first alu operand
    C[16] = 1'b0; // R[rs] for all I-type except lui

    // Select second alu operand: zero ext, sign ext immediate, or
    // For andi and ori, use ZeroExtImm
    if (opcode === 6'h0c || opcode === 6'h0d) begin
        C[18] = 1'b0;
        C[19] = 1'b0;
        C[20] = 1'b0;
    end
    // For beq, bne, use R[rt]
    else if (opcode === 6'h04 || opcode === 6'h05)
        C[20] = 1'b1;
    // For the rest, use SignExtImm
    else begin
        C[18] = 1'b1;
        C[19] = 1'b0;
        C[20] = 1'b0;
    end

    // Select alu operation
    case(opcode)
        6'h08: C[25:21] = `ALU_OPRN_WIDTH'h01; // add
        6'h2b: C[25:21] = `ALU_OPRN_WIDTH'h01; // sw
        6'h1d: C[25:21] = `ALU_OPRN_WIDTH'h03; // muli
        6'h0c: C[25:21] = `ALU_OPRN_WIDTH'h06; // andi
        6'h0d: C[25:21] = `ALU_OPRN_WIDTH'h07; // ori
        6'h0a: C[25:21] = `ALU_OPRN_WIDTH'h09; // slti
        6'h04: C[25:21] = `ALU_OPRN_WIDTH'h02; // beq
        6'h05: C[25:21] = `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 (opcode === 6'h1b) begin
    // set RF ADDR_R1 to be 0, r1_sel_1 = CTRL[6];
    C[6] = 1'b1;
end
// Pop instruction
else if (opcode === 6'h1c) begin
    // Increment stack pointer
    // Alu op1 = 1, op2 = sp reg
    C[17] = 1'b0;
    C[19] = 1'b1;
    C[20] = 1'b0;
    C[16] = 1'b1;
    C[25:21] = `ALU_OPRN_WIDTH'h01; //add to sp
    C[15] = 1'b1; // SP LOAD
end
```

v. Memory Access

The memory write back phase is only applicable for lw, sw, push, and pop instructions. 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] by setting the control bits according to the data path.

```
// Lw instruction
if (opcode === 6'h23) begin
    // dmem_r = CTRL[27] dmem_w = CTRL[28]
    C[4]=1'b1; C[5]=1'b0;
    //mem_addr = ALU_RESULT;
    C[26] = 1'b0; C[31]=1'b0;
end
// Sw instruction
else if (opcode === 6'h2b) begin
    C[4]=1'b0; C[5]=1'b1;
    //C[27]=1'b0; C[28]=1'b1;
    //mem_data = r2data, mem_addr = ALU_RESULT;
    C[26] = 1'b0;
    C[29] = 1'b0; C[31]=1'b0;
    //mem_datax = RF_DATA_R2;
end
// Push instruction
else if (opcode === 6'h1b) begin
    C[4]=1'b0; C[5]=1'b1;
    //mem_addr = SP_REG;
    C[26] = 1'b1; C[31]=1'b0;
    //mem_datax = RF_DATA_R1; // M[SP_REG] = R[0]
    C[29] = 1'b1;

end
// Pop instruction
else if (opcode === 6'h1c) begin
    C[4]=1'b1; C[5]=1'b0; C[26] = 1'b1; C[31]=1'b0;
end
```

vi. 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].

```
// Write back to RF or PC_REG(beq, bne, jmp, jal)
// Increase PC_REG BY 1
C[0] = 1'b1;
C[1] = 1'b1;
C[2] = 1'b0;
C[3] = 1'b1;

// Set RF writing address and data/control to write back into RF
// R-Type
if (opcode === 6'h00 && funct !== 6'h08) begin
    C[7]=1'b0; C[8]=1'b1;
    //rf_addr_w = stored_rd; rf_data_w = ALU_RESULT;
    C[9]=1'b0; C[11]=1'b1;
    C[12]=1'b0; C[13]=1'b0; C[14]=1'b1;
end
// Jump register
else if (opcode === 6'h00 && funct === 6'h08) begin
    C[7]=1'b1; C[8]=1'b0; // rf_addr_r1 already is rs, turn on to get
    //PC_REG = RF_DATA_R1; // PC = R[rs]
    C[1]=1'b0;
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 is updated by adding the stored sign immediate value to itself.

```
// Write to R[rt]
C[7]=1'b0; C[8]=1'b1;

// If load word instruction, get memory data
if (opcode === 6'h23) begin
    //rf_addr_w = stored_rt;
    C[9]=1'b1; C[11]=1'b1;
    //rf_data_w = MEM_DATA; // R[rt] = memory data
    C[12]=1'b1; C[13]=1'b0; C[14]=1'b1;
end
// If lui instruction, extend imm
else if (opcode === 6'h0f) begin
    //rf_addr_w = rf_addr_r2;
    //rf_addr_w = stored_rt;
    C[9]=1'b1; C[11]=1'b1;
    C[13]=1'b1; C[14]=1'b1;
end
// If beq instruction, update PC if zero flag is on
else if (opcode === 6'h04) begin
    // If zero flag is on, R[rs] == R[rt]
    if (ZERO) begin
        //PC_REG = PC_REG + stored_signextimm;
        C[2]=1'b1;
    end
end
// If bne instruction (opp of beq)
else if (opcode === 6'h05) begin
    // If zero flag is on, R[rs] == R[rt]
    if (~ZERO) begin
        C[2]=1'b1;
    end
end
else begin
    //rf_addr_w = stored_rt;
    C[9]=1'b1; C[11]=1'b1;
    //rf_data_w = ALU_RESULT; // R[rt] = result from ALU
    C[12]=1'b0; C[13]=1'b0; C[14]=1'b1;
```

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 (opcode === 6'h02) begin
    //PC_REG = stored_jump_addr;
    C[3]=1'b0;
end

// Jal instruction
else if (opcode === 6'h03) begin
    // Write to R[31]
    C[7]=1'b0; C[8]=1'b1;
    //rf_addr_w = 31;
    C[10]=1'b1; C[11]=1'b0;
    //rf_data_w = PC_REG;
    C[14]=1'b0;
    //PC_REG = stored_jump_addr;
    C[3]=1'b0;
end
// Pop instruction
else if (opcode === 6'h1c) begin
    C[7]=1'b0; C[8]=1'b1;
    //rf_addr_w = 0;
    C[10]=1'b0; C[11]=1'b0;
    //rf_data_w = MEM_DATA;
    C[12]=1'b1; C[13]=1'b0; C[14]=1'b1;
end
else if (opcode === 6'h1b) begin
    C[17] = 1'b0;
    C[19] = 1'b1;
    C[20] = 1'b0;
    C[16] = 1'b1;
    C[25:21] = `ALU_OPRN_WIDTH'h02; //sub op
    C[15] = 1'b1; // SP LOAD
```

14) 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 declared. 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, alu
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(AL
                      .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(al

endmodule;
```

15) 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) 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.

```
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
```

2) Simulation and Test of DaVinci v1.0m

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 v1.0m. It is also possible to add other configurations to further test DaVinci v1.0m with other programs using the same CS147DV instruction set.

```
// 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;
```

3) Testing for Separate Components

Individual test benches are provided for every single component. Follow the same instructions to create a simulation and select only the module name of the component with its corresponding test bench module. Testing for most components is achieved by setting the input and observing the output waveform in ModelSim. An example for the waveform for testing the ALU is shown in Figure 38 below.

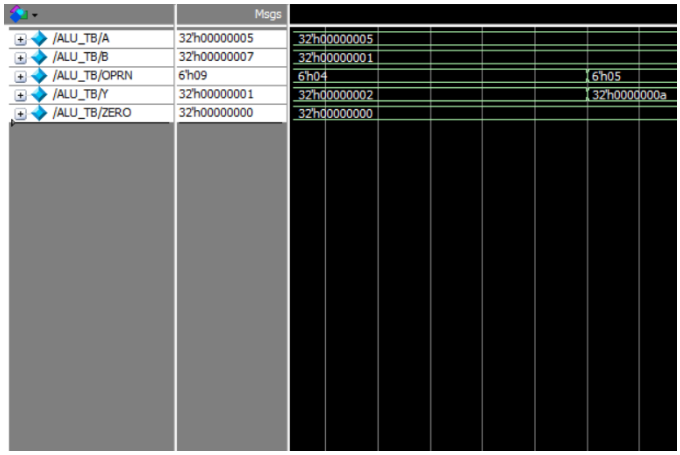

Fig.38. Waveform from testing the ALU

V. Conclusion

This project heavily focused on the idea of synthesizing smaller digital components together to create the individual parts of computer architecture such as the ALU and register file. The integration of the ALU, register file, and memory was successful through implementation of the data path and control unit. 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. Unlike the previous project, DaVinci v1.0m also required instantiating digital components in the data path to add meaning to the control signal issued from the control unit.

After completion or Project II or DaVinci v1.0, the understanding of the logic level of a computer system was clear. However, many problems arose due to small mistakes like incorrect wiring from using the wrong name. Those mistakes were easier to debug if a generate statement was not used. The most difficult part of this project was implementing the register file at the gate level since it is a newer concept in the class compared to the ALU. 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 a computer system.

References

[1] K. Patra. CS 147. Class Lab, Lab 11. San Jose State University, San Jose, CA, May 15, 2017.

[2] K. Patra. CS 147. Class Lecture, Lecture 08. San Jose State University, San Jose, CA, May 15, 2017.

[3] K. Patra. CS 147. Class Lab, Lab 12. San Jose State University, San Jose, CA, May 15, 2017.

[4] K. Patra. CS 147. Class Lab, Lab 13. San Jose State University, San Jose, CA, May 15, 2017.

[5] K. Patra. CS 147. Class Lecture, Lecture 01. San Jose State University, San Jose, CA, March 31, 2017.

[6] K. Patra. CS 147. Class Lab, Lab 14. San Jose State University, San Jose, CA, May 15, 2017.

[7] K. Patra. CS 147. Class Lab, Lab 15. San Jose State University, San Jose, CA, May 15, 2017.

[8] K. Patra. CS 147. Class Lecture, Lecture 07. San Jose State University, San Jose, CA, May 15, 2017.

[9] K. Patra. CS 147. Class Lecture, Lecture 02. San Jose State University, San Jose, CA, April 4, 2017.

[10] K. Patra. CS 147. Class Lab, Lab 16. San Jose State University, San Jose, CA, May 15, 2017.

[11] K. Patra. CS 147. Class Lab, Lab 17. San Jose State University, San Jose, CA, May 15, 2017.

[12] K. Patra. CS 147. Class Lecture, Lecture 11. San Jose State University, San Jose, CA, May 15, 2017.