[[1]](#footnote-1)

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

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.

# 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.

# 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.

## 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.

### 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.

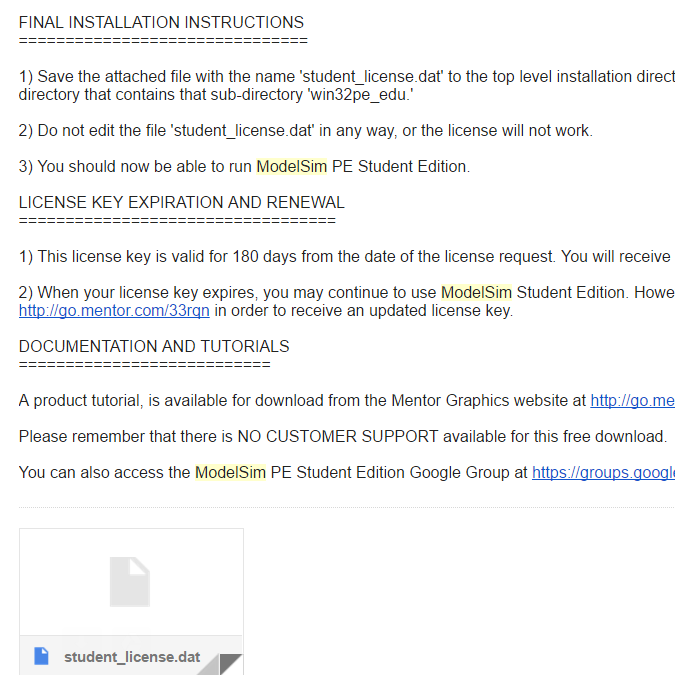


Fig. 2. Downloading ModelSim – License

### 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”.

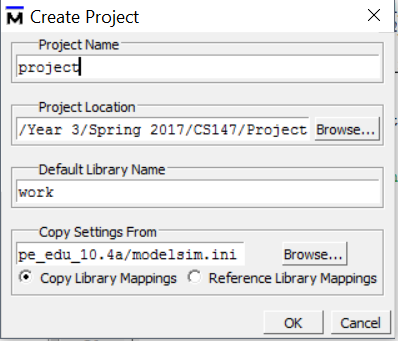


Fig. 3. Creating a project in ModelSim

### 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.

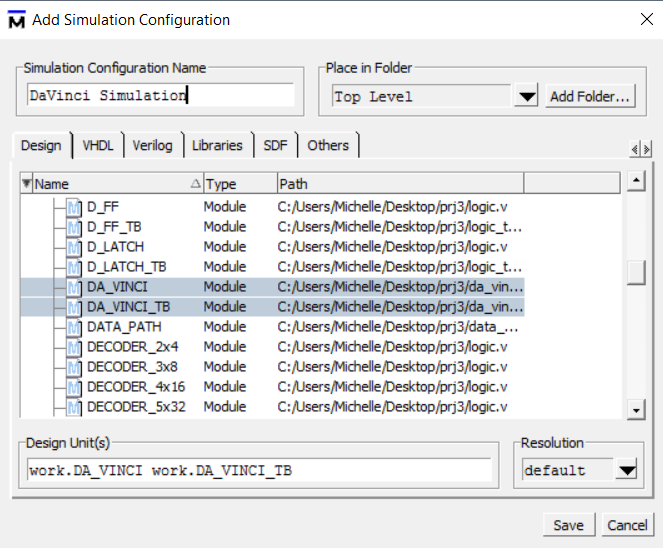


Fig. 4. Creating a simulation – Configuration window

### 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.

### 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.

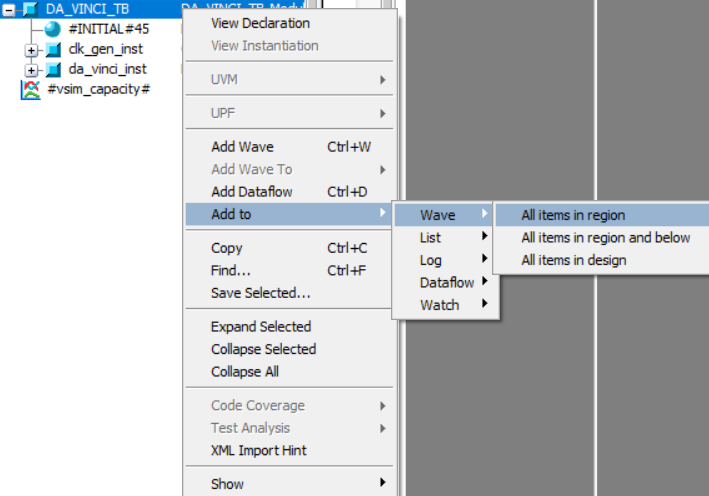


Fig. 5. Observing waveforms – Adding a wave

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.

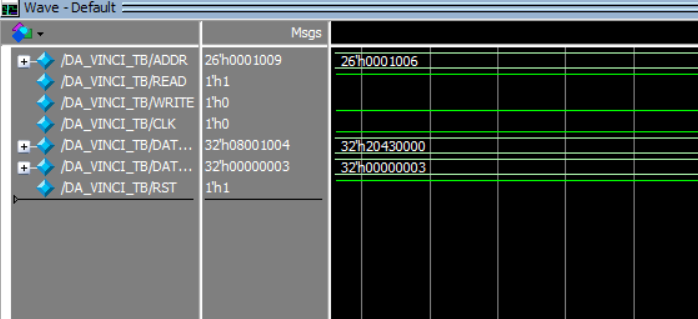


Fig. 6. Observing waveforms – Viewing the wave

## 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.

### 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.

1. *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.

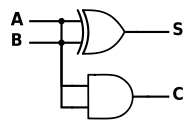


Fig. 7. Half adder circuit

1. *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.

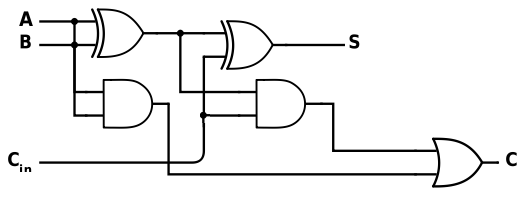


Fig. 8. Full adder circuit

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.

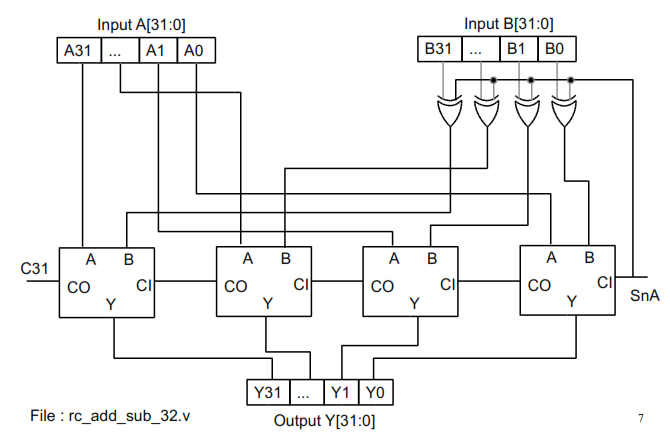


Fig. 9. Binary carry ripple adder-subtractor

### 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 |

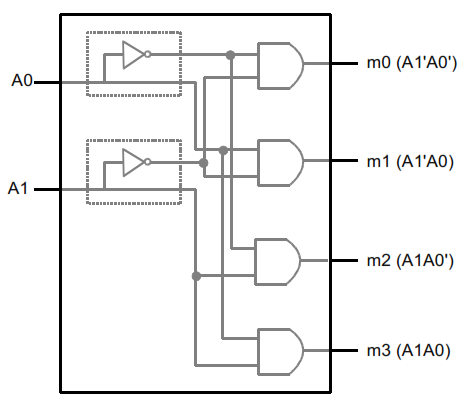


Fig. 10. 2x4 decoder circuit

### 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 log24 = 2.

TABLE III

Truth table for 4x1 Multiplexer

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

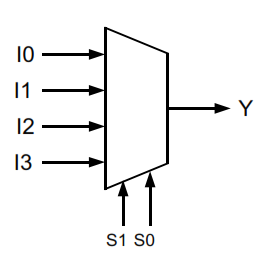


Fig. 11. General 4x1 multiplexer

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

### 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.

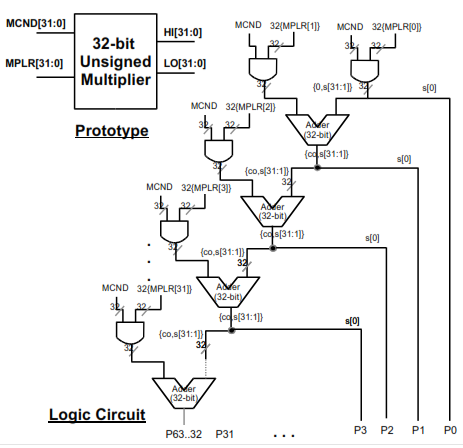


Fig. 13. Unsigned multiplier circuit

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.

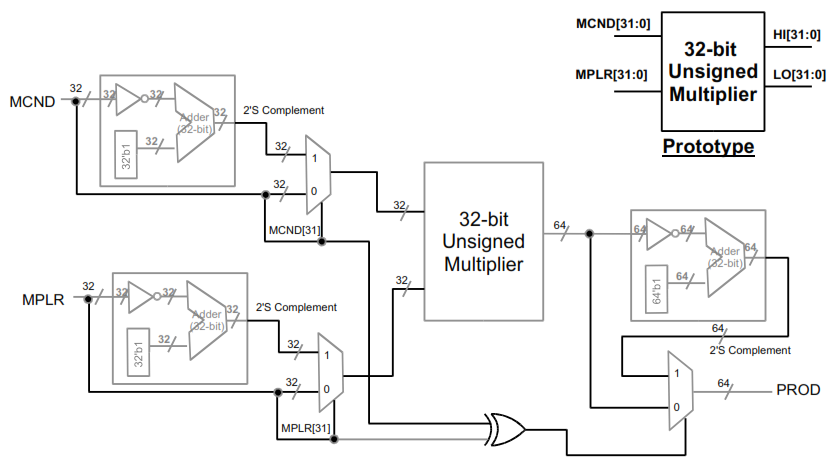
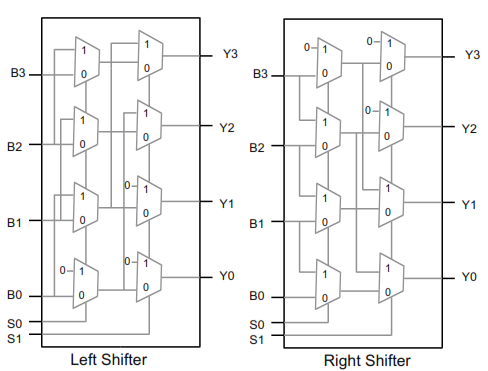


Fig. 14. 32-bit signed multiplier circuit

### 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

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.

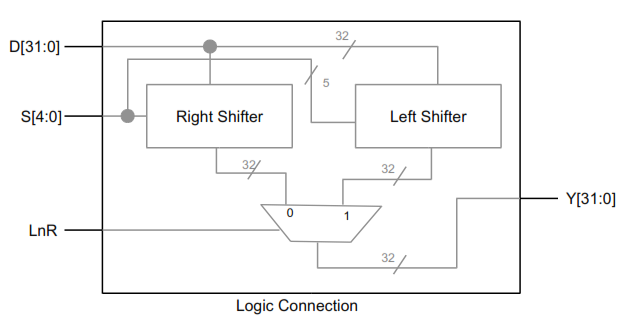


Fig. 16. Barrel shifter with left and right control

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.

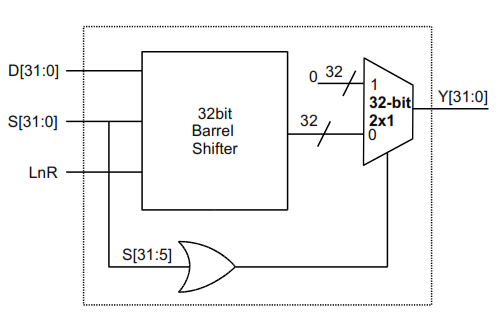


Fig. 17. Barrel shifter with 32-bit shift amount

### 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.

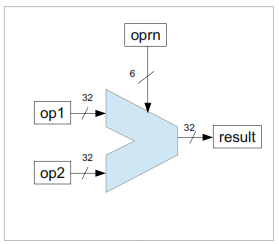


Fig.18. Schematic diagram representation of ALU

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.

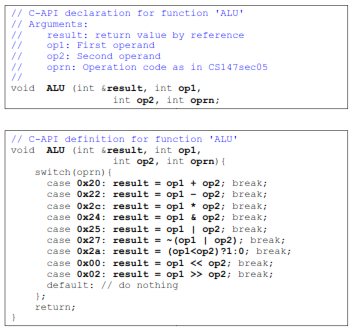


Fig.19. Corresponding C code for ALU

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.

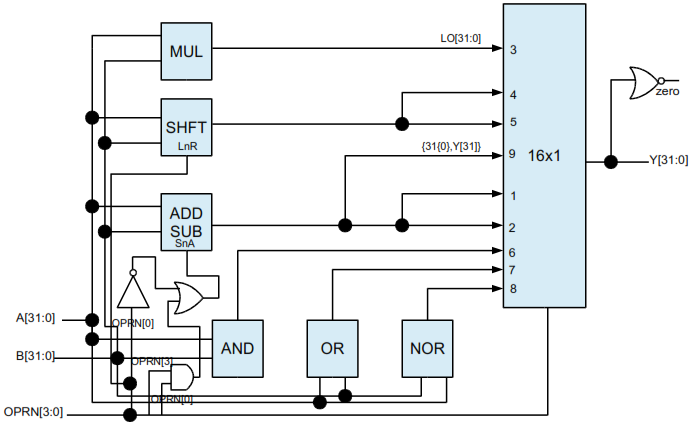


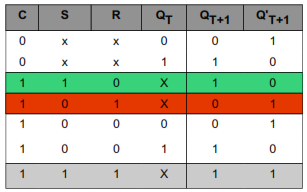
Fig.20. Gate level representation of the ALU

### 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, QT. The next output QT+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 QT+1 = Q’T+1 which is undefined since it contradicts circuit operations. Fortunately, the D latch resolves this issue.



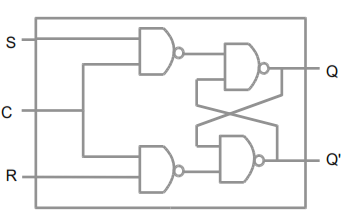
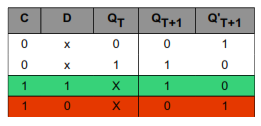


Fig.21. Schematic diagram of SR Latch

### 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.



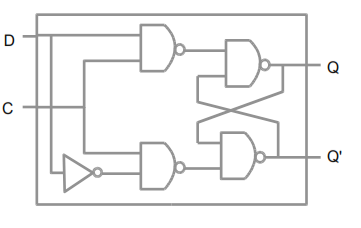


Fig.22. Schematic diagram of D Latch

### 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.

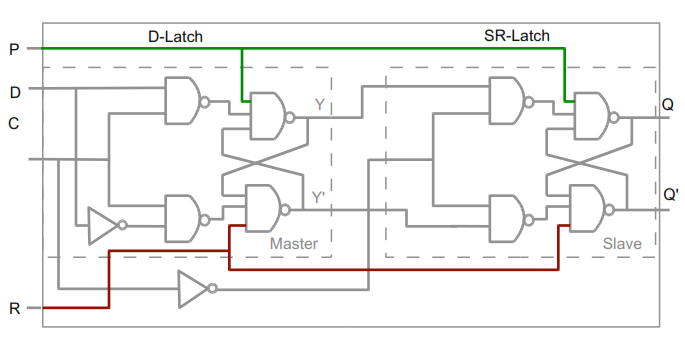


Fig.23. Schematic diagram of D Flip Flop

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.

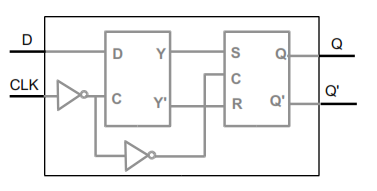


Fig.24. Positive edge triggered D Flip Flop

### 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.

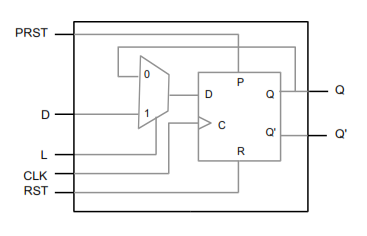


Fig.25. Diagram of a single bit register

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.

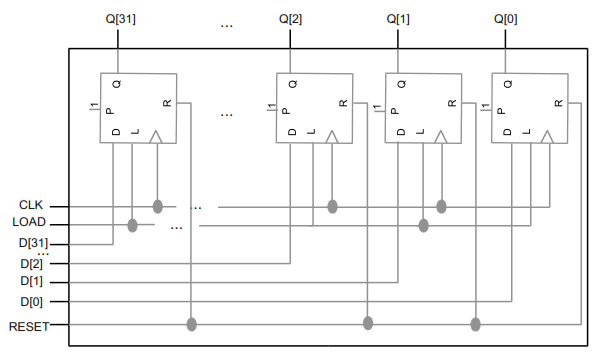


Fig.26. Diagram of a 32-bit register

### 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.

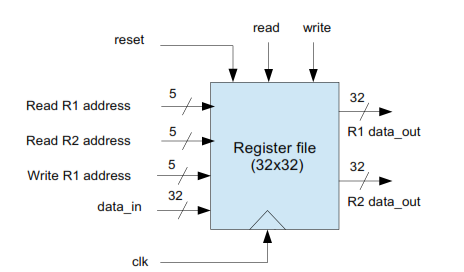


Fig.27. Schematic diagram of the register file

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.

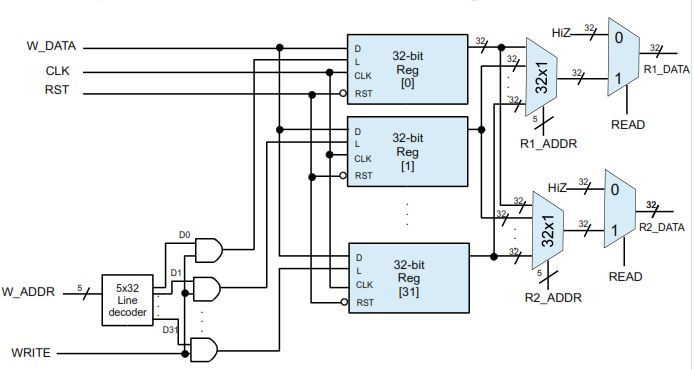


Fig.28. Gate level representation of register file

### 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.

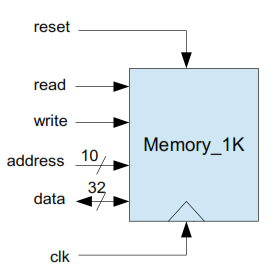


Fig.29. Schematic diagram of the memory

### 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.

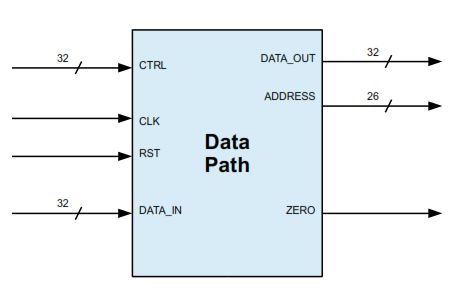


Fig.30. Schematic diagram of the data path

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.

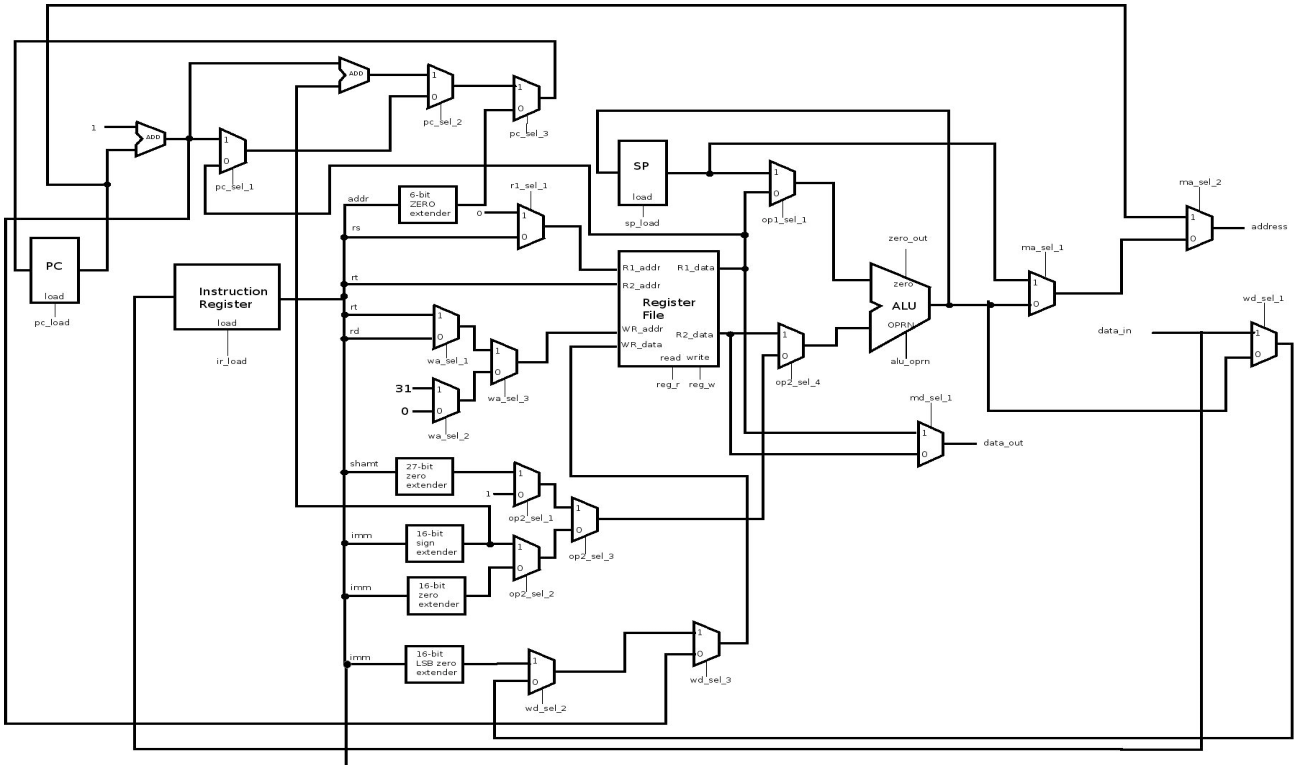


Fig.31. Gate level representation of the data path

1. *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.

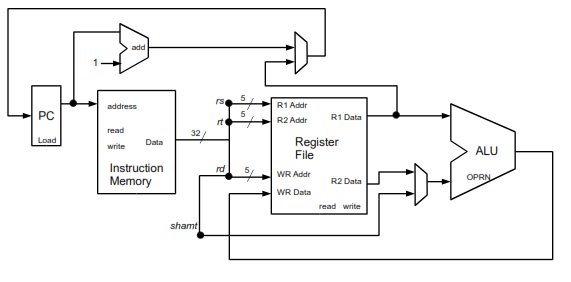


Fig.32. Data path for R-type instructions

### 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.

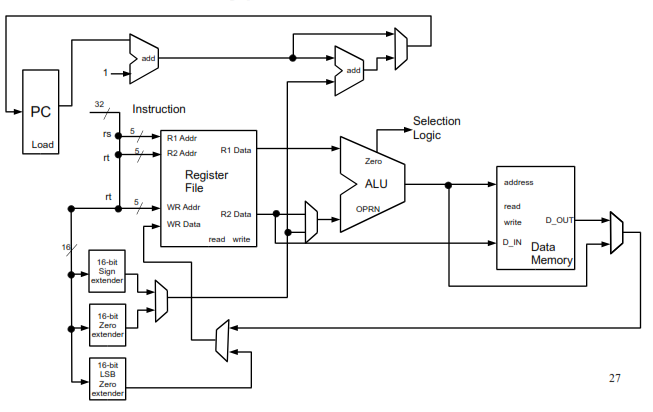


Fig.33. Data path for I-type instructions

1. *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.

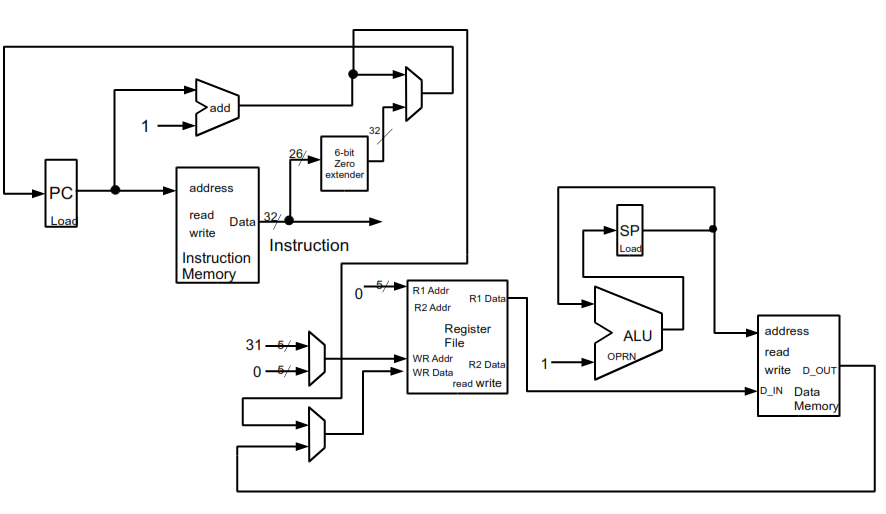


Fig.34. Data path for J-Type instructions

### 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.

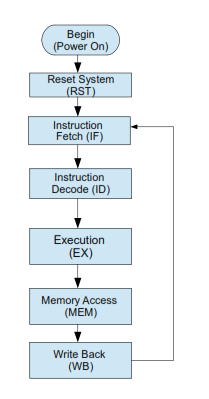


Fig.35. Control System Stages

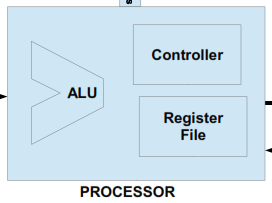


Fig.36. Processor

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.

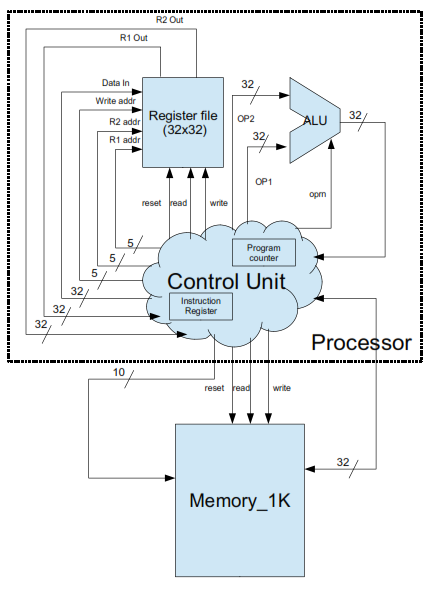


Fig.37. Control unit

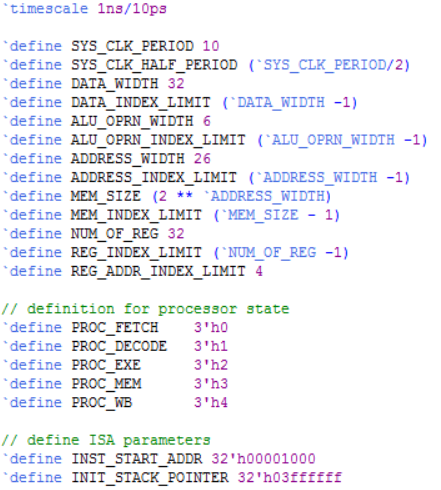
### Clock

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%.

# implementation

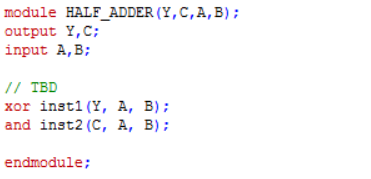
### prj\_definition.v

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.



### half\_adder.v

The “half\_adder.v” file initializes 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 for the single bit addition result and the AND gate for the carry bit result.



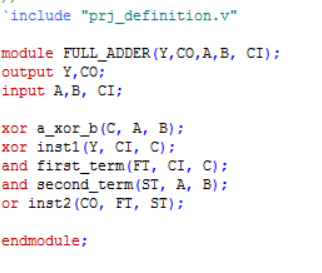
### full\_adder.v

The full adder implementation in “full\_adder.v” initializes 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 ⊕ B ⊕ C and the expression for the carry out is

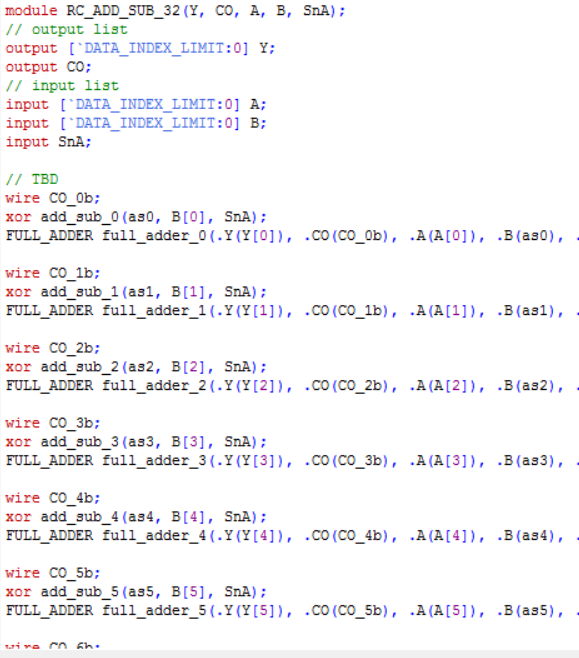
CO = CI(A ⊕ B) + AB.



### rc\_add\_sub\_32.v

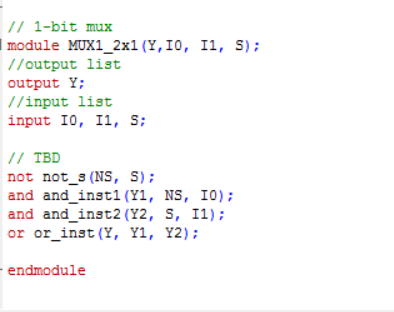
In the RC\_ADD\_SUB\_32 module, the output list and input list ports are declared in the beginning of the file. Since the adder-subtractor is 32-bit, the output Y and inputs A and B are 32-bit or the data index wide. The output CO is a single bit and the SnA signal is a single bit signal.

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.

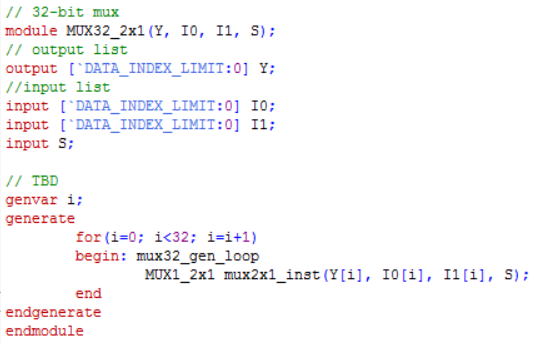


### mux.v

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.

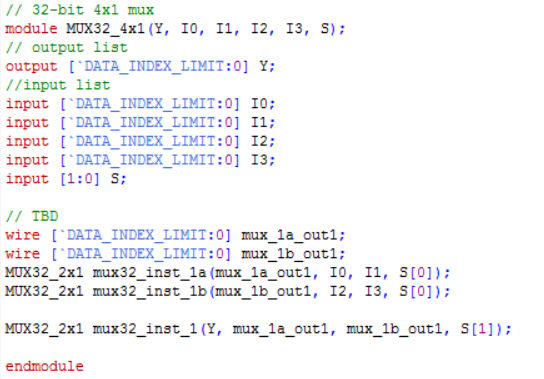


To obtain a 32-bit 2x1 multiplexer, the same 1-bit 2x1 multiplexer must be generated for every single bit. Instead of instantiating 32 individual multiplexers, it is more efficient to use a generate statement since the wiring for every multiplexer is similar, using the ith bit of the input and output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



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.

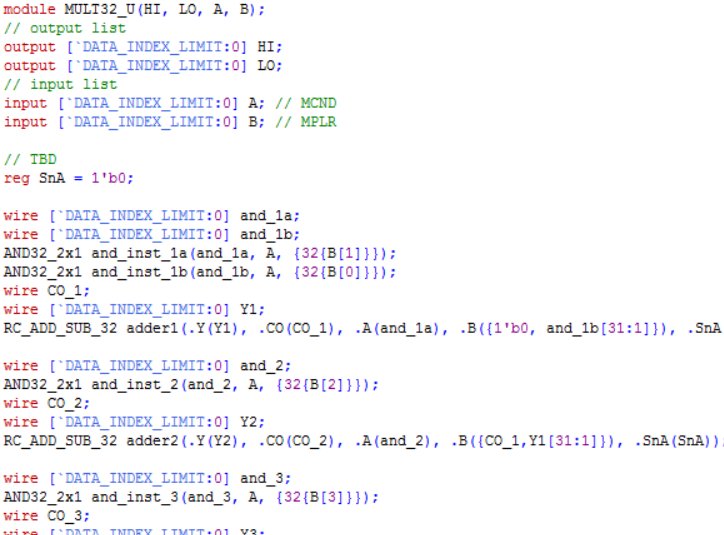
The 32-bit 8x1 multiplexer was implemented using the same mechanism of combining multiplexers to obtain a final selection output in Y. In the same way, the 8x1 multiplexer was used to build the 16x1, and the 16x1 multiplexer was used to build the 32x1 multiplexer.



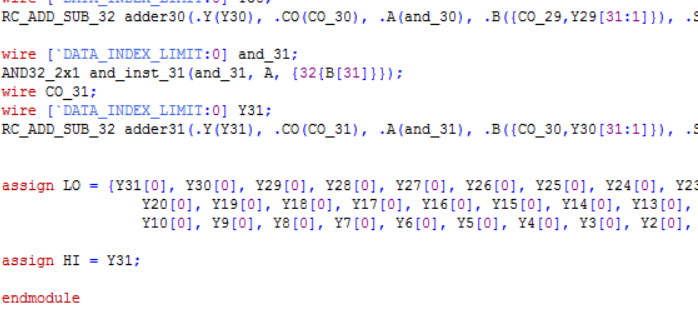
### mult.v

In the “mult.v” file, the input and output ports are initialized 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 and set to 0 to set the adder subtractor to add mode.

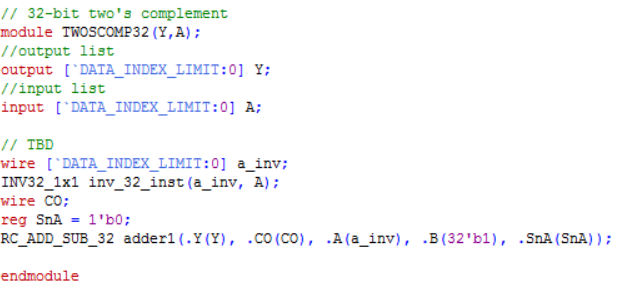
Following the circuit as discussed, every bit of the multiplier is AND’ed with the multiplicand and stored into a named wire to figure out if there is a value to add. The wire from the AND result is then used as input for the next adder. The other input for the adder is comprised of the carry out from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



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.

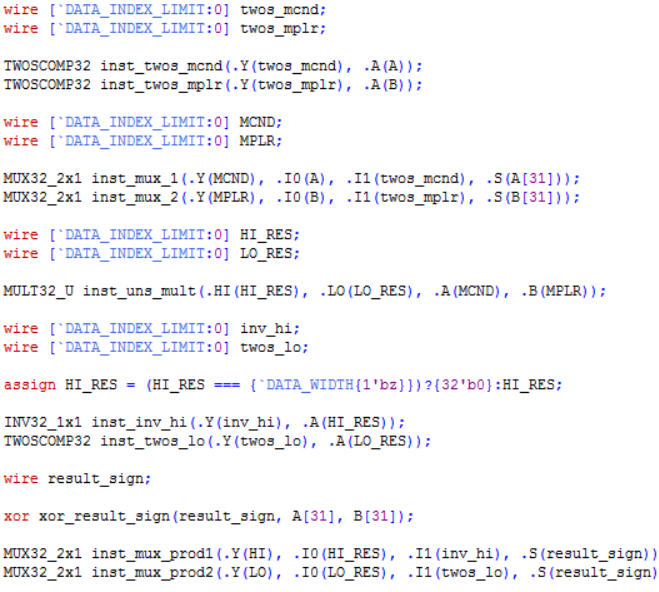


The two’s complement module needs to be implemented for the purpose of signed multiplication. After initializing the output port for Y and the input port for A, the input A is inverted by instantiating an inverter. Finally, the inverted result ~A must be added by 1. To do that, an adder-subtractor is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

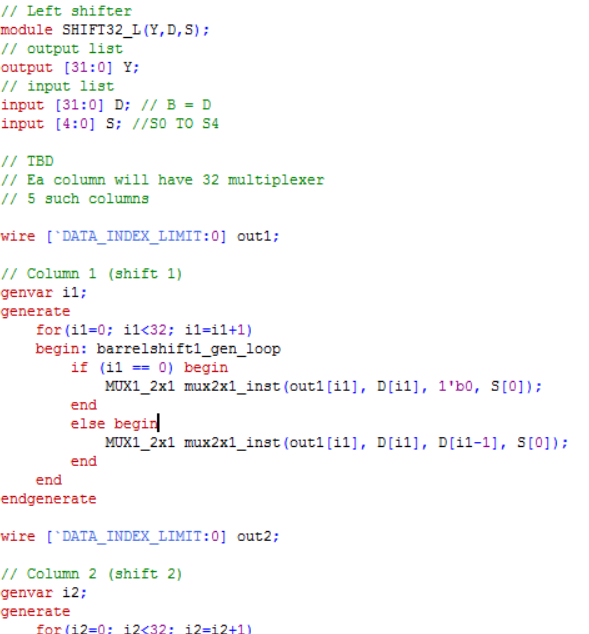


To implement the signed multiplier, two twos complement instances are created to compute the twos complement for both A and B. The two’s complement version is selected for A and B only if the value is negative. The selection is done by a multiplexer and the selection signal is determined by the most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

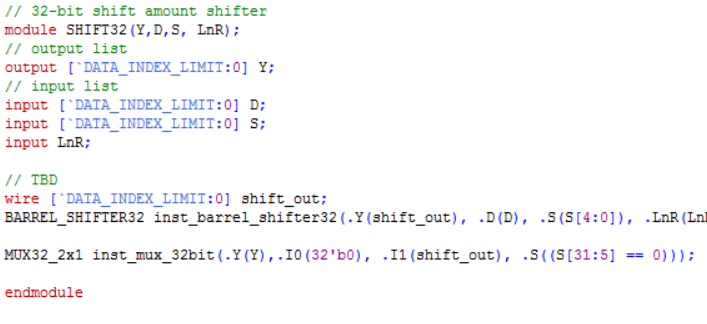
To ensure that the unsigned multiplier product’s sign is correct, the sign of the product is determined using XOR on the most significant bits of the A and B. The two’s complement of the result HI and LO registers is computed using the exact same method of selection by inputting the sign bit into a multiplexer.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



### barrel\_shifter.v





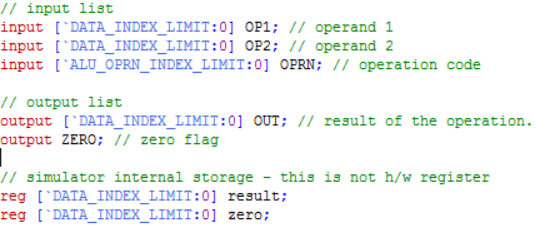


### alu.v

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

1. *Initializing the ports*

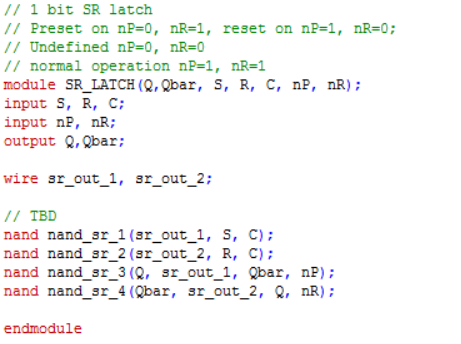
In the lines of code proceeding the declaration, whether the port is input or output is specified along with the port width. In this case, the ports op1, op2, and oprn are wire. Only the result and zero port needs to be specified as reg.

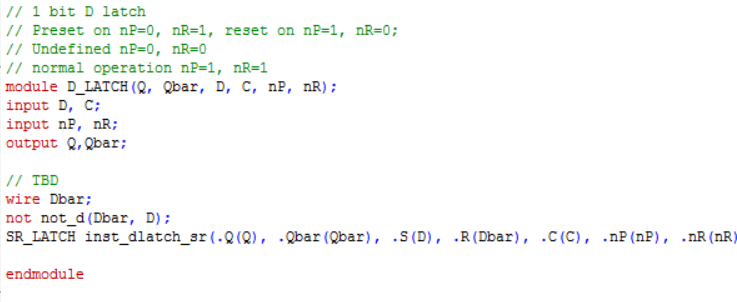


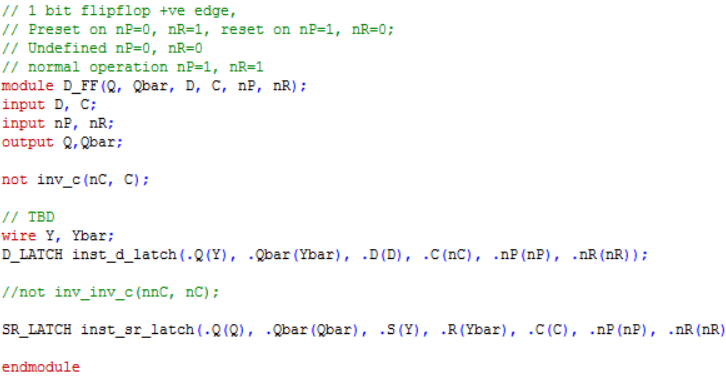
1. Instantiation of ALU components

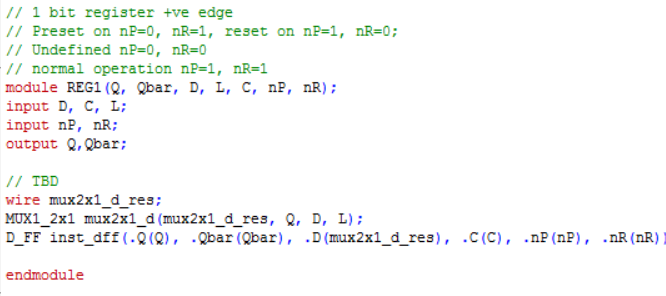
In the ALU, the following components are instantiated: adder/subtractor, multiplier, barrel shifter with left and right shift control, and, or, and not. All components must support 32-bit width of data and a 32 bit wire is initialized for every operation result and assigned to the results of the components.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

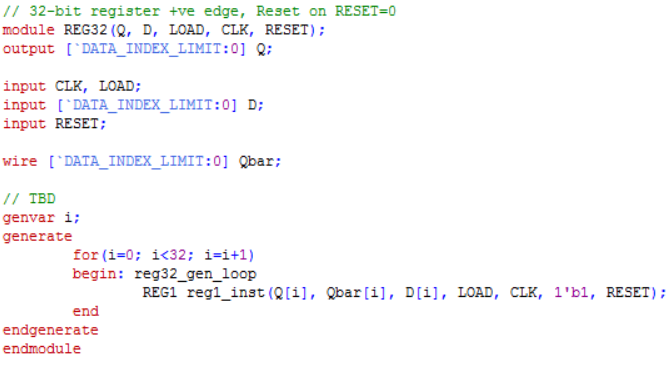
### logic.v

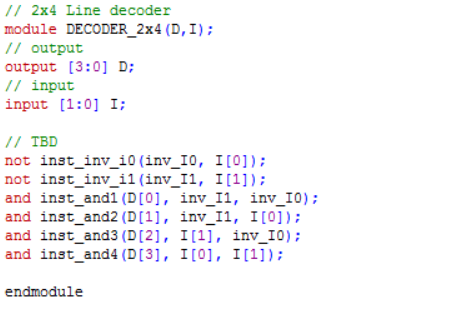


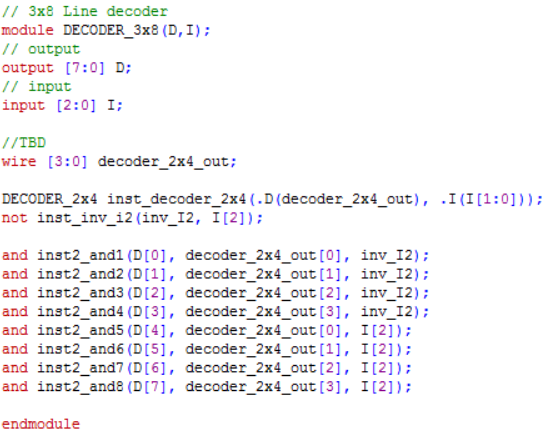












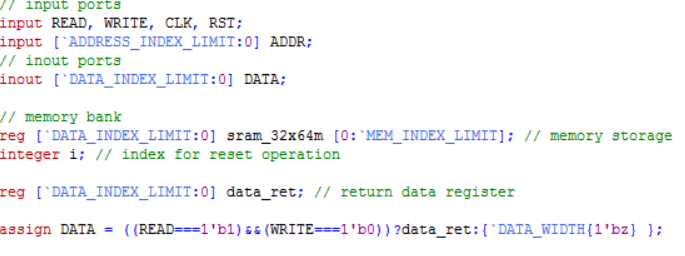
### register\_file.v

### 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.

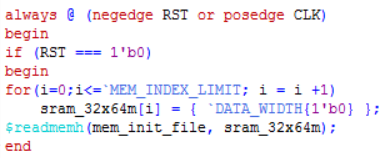
1. *Initialization*

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



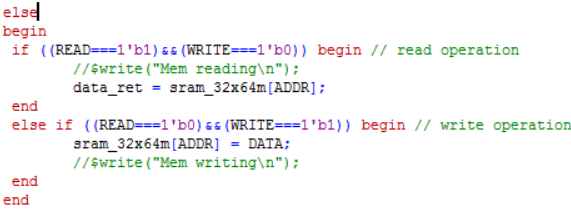
1. *Resetting the Memory Content*

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



1. *Data reading and writing*

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



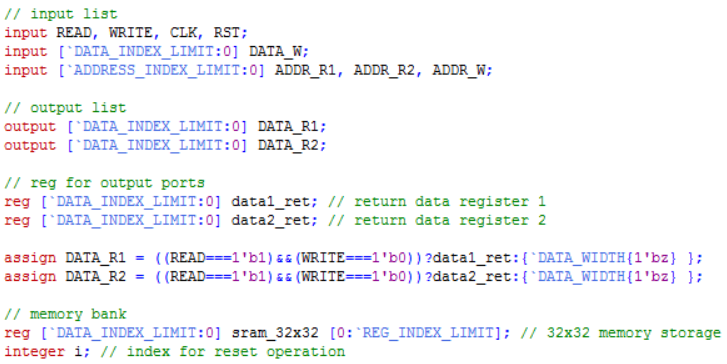
### data\_path.v

### register\_file.v

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

1. *Initialization*

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



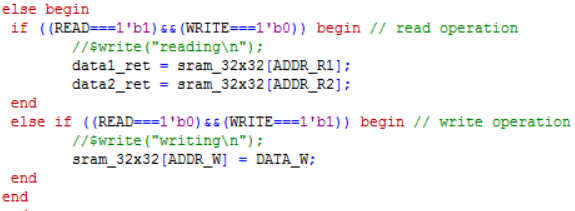
### Resetting the Register File Content

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



### Reading and Writing to Register File

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

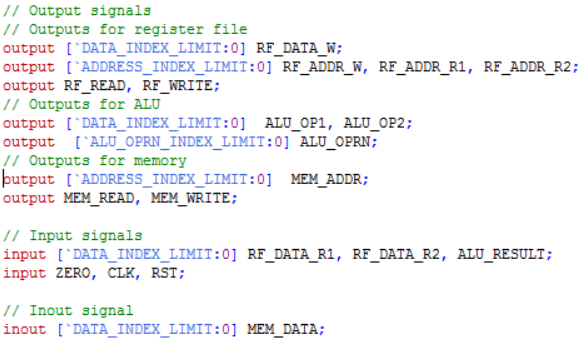


### control\_unit.v

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.

1. *Initializing the ports*

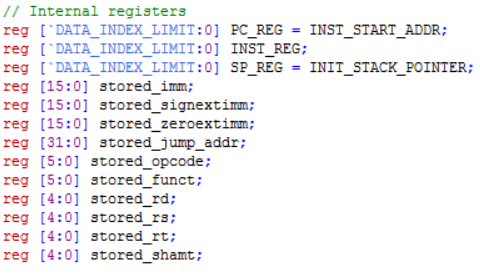
In the control unit, all of the ports from ALU, memory, and register file are combined. Here, they are defined similar to how they were defined in the ALU, memory, and register file. However, in the control unit, the inputs are outputs and vice versa.



1. *Registers*

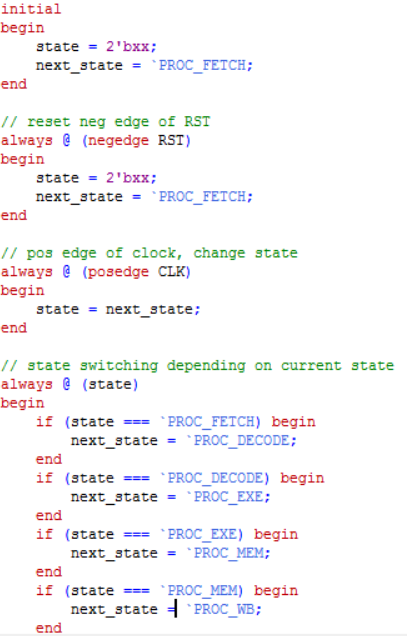
The special registers for the control unit are the program counter and instruction register. The ISA specification is defined in the project definition file the definition INST\_START\_ADDR is used to initialize the program counter register value to 32'h00001000.

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



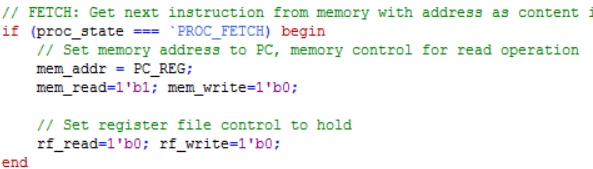
1. *State machine*

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.



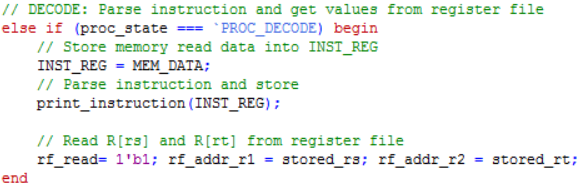
1. *Instruction Fetch*

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

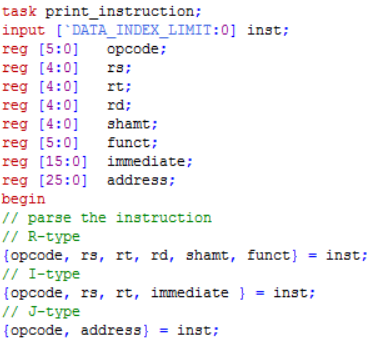


1. *Instruction Decode*

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.

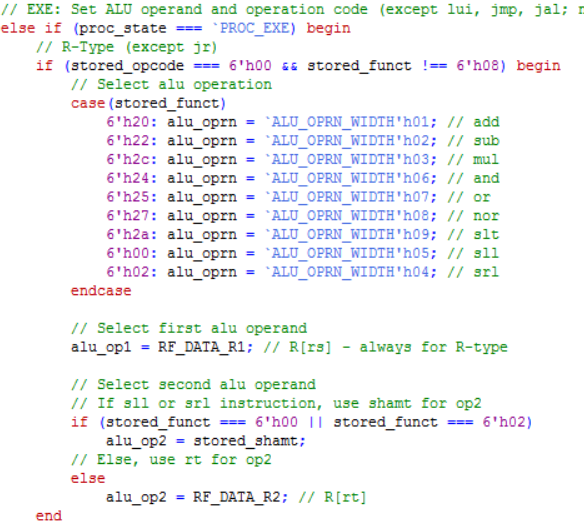


In the print instruction task, the instruction is parsed into a 6-bit opcode, 5-bit rs, rt, rd, shamt, and 6 bit function code for a R-type instruction. For I-Type, the instruction is parsed into a 16-bit immediate instead of rd, shamt, and funct. For J-Type, only the opcode and a 26-bit address is obtained.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

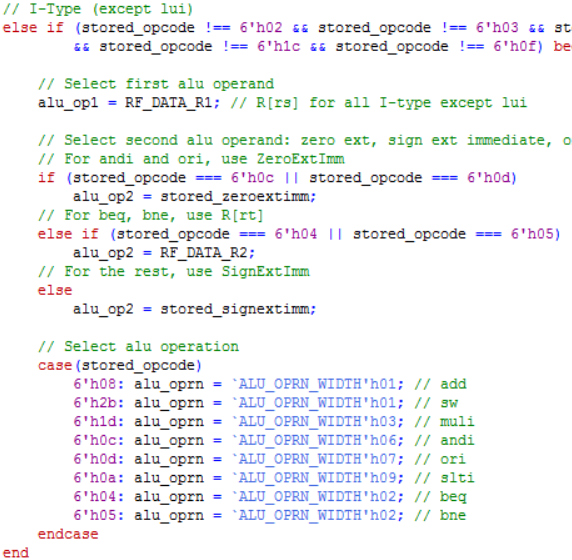


1. *Execution*

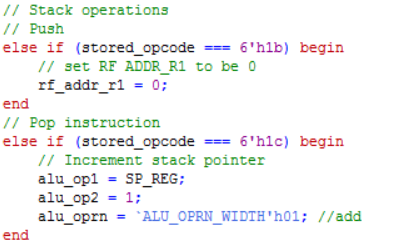
In the execution phase, the majority of the instructions need to perform computations using the ALU. For R-type instructions excluding jump register, the operands are rs and rt/shamt. Depending on the function code, the correct ALU operation code is selected. The first ALU operand is always rs and the 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.



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.

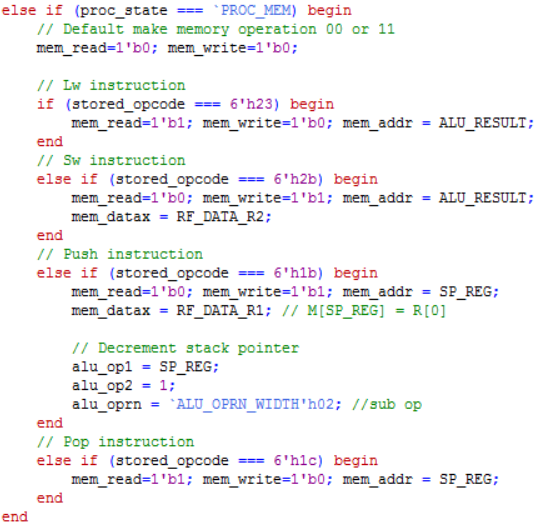


Only the push and pop instructions of the J-Type are configured at this stage. For push, set the register file read address to be 0 to prepare to write the result R[0] into the memory since the data result from the register file takes one clock cycle to obtain. For pop, increment the stack pointer by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



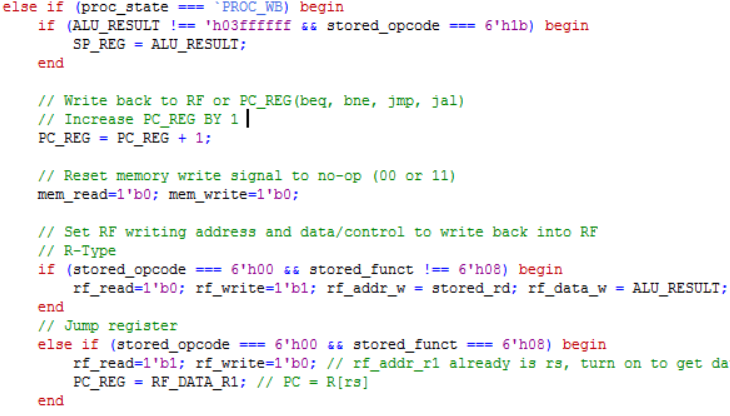
1. *Memory Access*

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



1. *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].



For I-Type instructions, ALU\_RESULT is written back into

the destination of R[rt] by selecting the address of rt for the

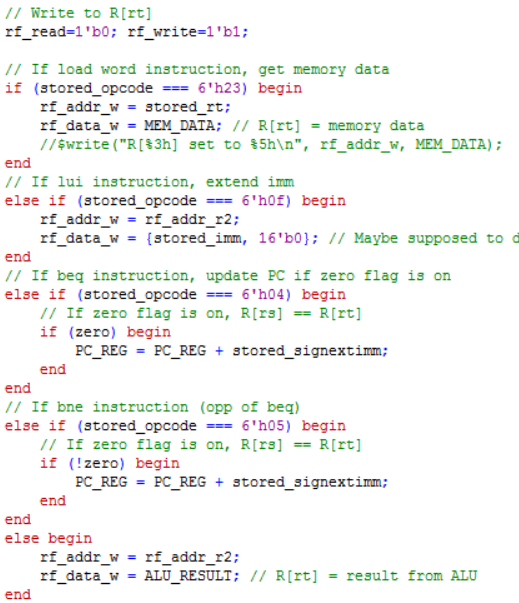
register file. However, there is an exception for lui and branch

instructions. For lui, the lower half of rt is set 16 bits of 0

while keeping the upper half. For branch instructions, the zero

flag is checked, and if the condition is satisfied, the PC

updates by adding the stored sign immediate value to itself.



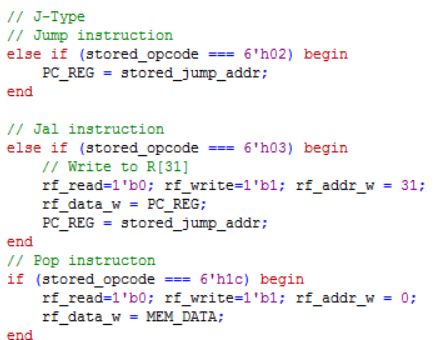
The jump instruction sets the program counter to the jump

address. The jump and link instruction stores the current value

in the PC register while updating the PC. The pop

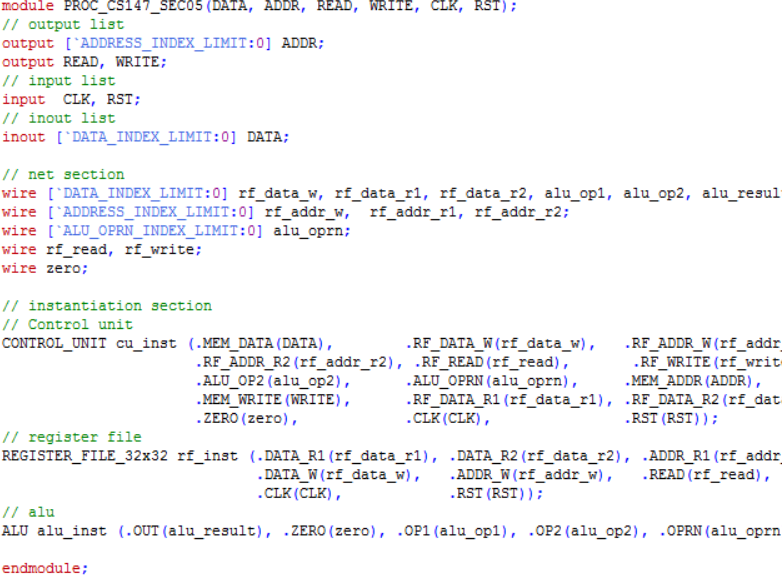
instruction writes to R[0] and the data written is the resulting

memory data value from the memory access stage.



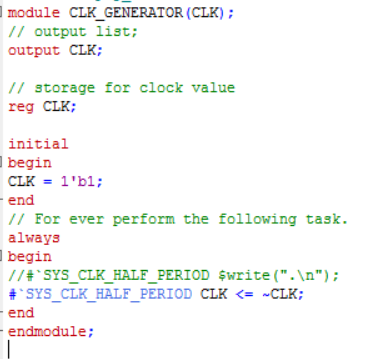
### processor.v

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



### 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.



# 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.

### Testing procedure

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

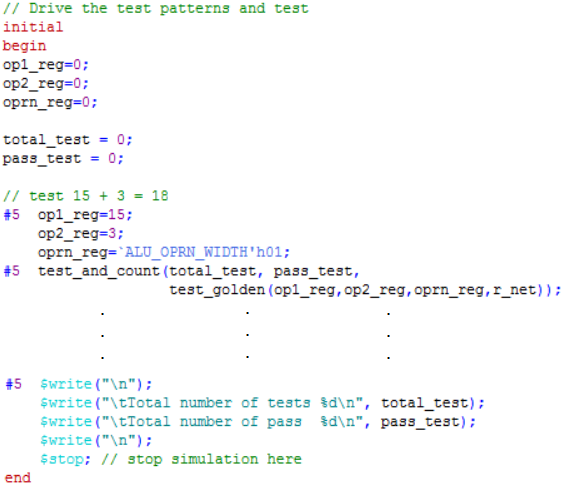
1. *Timing control*

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.

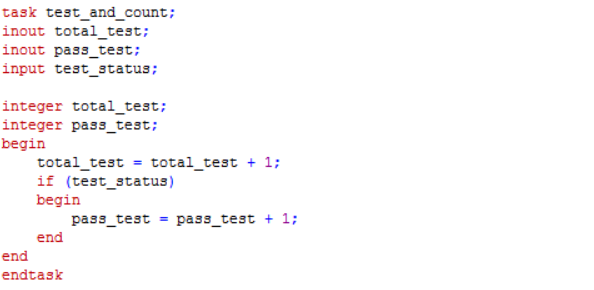
1. *ALU Testing*

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

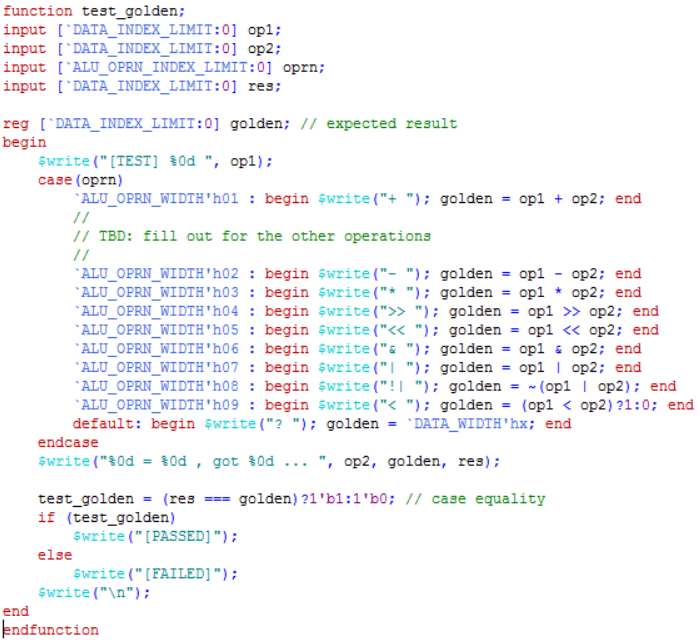
After initializing the integer representing the total number of tests cases, passed tests, and registers to a value of 0 for normalization purposes, test cases were created for each operation. Note that the majority of the test cases in the following code snippet were removed for readability purposes.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



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

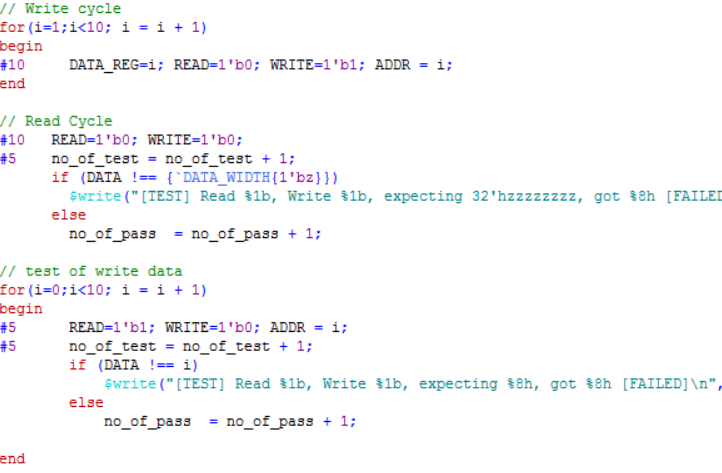


The “test\_golden” task tests the ALU result to a golden result depending on the operation code. Following the case block is the comparison of the golden result to the ALU result written as output. This task tests for case equality and writes “PASSED” or “FAILED” depending on the outcome of the test.

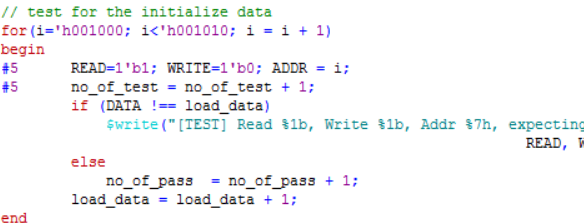


1. *Memory testing*

The test bench of the memory is named “mem\_64MB\_tb.v”. In this test bench, the testing is done by writing values into the memory and reading them to see if they are the equal. There is also a test for checking the Hi-Z state of the memory by setting the read signal to 0 and the write signal to 0.

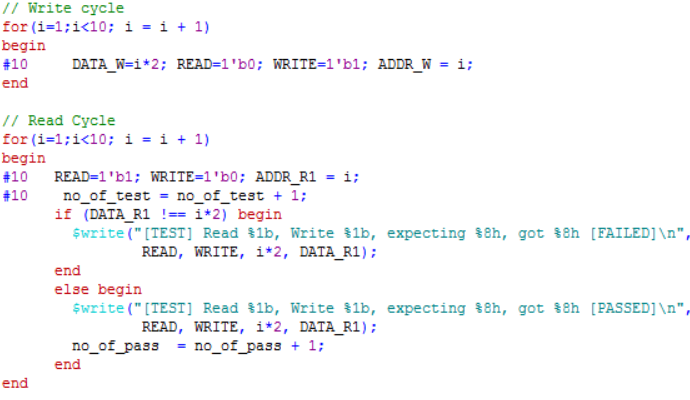


The memory also has the option to initialize data from a data file and that must be tested as well. In the beginning of the test bench, the initialization file is defined as “mem\_content\_01.dat”. The following snippet of code tests for the initialization of data, checking if the load\_data variable is equal to data in the memory.



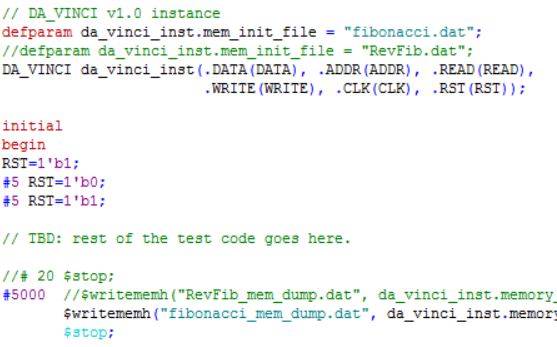
1. *Register file testing*

The testing of the register file is done in “reg\_32x32\_tb.v”. The testing procedure is similar to that of the memory without the test for initialization since data cannot be loaded into the register file from the reset signal. To test the register file, data is written into the register file and checked during the read cycle. If the data from read cycle is equal to the data supposedly written during the write cycle, the test will pass with 10/10.



1. *Testing for entire system*

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



# Conclusion

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

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

# References

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

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

1. [↑](#footnote-ref-1)