Arithmetic & Logic Unit (ALU)

David-Eric Thorpe

3554, SJSU

DavidEric.Thorpe@sjsu.edu.

Abstract: **This report mainly deals with the implementation of a behavioral model of a computer system with a 32-bit processor and 256MB memory that support the CS147Sec05 ISA using using verilog. This project report also includes the following:**

General Information

Table I.1: List of Tools Used:

Name	Company	Used for	Free?
ModelSim	Mentor Graphics	IDE, Simulation & testing	Yes (for students)

Requirements for System

A processor exists for the purpose of executing instructions and for that reason there are nearly as many instruction sets and processor's that implement them. AMD, IBM, Intel, Sun Microsystems for example all have this 1:1 relationship. We are also using a unique language for our unique processor called 'CS147sec05' instruction set which has the benefit of being streamlined for our purpose.

A full descriptions of the language can be found in the Lect_01_T.pdf document, through a list of the overed operations are provided below:

- Addition (by register and immediate)

- Subtraction (by by register)
- Multiplication (by register and immediate)
- Logical AND (by register and immediate)
- Logical OR (by register and immediate)
- Logical NOR (by register)
- Set less than (by register and immediate)
- Shift logical left (by shmat)
- Shift logical right (by shmat)
- Jump Register (by register)
- Branch on equal (by address)
- Branch on equal (by immediate)
- Load word (by immediate)
- Store word (by immediate)
- Jump to address (by address)
- Jump to link (by address)
- Push to Stack
- Pop from stack

Design and implementation of Memory

Using Verilog it is simple to implement a memory model. ModelSim performs the low level simulation like RTL level and Gate level implementation.

Design Strategy

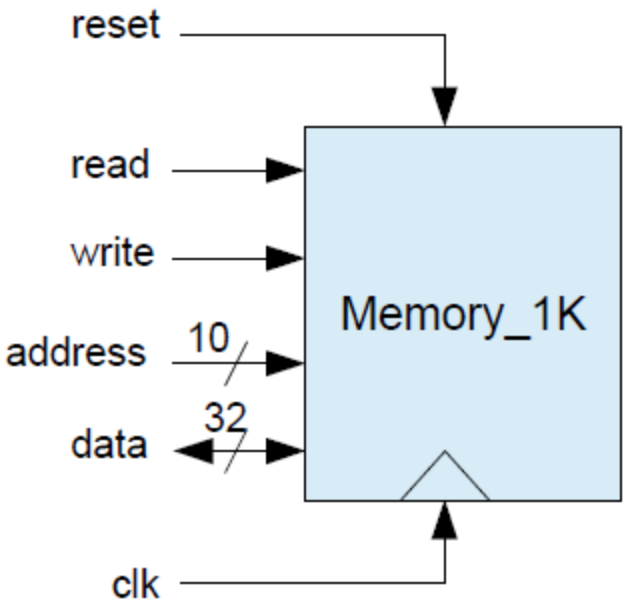
- Treat the memory unit as a module and note the i/o ports. See figure below. Incoming ports are: Read, Write, index (the address to be manipulated).
- There is also an InOut port required for info, the data.
- Outline

```
module memory (R, W, rst, I, D)
    input Read;
    input Write;
    input rst;
    input addr[0:9];
    inout Data[0:31];
    reg [0:31] data_str [0:63]
//operation code
endmodule
```

- We can see from above that the Read, Write, and rst controls are all single bit inputs. The address input I is 10 bits and the memory inout Data has a word size of 32 bits.
- It is also necessary to create the data structure to specifications, for which we have implimented a 2d array 32x64
- The remainder of the module saves the data structure to file per cycle, tests the Read & Write conditions to select Read Write or highZ operation and the performed memory

manipulation as required on the specified address, and upon reset signal writes the datastructure to 0.

NOTE: Regestry names here differ from the attached file for the sake of brevity.



Set of Operations of Memory

In this project the operations of memory are as follows:

1. Read (by address)
2. Write (by address)
3. Reset

- Each operation can be selected by putting the machine in the correct state. For example: a Read state is selected by setting Read=1, Write=0, and Address=Location_of_Manipulation. The result of the operation is returned along the D output.
- These operations described in the verilog code below:

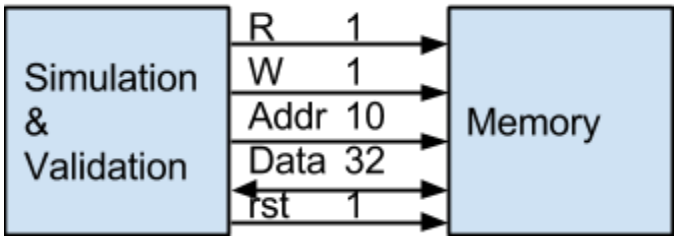
```
always @ (negedge RST or posedge CLK) begin
    if (RST === 1'b0) begin
      for(i=0;i<=`MEM_INDEX_LIMIT; i = i +1) begin
         sram_32x64m[i] = { `DATA_WIDTH{1'b0} };
      end
   $readmemh(mem_init_file, sram_32x64m);
   end
   else begin
      if ((READ===1'b1)&&(WRITE===1'b0))/*read*/ begin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

```
        sram_32x64m[ADDR] = DATA;
      end
    end
end
```

If the read write flags are conflicting, IE if they are 11 or 00 then the state highz will return.

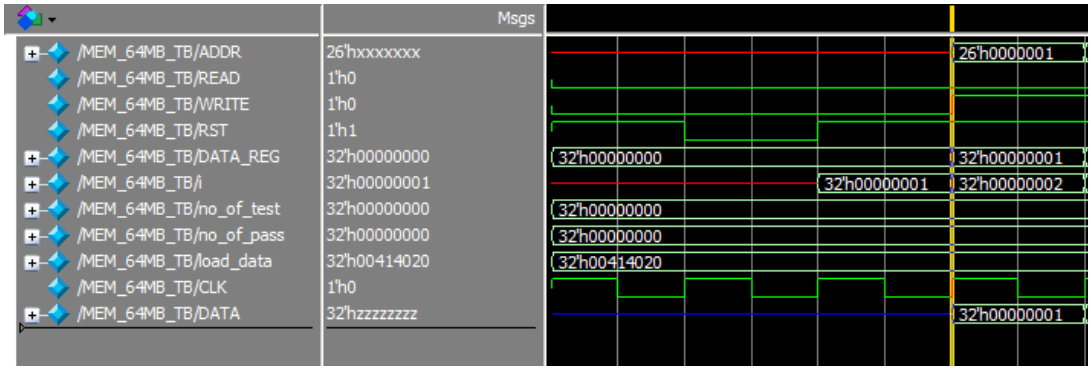
Test Strategy and Implementation for Memory

One the verilog code has been implemented the module should be tested to insure correct operation. Memory operation can be tested using the simulation function of ModelSim and custom Test Bench code and predefined input and expected output to be tested against. A simulation diagram is outlined below.

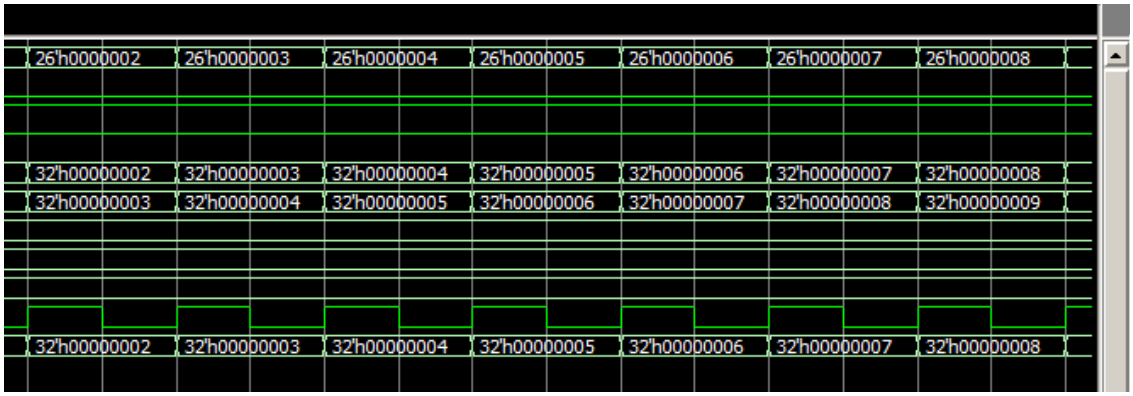


As shown the simulation test bench contains the values of the input regesters and will check the output from the Data registers.

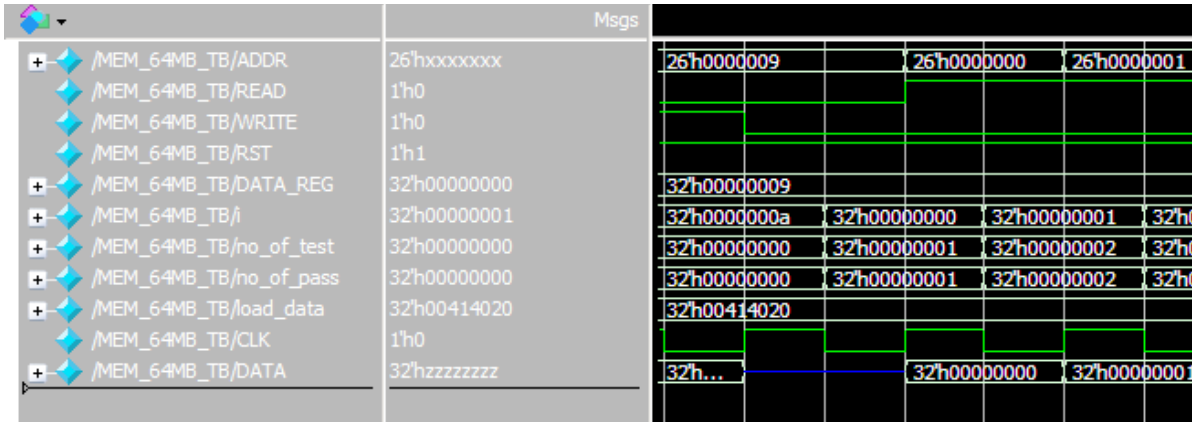
An operation has it's waveform shown below:



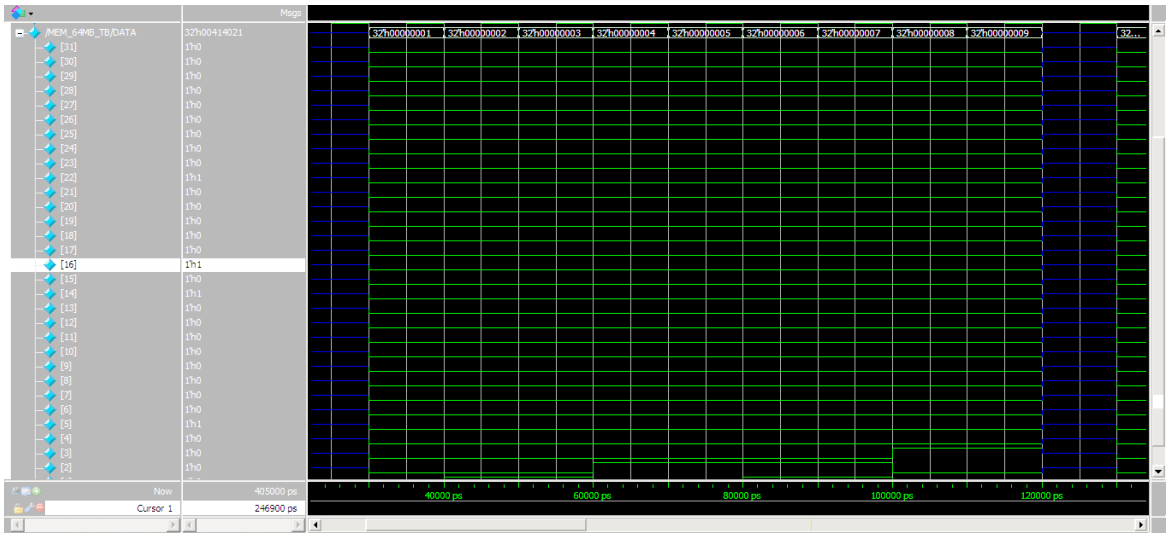
We can see here the initialisation of the memory, the rst signal, setting the operation to write and the beginning of writing file data into the memory data structure.



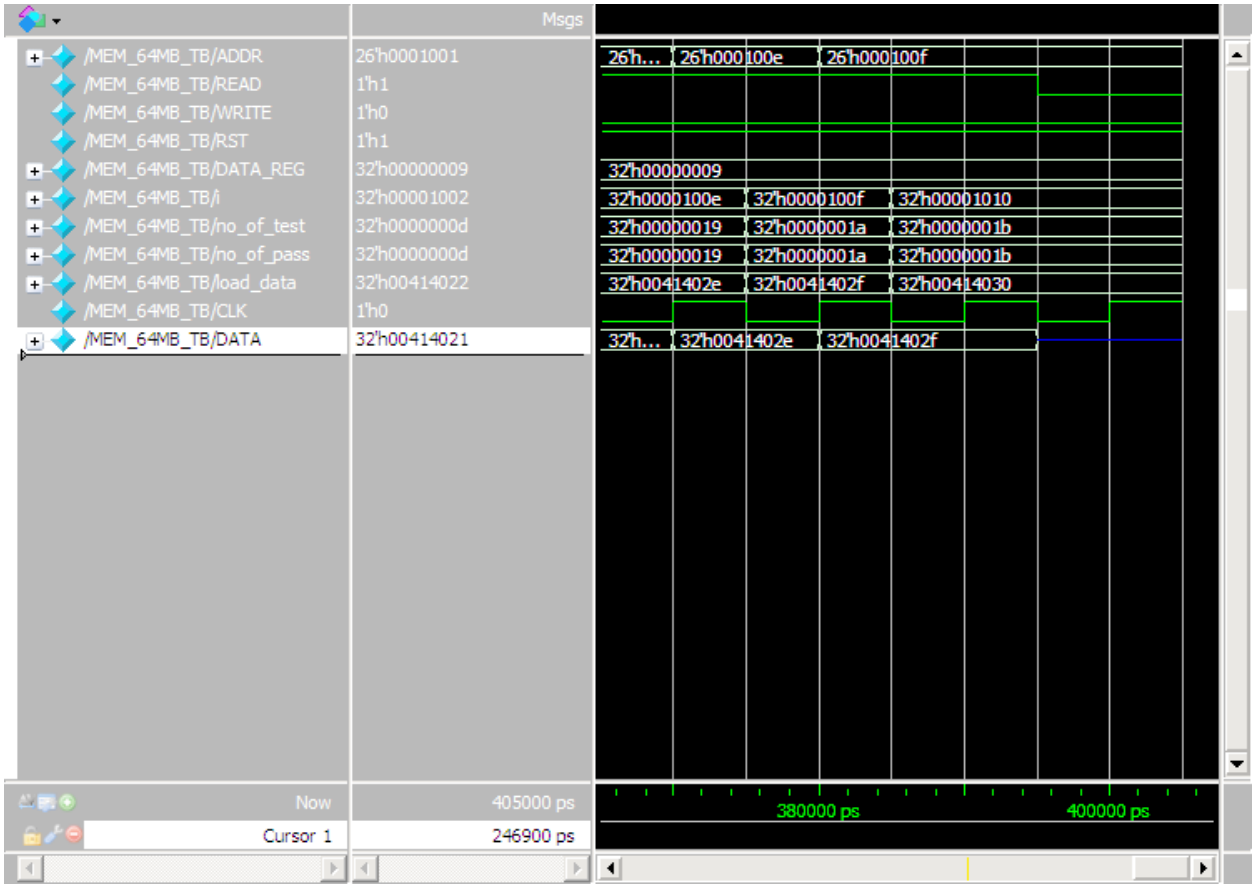
Seen here the process continues for some time.



Seen here the signal is changed from Write to Read and the index returns to start reading the data structure from the begining.



Here we can see the change in waveform as the file is being loaded into the memory. The blue sections represent when the system is in highZ, first at initialization and second for a short period of time while the read is switching to write.



Here we see that through the end of operation the number of tests run is equal to the number of tests passed.

Design and implementation of Processor

Using Verilog a processor is implemented as a summation of its components. Our processor instantiates a Control Unit, a 32x32 Register and an ALU.
NOTE: The ALU is discussed in separate ALU specific documentation. This implementation is the same with an additional output coded. This is a 1 bit Zero register, which allows us to flag operations whom result in zero.

Design Strategy

- Treat the processor as a module and note the i/o ports. See figure XXXXXXXXXX. Incoming ports are: Data, CLK and RST.
- There are out ports for ADDR, READ and WRITE.
- There is also an InOut port required for DATA.
- Outline

```
module memory (R, W, rst, I, D)
    input CLK;
    input RST;
    output READ;
    output WRITE;
```

```
    output ADDR[0:9];
    inout DATA[0:31];
    Control_Unit cu_init(...);
    Regester_File rf_init(...);
    ALU alu_init(...);
endmodule
```

- We can see from above that the R, W, and rst controls are all single bit. The address input I is 10 bits and the memory inout D has a word size of 32 bits.
- The remainder of the module initializes the three modules which are considered components of a processor:
    - Control Unit
    - Regester
    - ALU

NOTE: The Register file implementation is similar to the memory implementation, only the differences will be noted here and not as an individual section. Register addressing and writing is controlled with four inputs: (ADDR_R1, ADDR_R2, ADDR_W, and DATA_W). Register reading is controlled with the associated outputs (DATA_R1 and DATA_R2). There is no all encompassing DATA or ADDR. The data structure here is only a 32x32 2d array. Refer to memory for other implementation and testing.
NOTE: ALU implementation has been described in other documentation. Refer to ALU documentation for implementation and testing.

Control Unit Implementation

The Control Unit is the most complex piece of software in this project. Its purpose is to operate on various inputs, writing to and reading from the register and memory depending on the instructions given. As such the controller has various inputs and outputs.

- Inputs:
    - RF_DATA_R1/R2 is the register returning the contents of the register at a specified address.
    - ALU_RESTUL is the register returning the result of a specified ALU operation.
    - CLK is the system clock signal.
    - RST is the system reset signal.
    - ZERO specifies if the ALU operation produced zero.
- Outputs:
    - RF_DATA_W is the return from register of data at a specified address.
    - RF_ADDR_W/R1/R2 are the registers specifying the address needing to be returned.
    - RF_READ/WRITE are the singles for read and write operations for the register.
    - ALU_OP1/OP2/OPRN are the operators and operand for the ALU calculation.
    - MEM_ADDR specifies the memory address to be manipulated.
    - MEM_READ/WRITE are signals for read and write operations for the memory.
- Inout:
    - MEM_DATA is the data to be read from memory at the specified address.

NOTE: Because of Verilog syntax data written out to MEM_DATA can not be done so directly from within an always loop. Therefor an additional intermediary register must be declared, mem_data_ret/
The general format is thus

```
module Control_Unit(...)
   output [31:0] RF_DATA_W, ALU_OP1, ALU_OPT2;
   output [25:0] RF_ADDR_W, RF_ADDR_R1, RF_ADDR_R2, MEM_ADDR;
   output RF_READ, RF_WRITE, MEM_READ, MEM WRITE;
   input [32:0] RF_DATA, RF_DATA_R1, RF_DATA_R2, ALU_RESULT;
   input CLK, RST, ZERO;
   inout MEM_DATA;
   reg [25:0] PC_REG, INST_REG, SP_REG;
   assign MEM_DATA = ((MEM_READ==0)&&(MEM_WRITE==1))?mem_data_ret:{z};
   initial begin
      PC_REG = 000001000;
      SP_REG = 03ffffff;
   end
   //operation code
endmodule
```

```
module PROC_SM
   input, CLK, RST;
   output STATE;
   //state machine code
end module
```

- We see from above the registers carrying data have 32 bits.
- Registers carrying address have 26 bits.
- CLK, RST and ZERO all single bit signals.
- The mem_data_ret conditional assignment is required for Verilog because within the always loop assignments can not be directly made to inout registers.
- The remainder of the module is associated with various operations as the state of the processor changes
- The additional module is used to control the state machine. Advancing the machine on the clock cycles to the next state via a “look ahead” method.

Set of Operations of Processor

There are no operations of the processor beyond the component instantiation.

NOTE: The Register file operation is similar to the memory operation. Only it does not handle READ=WRITE states, allowing it to retain previous data.
NOTE: ALU operation has been described in other documentation. Refer to ALU documentation for implementation and testing.

Control Unit Operations

A. Main module:

a. Fetch- This prepares the Mem for reading from the correct program counter address and prepares the register for non operation.
b. Decode- This retrieves the next instruction from memory, divides the instruction into component parts, pads and extends those parts as future manipulation will require, preliminarily sets the most common register addresses and prepares the register for read access.
c. Exe- With a case switching on opcode this performs the initial ALU operations when needed. Outputs an error statement if the instruction is not found.
d. Mem- Performs memory operations via small inclusive case switching. Load word, store word, push and pop. For these the ALU results are applied and operations not requiring register modification are completed. The state pointed is also advanced to ISA specification.
e. Write Back- The register is set to write and using an inclusive and exclusive case statement any remaining operations and all register write backs are performed. The program counter is advanced in the end.

```
case(proc_state)
`FETCH
   MEM_ADDR=PC_REG;
`DECODE
   {opcode, rs, rt, immediate} = INST
   {opcode, address} = INST
   {opcode, rs, rt, rd, shmat, funct} = INST
//various padding assignments
`EXE
   case(opcode)
   r-type:
      case(funct)
      //various r-type tests and operations
      default: $write(“error”);
  //individual J and I type operations
  default: $write(“error”);
`MEM
   MEM_READ = MEM_WRITE = 0;
   case(opcode)
   Load Word:
      //mem opperations
   Store Word:
      //mem opperations
   Push:
      //mem opperations
   Pop:
      //mem opperations
   default: //not used
`WB
   RF_READ=0;
```

```
RF_WRITE=1;
case(opcode)
    r-type:
        if(not jr) RF_DATA = ALU_RESULT
        else PC_REG = RF_DATA_R1
    beq:
        if(ZERO == 0) PC = PC + BranchAddress;
    bne:
        if(ZERO != 0) PC = PC + BranchAddress;
    sw://does nothing here
    jmp:
      PC_REG=jumpAddress - 1; (for wb closing offset)
    jal:
      RF_ADDR_W = 31;
      RF_DATA_W = PC_REG + 1;
      PC_REG=jumpAddress - 1; (for wb closing offset);
    push: SP_REG = ALU_RESULT
    pop: RF_DATA_W = MEM_DATA;
    lui: RF_DATA_W = LUIimmediate;
    lw: RF_DATA_W = ALU_RESULT;
    default: //handles addi, muli, andi, ori, lui, slti
       RF_ADDR_W = rt;
       RF_DATA_W = ALU_RESULT;
 endcase
PC_REG++;
```

```
case (NEXT_STATE)
      `PROC_FETCH :
      begin
         STATE = NEXT_STATE;
         NEXT_STATE = `PROC_DECODE;
      end
      `PROC_DECODE :
      begin
         STATE = NEXT_STATE;
         NEXT_STATE = `PROC_EXE;
      end
      `PROC_EXE :
      begin
         STATE = NEXT_STATE;
         NEXT_STATE = `PROC_MEM;
      end
      `PROC_MEM :
      begin
         STATE = NEXT_STATE;
         NEXT_STATE = `PROC_WB;
      end
```

```
`PROC_WB :
begin
   STATE = NEXT_STATE;
   NEXT_STATE = `PROC_FETCH;
end
default:
begin
   STATE = 2'bxx;
   NEXT_STATE =`PROC_FETCH;
end
endcase
```

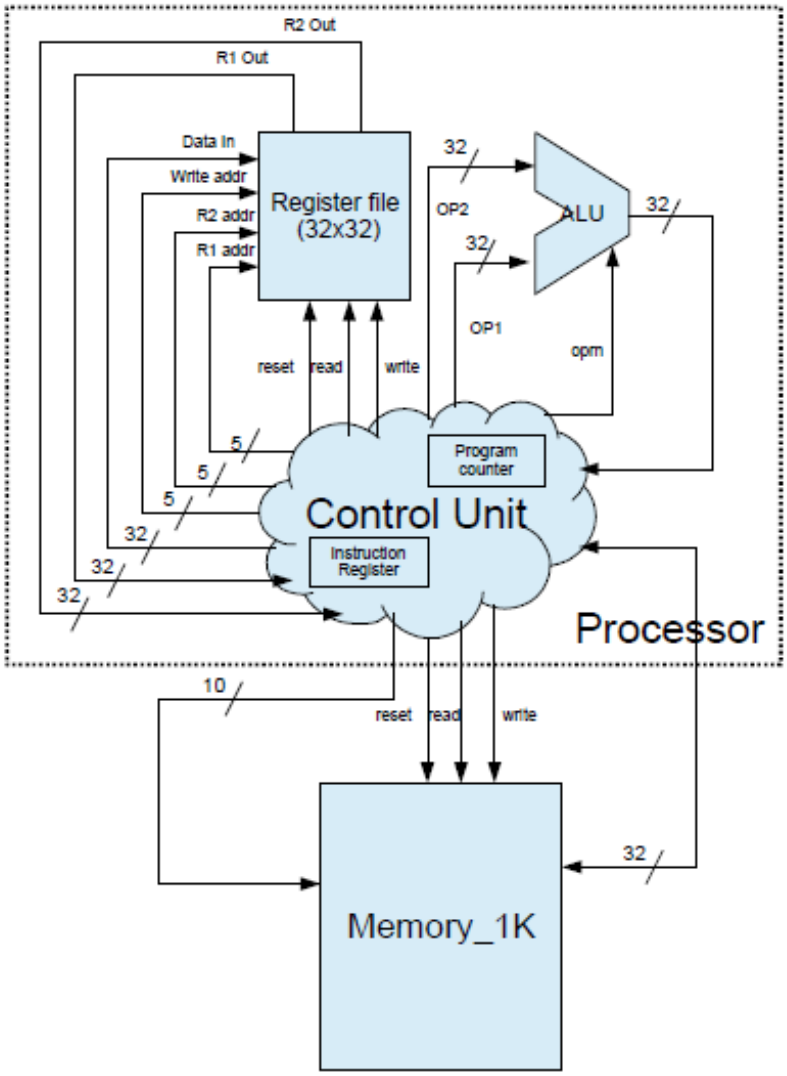
There is no specific end case for this program. Most inputs will leave a great deal of memory free and will result in a large amount of empty instructions. Otherwise the simulation is controlled in the testbench which will usually time out after 5000 milliseconds. In real application the system would enter an idle process and while awaiting further instructions would run a loop indefinitely.

Test Strategy and Implementation for Processor

One the verilog code has been implemented the module should be tested to insure correct operation. The control unit should be tested with the Memory module, as a lesser test would have little benefit and be as challenging to implement. In order to test this system both must be initialized and the Memory loaded with data for which the result is known. For this we implement a Da_Vinci machine with:

- an address, read and write output
- a clock and reset input
- and a data inout

This machine also initializes the processor and memory components. Seen below:



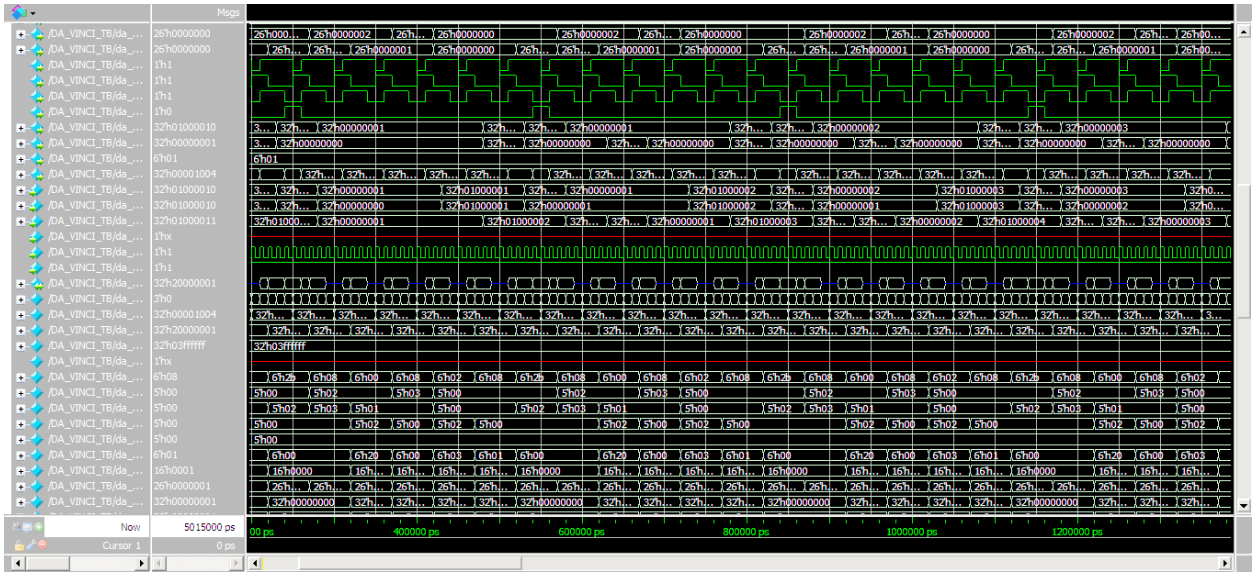
Test Strategy and Implementation for Di Vinci System

A Test Bench can now be built to test the entire system.

Three tests implemented were:

- The fibonacci sequence- this tested various i and j type operations
- Reverse fibonacci sequence- this further tested various i and j type operations
- R-type sequence- this tested the remaining untested r-type eoperations

At this point due to the density of data the waveform becomes less effective to review the simulation.



We can compare the mem dump files to expected “golden” output:

// memory data file (do not edit the following line - required for mem load use)
//
instance=/DA_VINCI_TB/da_vinci_inst/memory_inst/sram_32x64m
// format=hex addressradix=h dataradix=h version=1.0 wordsperline=1 noaddress
00000000
00000001
00000001
00000002
00000003
00000005
00000008
0000000d
00000015
00000022
00000037
00000059
00000090
000000e9
00000179
00000262
000003db
	// memory data file (do not edit the following line - required for mem load use)
	//
	instance=/DA_VINCI_TB/da_vinci_inst/memory_inst/sram_32x64m
	// format=hex addressradix=h dataradix=h version=1.0 wordsperline=1 noaddress
	00000000
	00000001
	00000001
	00000002
	00000003
	00000005
	00000008
	0000000d
	00000015
	00000022
	00000037
	00000059
	00000090
	000000e9
	00000179
	00000262

We can see the expected output matches the actual output for the fibonacci sequence.

// memory data file (do not edit the following line - required for mem load use)	// memory data file (do not edit the following line - required for mem load use)

// instance=/DA_VINCI_TB/da_vinci_inst/memory_inst/sram_32x64m // format=hex addressradix=h dataradix=h version=1.0 wordsperline=1 noaddress 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 0000000f 00000002 00000008	// instance=/DA_VINCI_TB/da_vinci_inst/memory_inst/sram_32x64m // format=hex addressradix=h dataradix=h version=1.0 wordsperline=1 noaddress 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 0000000f 00000002 00000008

Likewise for our remaining simulated operations.

Conclusion

From project 2 I have dramatically improved my understanding of the Verilog language, the ModelSim IDE, and the operation of a simple computer. I have learned the importances of the sequence of operations in relation to the clock cycles and the division of operation for a command execution. Most importantly my conceptualization of the system has a whole has shifted from a process “recipe” view to a system state, in which a process is driven by the clock but the entire system state shifts instantly with each cycle.