Da Vinci System, Gate Level Model

David-Eric C. Thorpe

SJSU, CS 147-05
DavidEric.Thorpe@SJSU.edu
ID#3554

***Abstract*— This report deals with the theory and implementation of a bare minimum gate level and behavioural level model of a computer system. This computer system consists of a 32-bit processor, Arithmetic Logic Unit, Register File and shared Memory of 64MB. The processor supports the CS147Sec05 Instruction Set. This system is implemented using the language Verilog within the IDE ModelSim. Here-in this is referred to as a Da Vinci System.**

I. Da Vinci System

A. Programs used

Name	Developer	Used for	Free?
ModelSim	Mentor Graphics	IDE, Simulation & testing. Code.	Yes (for students)
LogiSim	Dr. Carl Burch	Simulation and testing. Visual Based	Yes

B. esign

A Computer System is a general purpose machine that uses digital circuits to manipulate stored data via gates. It reads instructions from memory and writes results to memory, in a mathematical and predictable manor. It takes advantage of the ability to quickly change Voltage levels to perform calculations in an extremely fast manor.

C. Implimentation

Our Da Vinci model is optimized to best illustrate the operation of a computer. As such it features discrete components with singular purposes and is ideal for the purpose of education of implementation and operation. No emphasis is placed on gate minimization, power requirements, and physical circuit design constraints.

1) Major Components

The Da Vinci model is optimized to most clearly illustrate the operation of a computer. Components include:

- Arithmetic Logic Unit (32-bit)
- Memory (64MB)
- Clock (0.2GHz)
- Processor (32-bit, 32 registers)
- Register (32-bit)
- Data path (32-bit)

2) CS147Sec05 Instruction Set

A Da Vinci System is designed to operate on the CS147Sec05 Instruction Set. This brief set of operations is optimized for educational purposes and consideration is taken for clarity and simplicity. CS147Sec05 features the following properties:

- 32-bit instructions.
- Address space implicit
- Explicit registers lists (32)
- Word addressable (32-bit)
- Discrete Stack Pointer Register
- Discrete Program Counter
- Big Endian
- Optimized for human simplicity and clarity
- MIPS

The following tabes describe CS147Sec05

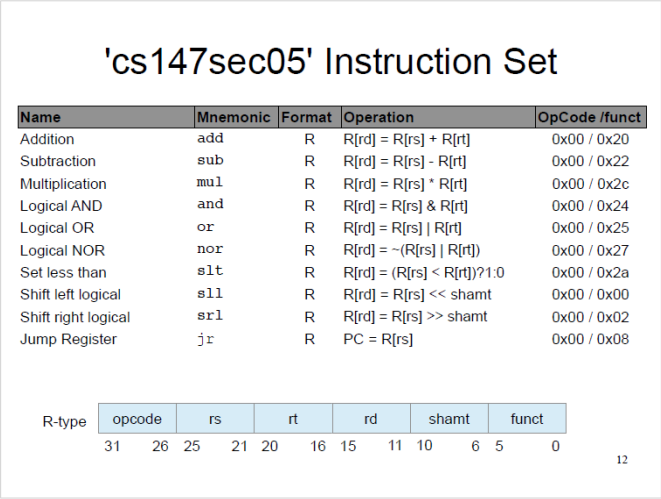
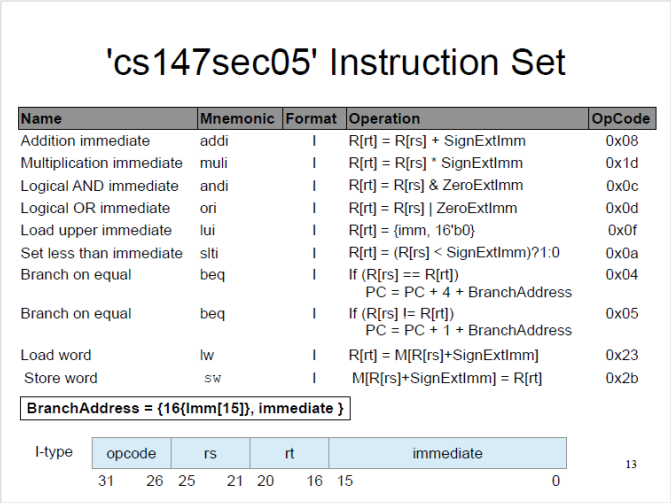


Fig. R-Type Register Instructions

Instructions, with the exception of jr, are generally register-to-register. Operand Register locations are defined in the rs and rt codes. Results Register locations is defined in the rd code. The funct is used explicitly for ALU operation. Shamt is used for bit shift quantity. When jumping is required for program looping jr can manipulate the PC register. The opcode determines R-type and funct the particular instruction.



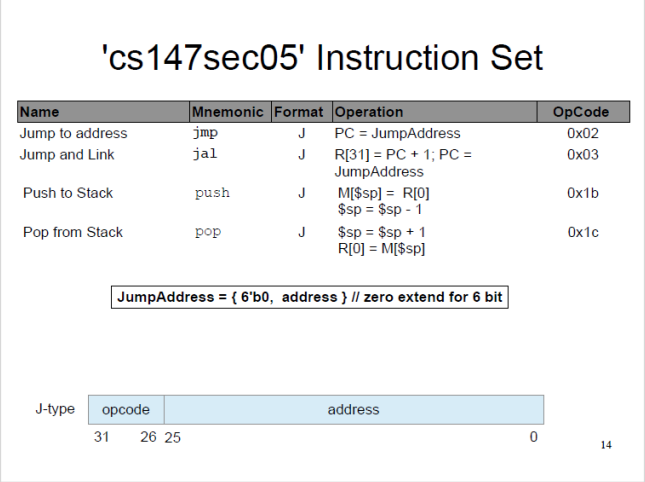
'cs147sec05' Instruction Set

Name	Mnemonic	Format	Operation	OpCode
Addition immediate	addi	I	R[rt] = R[rs] + SignExtImm	0x08
Multiplication immediate	muli	I	R[rt] = R[rs] * SignExtImm	0x1d
Logical AND immediate	andi	I	R[rt] = R[rs] & ZeroExtImm	0x0c
Logical OR immediate	ori	I	R[rt] = R[rs] | ZeroExtImm	0x0d
Load upper immediate	lui	I	R[rt] = {imm, 16'b0}	0x0f
Set less than immediate	slti	I	R[rt] = (R[rs] < SignExtImm)?1:0	0x0a
Branch on equal	beq	I	If (R[rs] == R[rt]) PC = PC + 4 + BranchAddress	0x04
Branch on equal	beq	I	If (R[rs] != R[rt]) PC = PC + 1 + BranchAddress	0x05
Load word	lw	I	R[rt] = M[R[rs]+SignExtImm]	0x23
Store word	sw	I	M[R[rs]+SignExtImm] = R[rt]	0x2b

BranchAddress = {16{Imm[15]}, immediate }

Fig. I-Type Immediate Instructions

Instructions are to-register-from-immediate, control statements, and memory operation. An Operand Register locations is defined in the rs, rt as well for control statements. Result register is defined by rt, with the exception of sw where this defines the read location. Branch operations have the ability to mutate the Program Counter register. The opcode determines the particular instruction.



'cs147sec05' Instruction Set

Name	Mnemonic	Format	Operation	OpCode
Jump to address	jmp	J	PC = JumpAddress	0x02
Jump and Link	jal	J	R[31] = PC + 1; PC = JumpAddress	0x03
Push to Stack	push	J	M[$sp] = R[0] $sp = $sp - 1	0x1b
Pop from Stack	pop	J	$sp = $sp + 1 R[0] = M[$sp]	0x1c

JumpAddress = { 6'b0, address } // zero extend for 6 bit

Fig. J-Type Jump Instructions

Instructions manipulate the Program Counter, Stack Pointer and memory. These can only influence the 0¬th and 31¬th registers. The opcode determines the particular instruction.

3) Diagram of Design

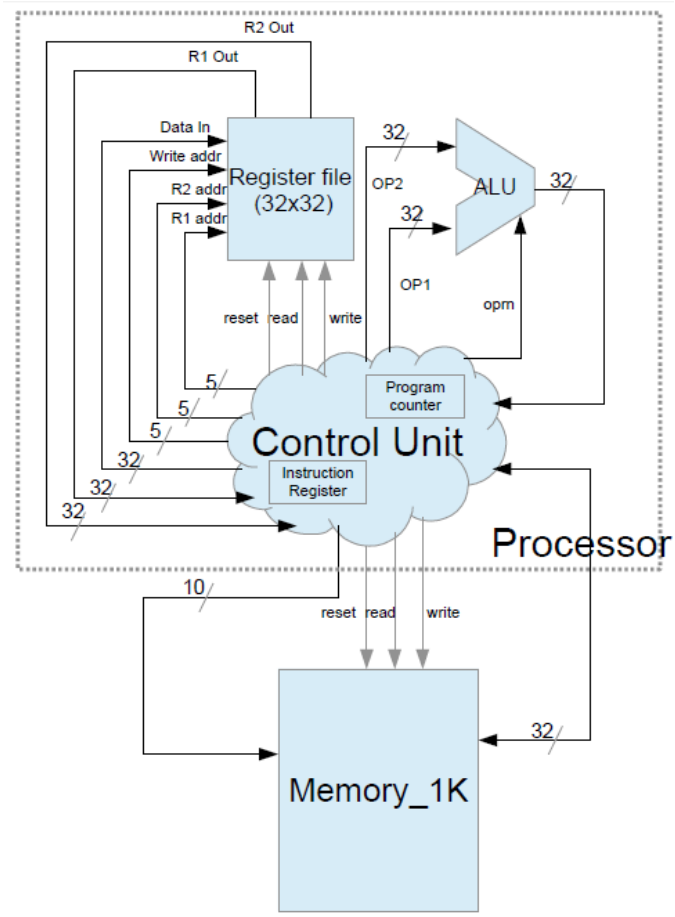


Fig. In this schematic many wires for control signal are not shown nor is the data path shows as a discrete unit. A large changes in implementation was Memory inout was changed to a 32-bit in and 32-bit out.

D. Initialalization

The following outline steps to create a Da Vinci System from the included files with this document utilizing the software listed above.

1) Files

Unpack the ThorpeProject3.zip in your working directory. It should include the many files, each file contains operation modules and test bench modules:

Alu.v- module for logical operations (gate level implementation).
Barrel_shifter.v- Left/right bit shifting operations (gate level).
Clk_gen.v-
Logical operations (behavioural implementation).
Control_unit.v-
Machine state and control signal generation (behavioural).
Da_vinci.v-
System instantiation.
Data_path.v-
Data and signal routing (gate level).
Logic.v-
Various low level support (gate level).

Memory.v-
Data storage structures (behavioural).
Mult.v-
Mathematical multiplication operations (gate level).
Mux.v- Selecting from a range of inputs (gate level).
Prj_definition.v-
Various definitions used in this project for readability.
Processor.v-
Control Unit and Data Path instantiation.
Rc_add_sub_32.v –
Addition subtraction operations (gate level).
REG32_PP.v-
SP and PC register setup (gate level).
Registser_file.v-
32x 32-bit register file and associated inouts (behavioural).
Fibonacci.dat-
Sample program for generating a Fibonacci sequence.
RevFib.dat-
Sample program for generating a Reverse Fibonacci sequence.
Fibonacci_mem_dump.golden.dat-
The expected results of running Fibonacci.dat
RevFib_mem_dump.golden.dat-
The expected results of running RevFib.dat
a_*_tb.v-
These are the test bench files associated with the *.v files.
*.circl
These are Logism GUI circuit models

2) Project Creation

Open ModelSim and create a New Project, File -> New -> Project

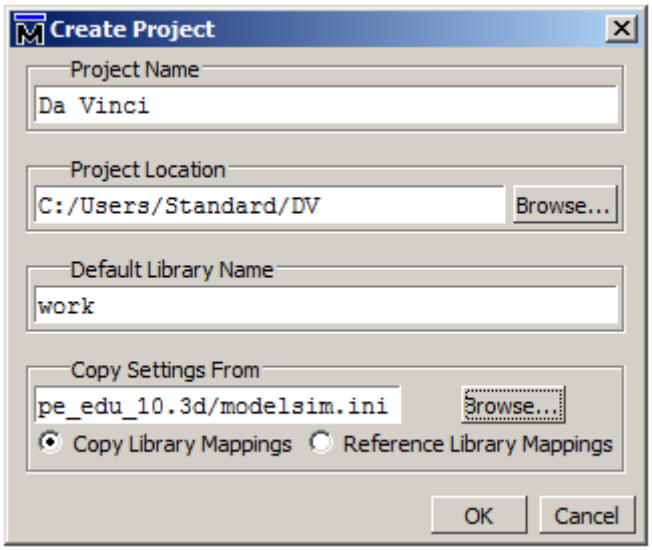
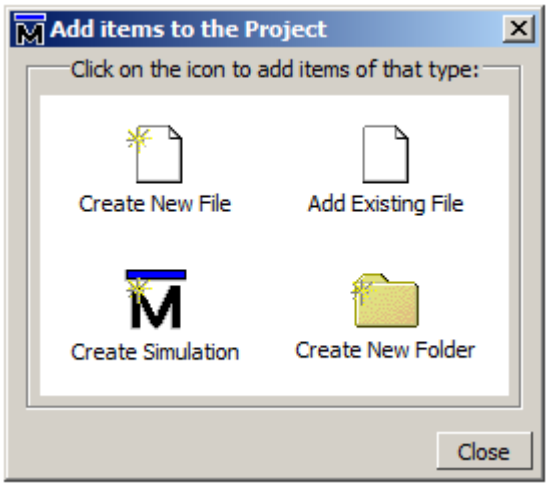
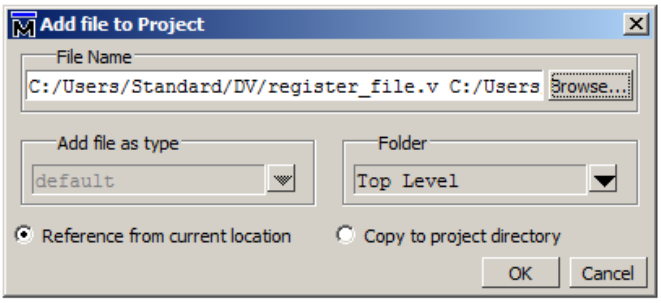
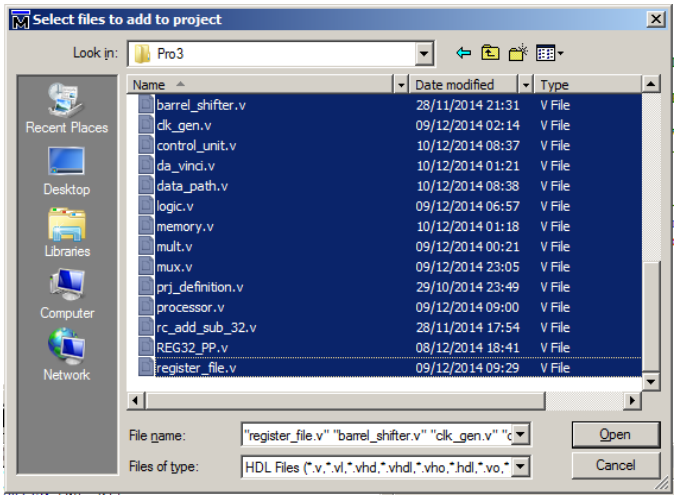


Fig. 4. Project Creation

Add files to the project via “Add Existing File”. Browse to the extraction directory and

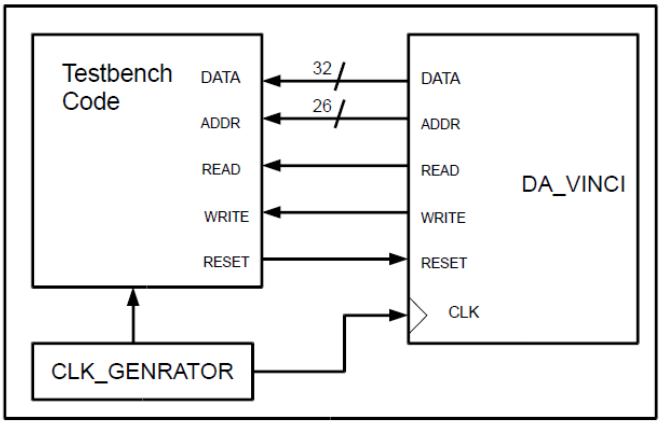






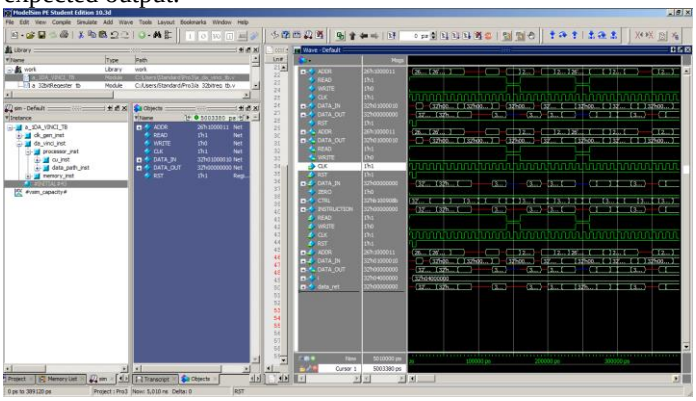
3) Testing

DaVinci v1.0 Testbench

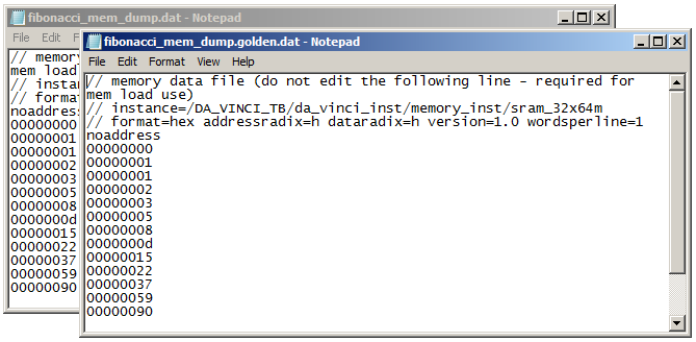


To start the simulation choose Simulate->Run->Run –all. To view various changes in the system state you must add components to the Wave Viewer. These are selectable via the sim window or the Objects window. The sim window can also be used to launch an instance of any module with relational value in alt text to tracked variables. A toggle name <-> leaf name button found in the Wave window can be beneficial for clarity.

The Da Vinci Test Bench does not need these tools to affirm activity nor does it print unique results into Transcript. All results will be found in the program director via .golden.dat file. Because it is too much information to effectively parse it is best to run it through controlled operations and check for expected output.



As we can see it is challenging to asertain meaningful data from this, though we can see the Da Vinci System is very active in all components as expected. Any given time can me measured to see the components state, which is useful for analysing individual components during runtime.



We can see here the output and expected output from a sample run of a fibonacci sequence. This is an effective test and we can see the Da Vinci System is performing as expected.

```
        // 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_IN(DATA_IN), .DATA_OUT(DATA_OUT), .ADI
                    .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.mem
//                    $writememh("fibonacci_mem_dump.dat", da_vinci_inst.m
                    $stop;
```

The test bench is very short to impliment. Included in ours are two prepaired squences and their expected result. Fibonaci and RevFib.

II. Design and Implementation of Memory

A. *Design*

Without memory we have no computer. It is the input and output of i/o. It is the state of the system. Memory can range in many different medium balancing access speed, cost pet bit and stability. Old punch cards are extremely slow, but extremely cheap and quite stable. Other materials include magnetic tape, CD, HDD, SSD, and RAM. In a general sense the as the cost per bit increases the speed increases. In order to take advantage of low cost but fast access we can employ a cache to minimize the times the system will have to access slower media for data that doesn't fit on the more costly media.

B. *Implementation*

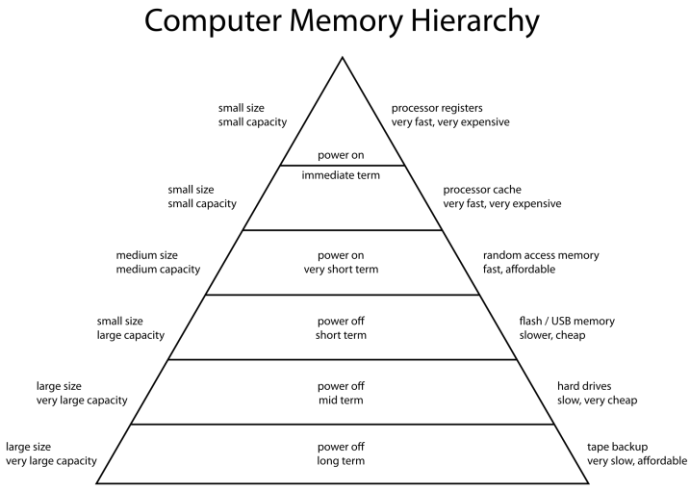
In our project memory is in two distinct systems. Memory, which is a shared space for instructions and data stored and Registers. Though we do not have a difference in access speed between main memory and the registers.

For this reason and for simplicity we also do not employ data cache.

1) Main Memory Design

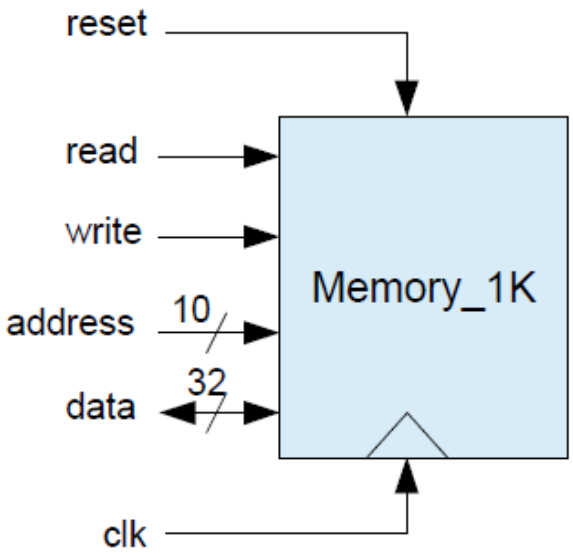
Memory provides storage for instructions and output from a processor. Without memory the processor state does not produce new results. Memory comes in many different formats. From punch cards to RAM. There is a tradeoff

between the cost of manufacture for each bit and the access speed. Because of this we use low cost/bit but slow access memory for 99% of our storage needs. We then cache that memory into smaller and small caches of higher cost/bit faster access memory. In a well-designed cache we will minimize the miss rate that requires a piece of data moved from slow access to faster access therefor taking advantage of high performance at lower prices.



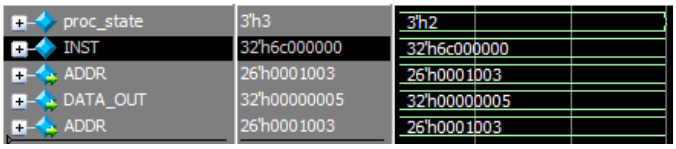
2) Main Memory Implimentation

Main Memory is implemented behaviorally through an array of 32-bit words. It has Read and Write controls for access.

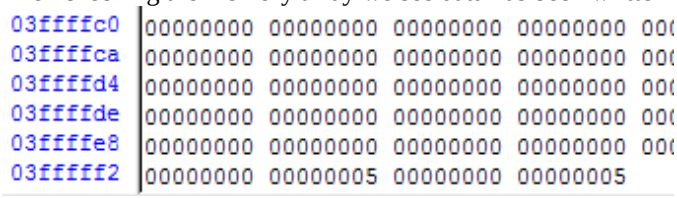


This design differs from out implementation slightly as we provide greater capacity requiring 26 bit addressing, and we split data input and output.

Testing of memory is done through the full system test of Da Vinci, after all without memory we would have no computer! By loading the Da Vinci Test bench and loading the instantiated modules we can assign a break point to the control unit at the end of memory. We assign proc code, instruction and Data Out to the wave form and then cycling through a few full changes of machine state we pass a call to push. Opcode 0x6c



Then checking the memory array we see data has been written

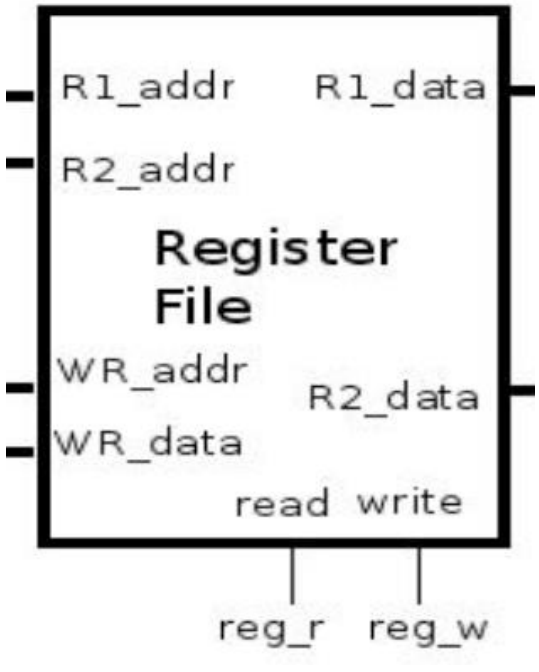


3) Register Design

Registers sit on the peak of the earlier pyramid. They are extremely expressive per bit but very fast. For this reason there are very few bits available in any given system but they are very well taken advantage of.

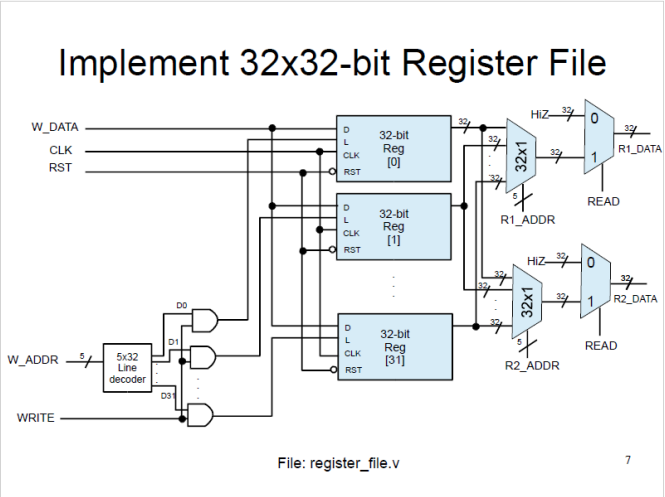
4) Register Implimentation

Our registers are implemented through a gate model. The Register File unit can manipulate three registers, two for reading and one for writing. Our register file is implemented behaviorally.



Our register file impimentation can perform opperations with three registers at different regester addresses. One for writing and two for reading. It also has a reset function to return it to an initial state and

Ours should be implimented a gate level, but is at behavioral. Gate level and has a number of interesting descrete components



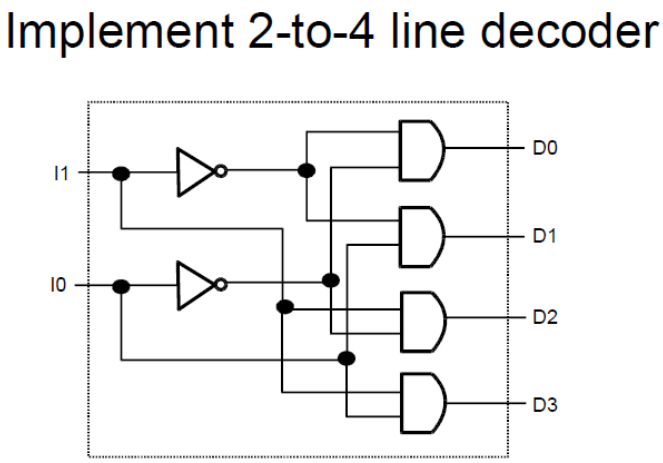
From left to right we observe decoders, and gates, single 32-bit registers, and 32x1 multiplexers

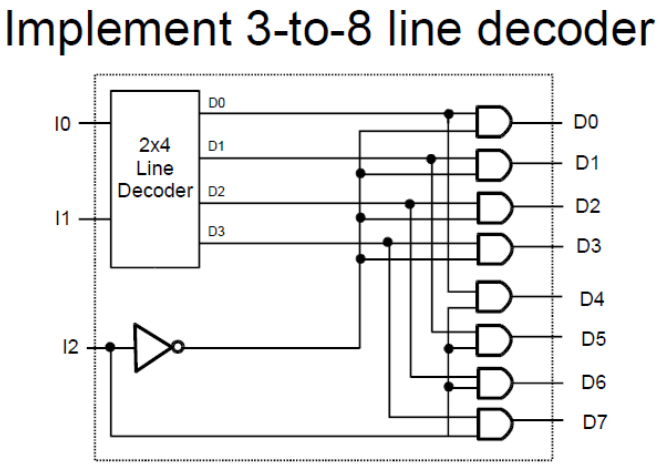
a) Decoder Design

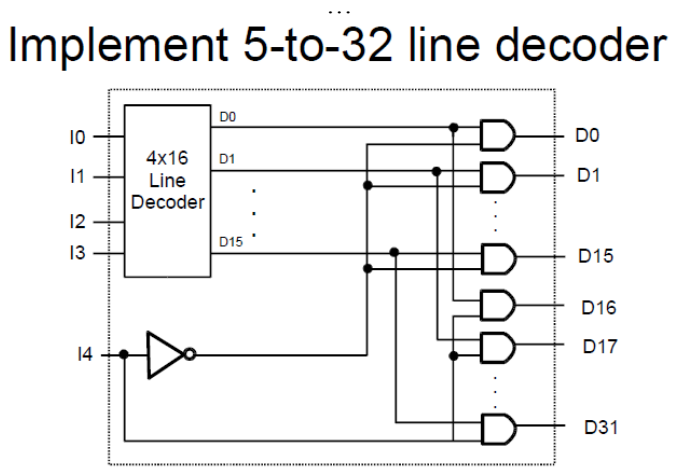
A decoder is used to make active a wire based on its input. We use the decoder here to select which register will receive data from an address signal when combined via AND with a write signal.

b) Decoder Implimentation

We are utilizing a 5x32 decoder here but it is the same as a 2-4 decoder by a factor of recursion.







File: logic.v

c) Decoder Testing

Because of the simplicity and simplicity of scalability testing was postponed until 5x32.

Testing only involved checking that 0 selected the first bit, 1 selected the 2nd and so forth:

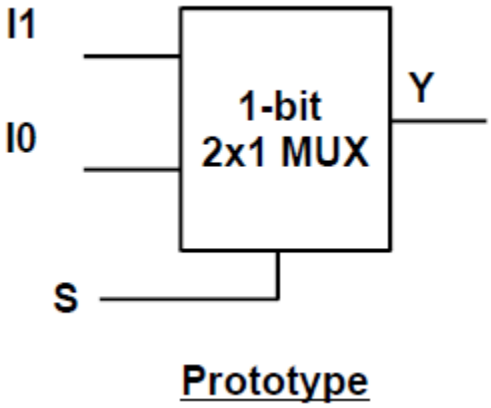
```
initial begin
        #5;
        #5 control='b0;
        #5 golden(result,'b1, control);
        #5;
        #5 control='b1;
        #5 golden(result,'b10, control);
        #5;
        #5 control='b10;
        #5 golden(result,'b100, control);
end
```

```
#          1 got          1, control  0[PASSED]
# Run line decoder 32 TB
#          2 got          2, control  1[PASSED]
# Run line decoder 32 TB
#          4 got          4, control  2[PASSED]
# Run line decoder 32 TB
```

Ehaustive testing was possible but not implimented due to the simplicty of this design

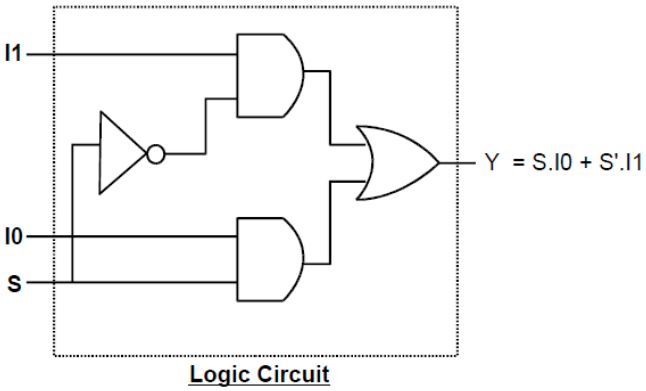
a) Mux Design

A multiplexer passes through an input based on a control signal. The most basic form of multiplex is a 2x1 and will choose one bit based on the control. Further multiplexers variations simply use arrangement of single multiplexers to achieve 32x1 or 32-bit 2x1 etc.

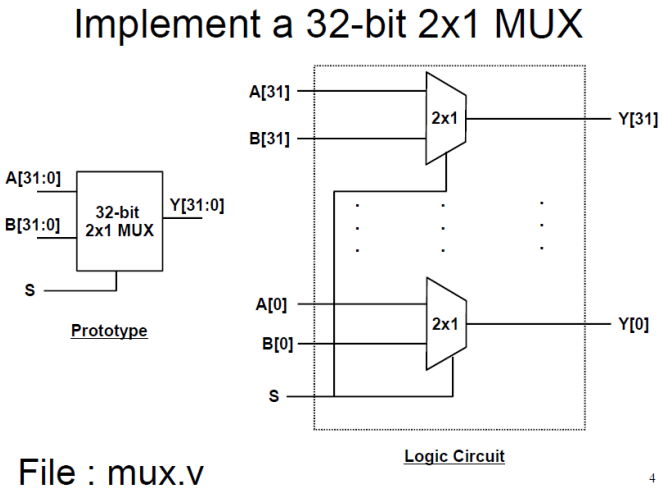


b) Multiplex Implimentation

In the decoder we use 2 types of mux. 32x1 and 32-bit 2x1. To select first from among 32 1-bit inputs and then between 2 32-bit inputs. The basic component is 2x1

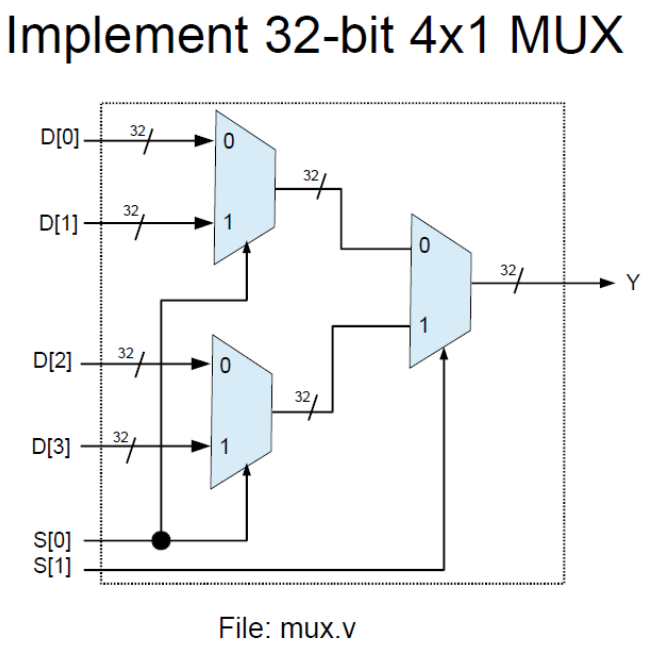


and summed to form a more complex MUX



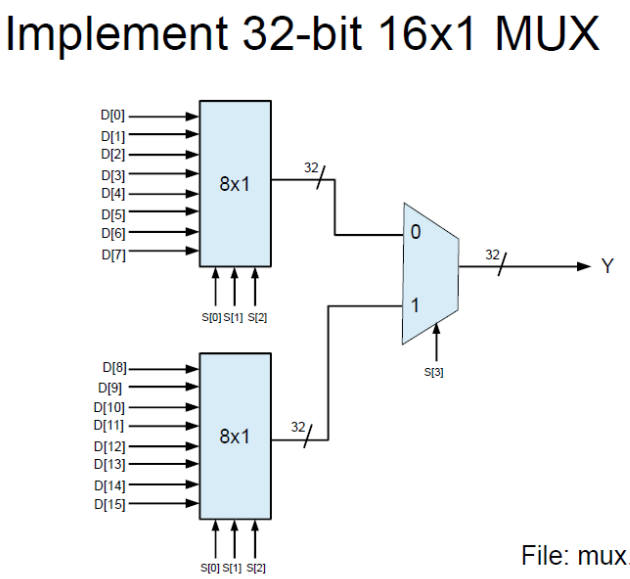
File : mux.v

Much like the decoder mutiple input Mux scale recurively



File: mux.v

…

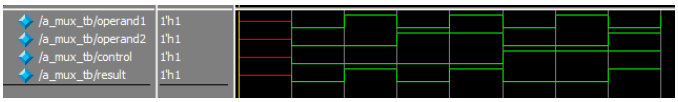


File: mux.v

c) Decoder Testing

Exhaustive testing was undertaken due to the frequency of use and only on the 2x1. Since the base worked Extensive testing was not required for larger multiplexers.

```
mux m1(.result(result), .operand1(operand1),
       .operand2(operand2), .control(control));
initial begin
       #5 control=0; operand1=0; operand2=0;
       #5 control=0; operand1=1; operand2=0;
       #5 control=0; operand1=0; operand2=1;
       #5 control=0; operand1=1; operand2=1;
       #5 control=1; operand1=0; operand2=0;
       #5 control=1; operand1=1; operand2=0;
       #5 control=1; operand1=0; operand2=1;
       #5 control=1; operand1=1; operand2=1;
end
```



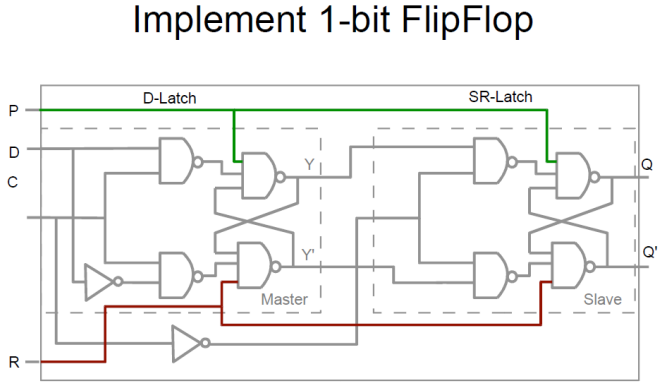
d) 32-Bit Register Design

A register is an interesting structure, in that its most basic unit, the flip flop, uses mutual input-output to make its state depend on its previous state. This allows for data storage!

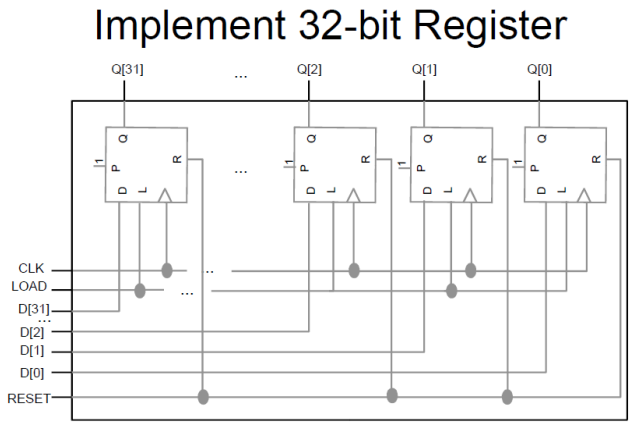
e) Register Implimentation

Currently we are using a behavioral implementation, which stores the data in a language structure register and returns it via controls.

A gate implementation begins with the flip-flop.



With a bit storted we can adding more for more bits of storage. A multiplexer for each allows and holding the preset permenently high us to avoid occilating input but looping back output the result.

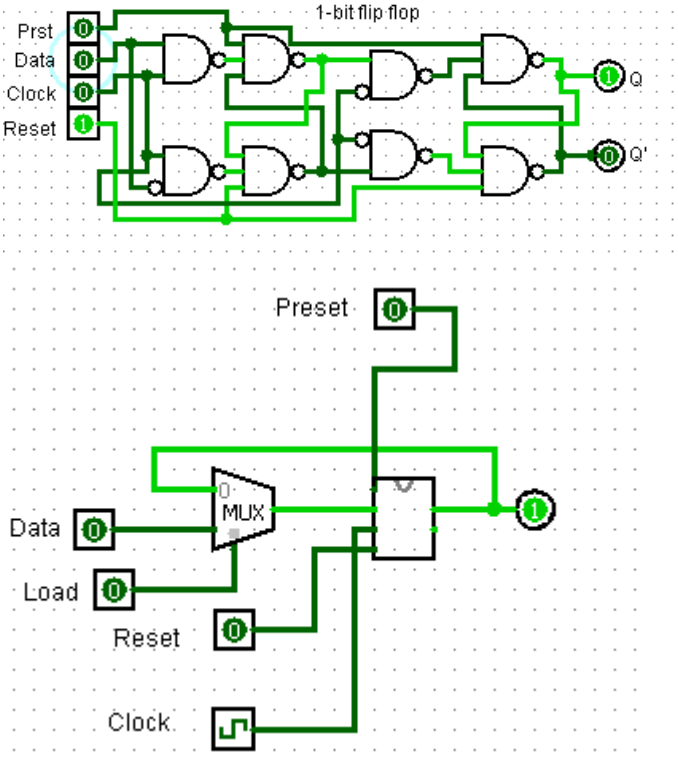


32x 32-bit registers makes our Register file.

f) Testing and Simulation

Testing was challenging for this because of its negative edge relationship to the clock. To verify our code results with the circuit behavior we utilized the GUI simulation tool Logisim. Replacing the input C with a clock input we can simulate a

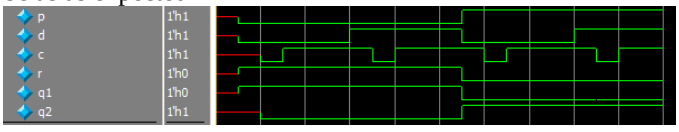
rythmic progression of time with variable changes.



See attached flip 1-bit flip flop.circ for flip-flop and 1-bit regester.

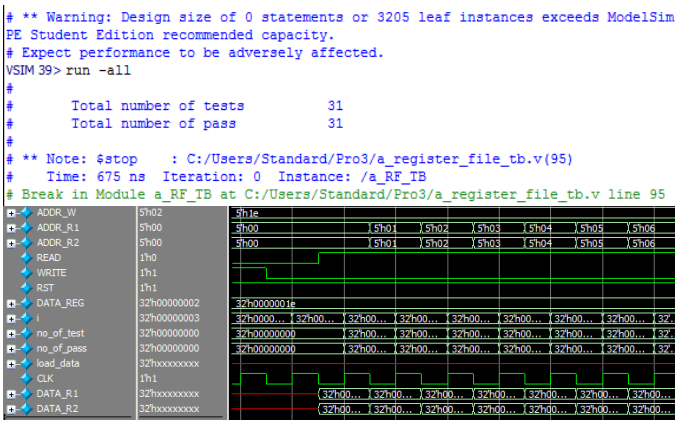
```
initial begin
        #5 d='b0; p='b0; r='b1;
        #5 c='b0;
        #5 c='b1;
        #5 golden(q1, 'b1, c, d, p, r);
        #5;
        #5 d='b1; p='b0; r='b1;
        #5 c='b0;
        #5 c='b1;
        #5 golden(q1, 'b1, c, d, p, r);
        #5;
        #5 d='b0; p='b1; r='b0;
        #5 c='b0;
        #5 c='b1;
        #5 golden(q1, 'b0, c, d, p, r);
        #5;
        #5 d='b1; p='b1; r='b0;
        #5 c='b0;
        #5 c='b1;
        #5 golden(q1, 'b0, c, d, p, r);
        #5;
end
```

Matching these results we could see that input changes would be as as expected



We implimented bahviour a 32-bit register for all other register componets. Instruction Register, Stack Pointer, and Program Counter, but the Register File fails to take data with on the datapath.

Testing the Regester File solo produced the intended results but owing to the large number of gates in play caused a minimum of 5 minute simulations and the failure could not be asertained.

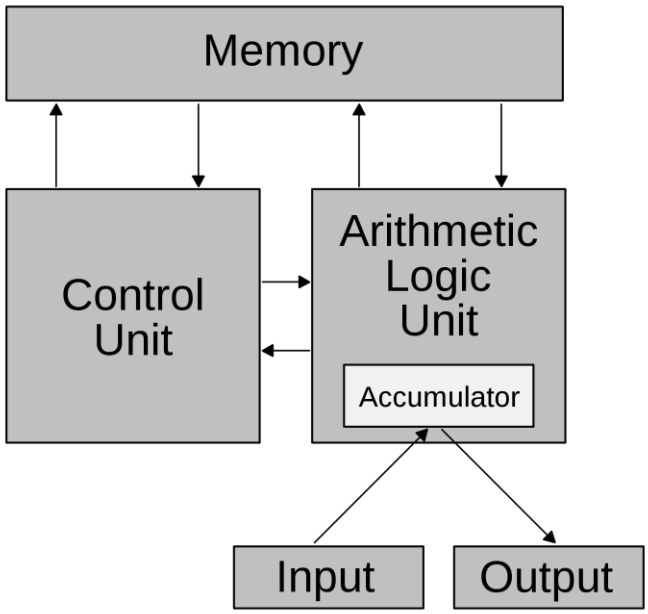


III. Design and Implementation of Processor

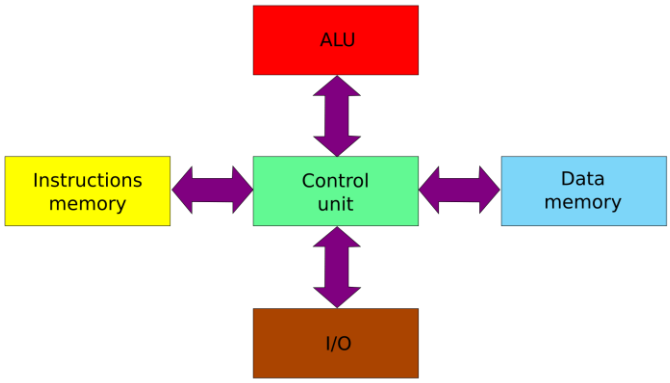
A. Design

A processor is a computer memory (we are excluding the minimal storage of the register file). A processor uses an instruction to perform operations on a word of data. Without data it is a collection of gates. With data it can do complex things very fast.

There are generally two models of processor design Princeton and Harvard. These relate to the layout of particular components to maximize toward particular purposes.



Princeton



Harvard

B. Implementation of Processor

Our layout processor was implemented as a hybrid of these designs. The module acts largely as a wrapper for the control unit and data path, instantiating each.

As most components will be individually tested and owning to the simplicity of this component a test bed was not built for it exclusively

1) Design Control Unit

The control unit is the workhorse of the Di Vinci system, and any computer. Continually cycles through system states that allow the other components to process instructions. It does this though sending control signals to the relevant device at the relevant time. The data pipeline is:

a) Fetch-

Instructions are retrieved from memory.

b) Decode-

Data is parsed for relevant information and data is prepared to be passed to correct components.

c) Execute-

Data is passed to ALU if relevant.

d) Memory-

Operations in main memory are performed.

e) Write Back-

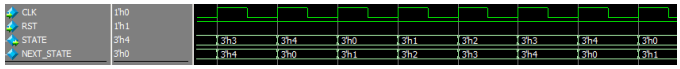
Results are written back to the register. Processor is prepared for the next instruction.

Our control unit is implemented behaviorally with a large case statement

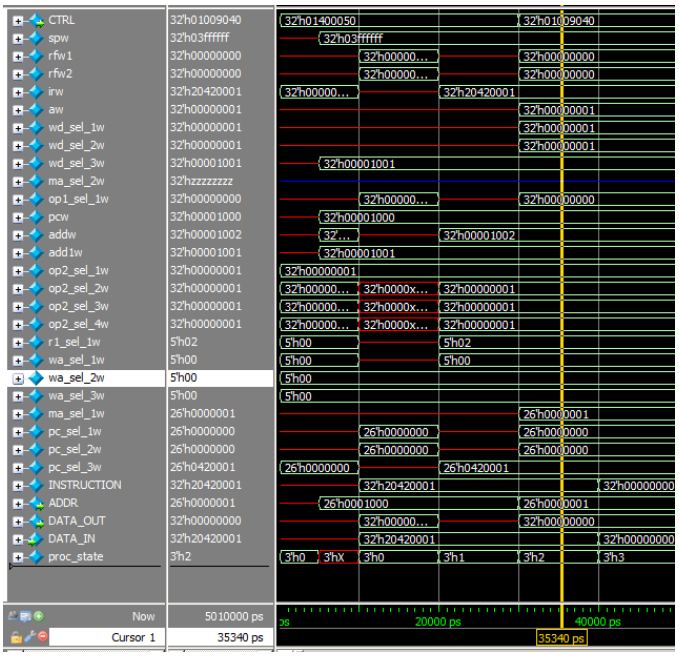
A second module is implemented with a "forecasting" variable to switch the machine state.

2) Control Unit Testing

This unit needs only be tested for state switching and control signals. The first will be tested extensively during Di Vinci simulation.



The second requires following an instruction through memory to be confident it passes evey component when it needs to and avoid having random data cause unexpected results.



This is only done through wave. At the following point, for example the instruction is an add at EXE. In this state we would expect interaction to occur with the ALU and we can follow the values of 0 and 1 from the instruction immediate and register file to the ALU and the correct result being returned.

We do not expect to see action with the SP, PC, Data Memory, or Register File Write and luckily we do not.

We can see the signal to the ALU is correct as are the expected inputs and outputs. We can see the results is not yet returned to the register file.

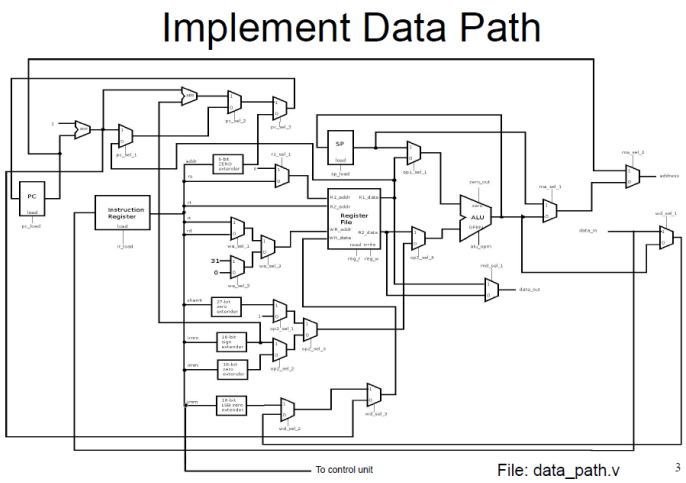
Our Control Unit testing was limited to just Fib and Rev Fib oriented control signals. While others were calculated they were not tested.

3) Design Data Path

The Data path is less the concrete component then the collection of all the extraneous wires that connect them. But because it chooses which and where the data runs it is extremely important

4) Implementation Data Path

Our data path is implemented at the gate level. With the exception of the Register File all the components there-in are as well. We us Multiplexers of 32x1 size to route data correctly.



Each control signal is influenced by one or six bits sent from the control unit.

5) *Testing Data Path*

This is the influencing factor for the Control Unit's test case. Both are solved under the same battery of tests.

IV. ALU

A. *Design ALU*

This unit is the jack of all trades. Taking any two inputs and a control it will dig into its box of tools are return a new result. Common ALU operations are:

1) *Add*
2) *Subtract*
3) *Multiply*
4) *Divide*
5) *Bit shift*
6) *Branch on Equal*
7) *And*
8) *NAND*

When implemented at the gate level all of these functions have components and each performs its action.
Testing of the ALU is performed both via Data Path testing and component testing.

B. *Implimentation ALU*

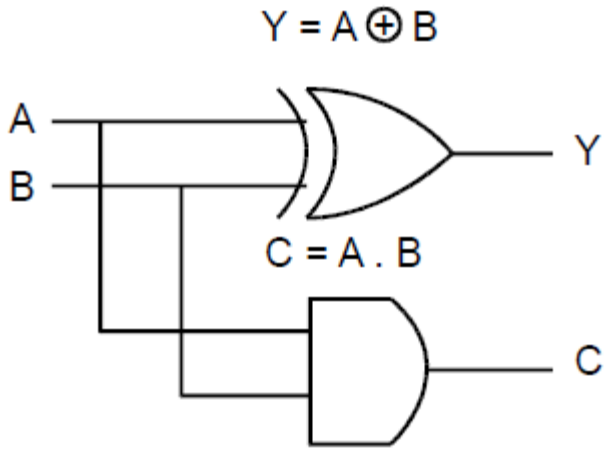
Our ALU is implemented at the gate level, as are each component. Our ALU can perform:

1) *Add/Subtract*
2) *Mulitplicatin*
3) *Left/Right Shift*
4) *And*
5) *Or*
6) *Nor*
7) *Less than*

Components such as Add/Subtract Left/Right shift have a control bit from the signal that differentiate the operation.

C. *Addition Implimentation*

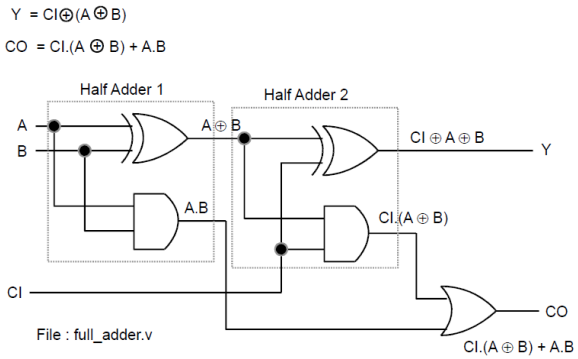
Addition of bits is the AND of the terms with an overflow if both are true. The most basic component of an Adder is the half adder:



a

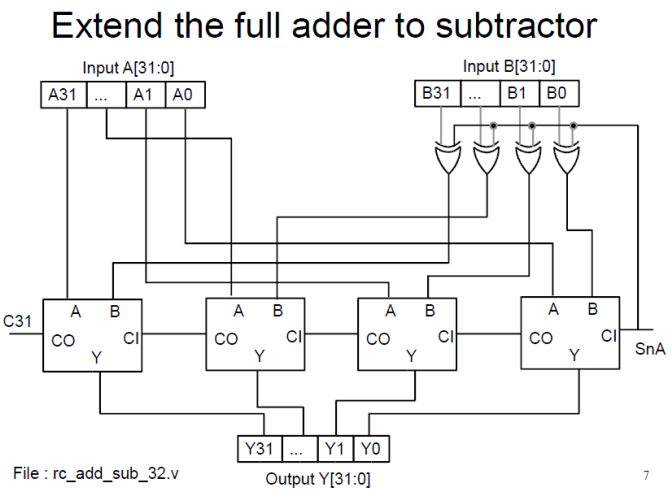
```
/** Basic half adder, does single bit addition only between two bits
* @param s is the result of the sum of the opperands
* @param c is the carry from the sum of the opperands if they exceed available digit
* @param o is one oppernad to be added
* @param p is the other opperand to be added
*/
module half_adder(s, c, o, p);
        input o, p;
        output s, c;
        xor i(s, o, p);
        and j(c, o, p);
endmodule
```

A Full Adder is closely associated



Owing to the simplicty of these systems it is not neccary to test until ilmimentation of the 32-bit ripple carry adder subtractor.
We use a generating for loop to impliment 32 full adders and other components

```
genvar i;
generate
        for (i=0; i<32; i=i+1) begin : rc_add_sub_32_loop
                xor xor_inst(xorProduct[i], subtractNotAdd, operand2[i]);
                if(i!=0 && i!=31) begin
                        full_adder fa(result[i], wireNext[i],
                                operand1[i], xorProduct[i], wireNext[i-1]);
                end else if(i==0) begin
                        full_adder fa(result[i], wireNext[i],
                                operand1[i], xorProduct[i], subtractNotAdd);
                end else if(i==31) begin
                        full_adder fa(result[i], carryOut,
                                operand1[i], xorProduct[i], wireNext[i-1]);
                end
        end
endgenerate
```



D. Addition Testing

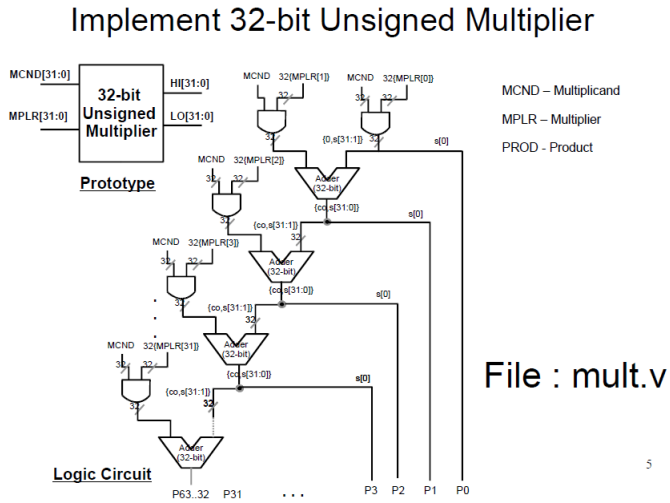
The test bench though is easy to prepare with the most complex compoent being calculation of base 10 values in binary.
Measuring through waveform is not particularly effective but via Transcript.

```
#
# Op1:          0 Op2:          0 sNa:0
#       = result:          0 carryOut:0
#
# Op1:          0 Op2:          0 sNa:1
#       = result:          0 carryOut:1
#
# Op1:          5 Op2:          2 sNa:0
#       = result:          7 carryOut:0
#
# Op1:          5 Op2:          2 sNa:1
#       = result:          3 carryOut:1
#
# Op1:          2 Op2:          5 sNa:0
#       = result:          7 carryOut:0
#
# Op1:          2 Op2:          5 sNa:1
#       = result:4294967293 carryOut:0
#
# Op1:        100 Op2:       1000 sNa:0
#       = result:       1100 carryOut:0
#
# Op1:        100 Op2:       1000 sNa:1
#       = result:4294966396 carryOut:0
#
# Op1:2147483647 Op2:2147483647 sNa:1
#       = result:          0 carryOut:1
#
# Op1:2147483647 Op2:          5 sNa:1
#       = result:2147483642 carryOut:1
```

We can see that overflow results are discarded and that negative numbers are not supported.

E. Multiplication Implimentaion

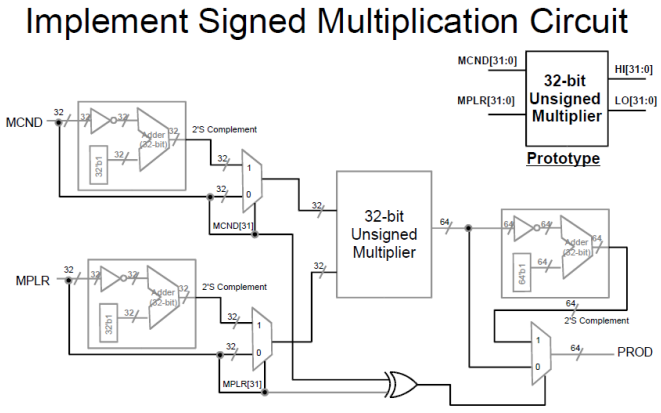
Multiplication was quite challenging to extensive use of hand drawn diagrams were used to simulate 3 bit multipliers in order to understand the generate statement.




```
and32 an(op2Wire[0], multiplicand, {32{multiplier[0]}});
buf b2(cOut[0], 1'b0);
buf b(resultLow[0], op2Wire[0][0]);

genvar j;
generate
        for (j=1; j<32; j=j+1) begin : loop_
                wire [31:0] op1Wire;
                and32 an(op1Wire, multiplicand, {32{multiplier[j]}});
                rc_add_sub_32 ad(op2Wire[j], cOut[j], op1Wire, {cOut[j-1],{op2Wire[j-1][31:1]}}, 1'b0);
                buf b(resultLow[j], op2Wire[j][0]);
        end
endgenerate
buf32 b4(resultHigh, {cOut[31],{op2Wire[31][31:1]}});
```

We see here an initial case involving a and gates. A repeating structure for 30 more associations. And the output of the last gate buffered to the carry out (not depicted on diagram).
By contrast creating a signed multiplication circuit was easier needing only the creation of a 2-compliment module and a new 64 bit adder (a extension to 64 of my generate for loop)



F. Mult Testing

Like the adder testing was very easy, involving only random or edge case operations and a calculator. For this test we employ a task to either print erroneous results or a PASSED statement.
Very easy to parse.

```
initial begin
        #5;
        #5 operand1='b0; operand2='b0;
        #5 golden(resultHigh, resultLow,'b0, 'b0, operand1, operand2);
        #5;
        #5 operand1='b1; operand2='b0;
        #5 golden(resultHigh, resultLow,'b0, 'b0, operand1, operand2);
        #5;
        #5 operand1='b0; operand2='b1;
        #5 golden(resultHigh, resultLow,'b0, 'b0, operand1, operand2);
        #5;
        #5 operand1='b1; operand2='b1;
        #5 golden(resultHigh, resultLow,'b0, 'b1, operand1, operand2);
        #5;
        #5 operand1='b11; operand2='b1;
        #5 golden(resultHigh, resultLow,'b0, 'b11, operand1, operand2);
        #5;
        #5 operand1='b11111111111111111111111111111111; operand2='b1;
        #5 golden(resultHigh, resultLow,'b0, 'b11111111111111111111111111111111, operand1,
        #5;
        #5 operand1='hffffffff; operand2='b1;
        #5 golden(resultHigh, resultLow,'hffffffff, 'hffffffff, operand1, operand2);
        #5;
        #5 operand1='b10; operand2='b10;
        #5 golden(resultHigh, resultLow,'b0, 'b100, operand1, operand2);
        #5;
        #5 operand1='b11; operand2='b11;
        #5 golden(resultHigh, resultLow,'b0, 'b1001, operand1, operand2);
        #5;
end
```

```
VSIM 16> run -all
# [PASSED]
# Run mult TB
# [PASSED]
# Run mult TB
# [PASSED]
# Run mult TB
# [PASSED]
# Run mult TB
# [PASSED]
# Run mult TB
# [PASSED]
# Run mult TB
# [PASSED]
# Run mult TB
# [PASSED]
# Run mult TB
# [PASSED]
# Run mult TB
```

Again the wave form generated is much less useful in checking opperations.

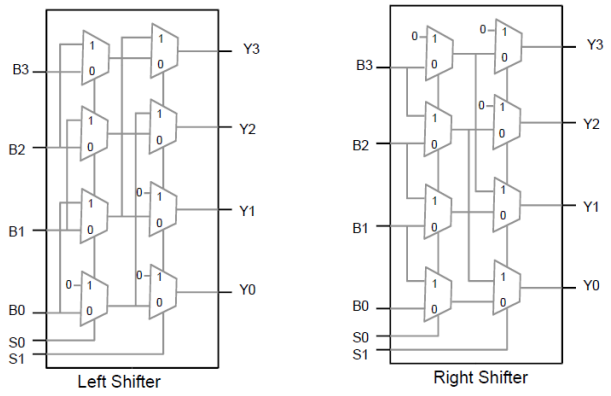
G. Shifting

Shifting is useful in many cases. Mathematically it is equivalent to putting a number to the power of its base. Our shifter was challenging to implement because of its extensive use of mux and it unique relationship of rows.

They can be built of many structures, most interestingly a flip flop but we used mux.

The following 4 bit shifter is not used but is useful in planning the lager 32 bit shifter.

Extend 4-bit Barrel Shifter to 32-bit

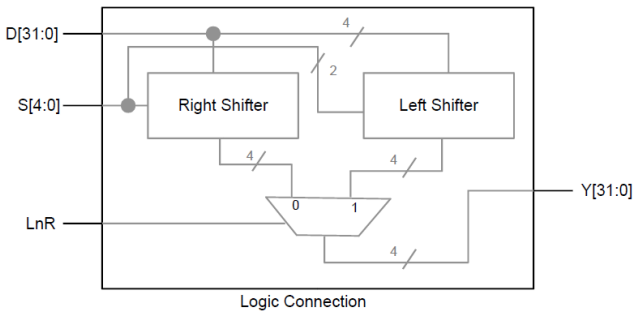


A 32 shifter has $32=2^n$ where n represent the number of control bits and rows. Our shifter therefor had 5 control bits. Across 32 rows that is 160 mux. Perhaps 200 more lines of wires and supporting code.
An effective for loop was ideal. I used 3. One covering the input muxes, one covering the output muxes. One covering the interior muxes.

```
genvar i, m, j, k;
generate
        for (i=0; i<=31; i=i+1)
        begin : input_mux_gen_loop_
                if(i>0) begin
                        mux m1(.result(mOut[0][i]), .operand1(operand[i]), .operand2(operand[i-1]), .control(shift[0]));
                end else begin
                        mux m2(.result(mOut[0][i]), .operand1(operand[i]), .operand2(1'b0), .control(shift[0]));
                end
        end
        for (m=0; m<3; m=m+1)
        begin : inner_col_mux_gen_loop_
                for (k=0; k<=31; k=k+1)
                begin : inner_mux_gen_loop_
                        if(k>=2**(m+1)) begin
                                mux m3(.result(mOut[m+1][k]), .operand1(mOut[m][k]), .operand2(mOut[m][k-2**(m+1)]), .control(shift[m+1]));
                        end else begin
                                mux m4(.result(mOut[m+1][k]), .operand1(mOut[m][k]), .operand2(1'b0), .control(shift[m+1]));
                        end
                end
        end
        for (j=0; j<=31; j=j+1)
        begin : result_mux_gen_loop_
                if(j>16) begin
                        mux m5(.result(result[j]), .operand1(mOut[3][j]), .operand2(mOut[3][j-16]), .control(shift[4]));
                end else begin
                        mux m6(.result(result[j]), .operand1(mOut[3][j]), .operand2(1'b0), .control(shift[4]));
                end
        end
endgenerate
```

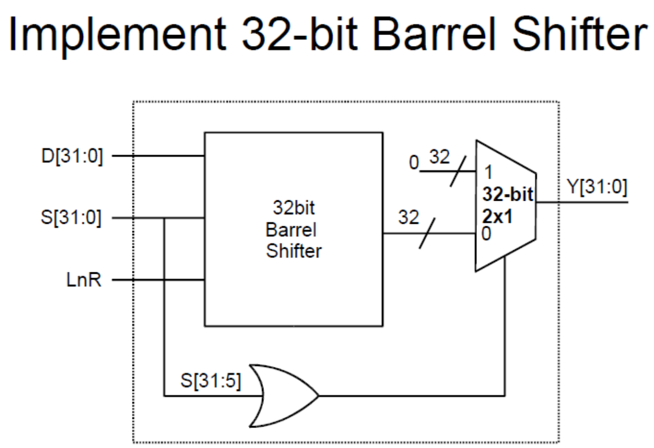
Lastly it is important to get the zero depth correct, since shifting adds empty digits. These were at a depth of 2^n where n was the depth number -1.
We then wrap it in a simple Barrel shifter with a control for left and right operation
Wrap

Implement 32-bit Barrel Shifter



File: barrel_shifter.v

and another control to allow larger shifts to fall away



H. Testing Shifter

Owing to the complexity of the loops testing this module was extensive.

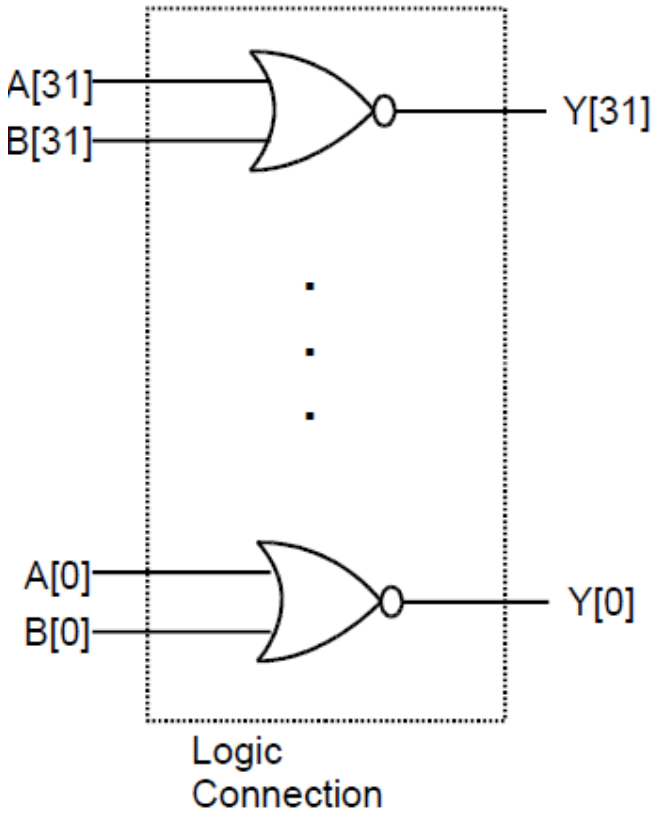
```
#5 operand='b1010; shift='b11; leftNotRight='b0;
#5 golden(result,'b1, operand, shift, leftNotRight);
#5 operand='b1011; shift='b11; leftNotRight='b0;
#5 golden(result,'b1, operand, shift, leftNotRight);
#5 operand='b1100; shift='b11; leftNotRight='b0;
#5 golden(result,'b1, operand, shift, leftNotRight);
#5 operand='b1101; shift='b11; leftNotRight='b0;
#5 golden(result,'b1, operand, shift, leftNotRight);
#5 operand='b1110; shift='b11; leftNotRight='b0;
#5 golden(result,'b1, operand, shift, leftNotRight);
#5 operand='b1111; shift='b11; leftNotRight='b0;
#5 golden(result,'b1, operand, shift, leftNotRight);
#5 operand='b1; shift='b1; leftNotRight='b1;
#5 golden(result,'b10, operand, shift, leftNotRight);
#5 operand='b10000000000000000000000000000000; shift='b1; leftNotRight='b1;
#5 golden(result, 'b100000000000000000000000000000000, operand, shift, leftNotRight);
#5 operand='b10; shift='b10; leftNotRight='b1;
#5 golden(result,'b1000, operand, shift, leftNotRight);
#5 operand='b100; shift='b10; leftNotRight='b1;
#5 golden(result,'b10000, operand, shift, leftNotRight);
#5 operand='b100; shift='b11; leftNotRight='b1;
#5 golden(result,'b100000, operand, shift, leftNotRight);
#5 operand='b11000; shift='b11; leftNotRight='b1;
#5 golden(result,'b11000000, operand, shift, leftNotRight);
#5 operand='b10000000000000000000000000000000; shift='b1; leftNotRight='b1;
#5 golden(result,'b100000000000000000000000000000000, operand, shift, leftNotRight);
#5 operand='b1; shift='b101; leftNotRight='b1;
#5 golden(result,'b100000, operand, shift, leftNotRight);
#5 operand='b10000000000000000000000000000000; shift='b1; leftNotRight='b1;
#5 golden(result,'b100000000000000000000000000000000, operand, shift, leftNotRight);
#5 operand='b10; shift='b10000; leftNotRight='b1;
#5 golden(result,'b10000000000000000, operand, shift, leftNotRight);
#5 operand='b100; shift='b10; leftNotRight='b1;
#5 golden(result,'b10000, operand, shift, leftNotRight);
#5 operand='b100; shift='b11; leftNotRight='b1;
#5 golden(result,'b100000, operand, shift, leftNotRight);
#5 operand='b1111111111111000; shift='b10000; leftNotRight='b1;
```

Result calculation was extremely simple though.

```
# Run barrel shift TB
# [PASSED]
# Run barrel shift TB
# [PASSED]
# Run barrel shift TB
# [PASSED]
# Run barrel shift TB
# [PASSED]
# Run barrel shift TB
# [PASSED]
# Run barrel shift TB
# [PASSED]
# Run barrel shift TB
# [PASSED]
# Run barrel shift TB
# [PASSED]
# Run barrel shift TB
# [PASSED]
# Run barrel shift TB
# [PASSED]
# Run barrel shift TB
# [PASSED]
# Run barrel shift TB
# [PASSED]
# Run barrel shift TB
# [PASSED]
# Run barrel shift TB
# [PASSED]
# Run barrel shift TB
# [PASSED]
# Run barrel shift TB
# [PASSED]
# Run barrel shift TB
# [PASSED]
# Run barrel shift TB
# [PASSED]
# Run barrel shift TB
# [PASSED]
# Run barrel shift TB
# [PASSED]
# Run barrel shift TB
```

I. Other Operations

Or, nor, and and where all trivial repeating structure.



J. Testing ALU

With a suite of operations testing the ALU is a matter of testing input and output under each signal. Expanding on an old test bench:

```
# [TEST] 15 + 3 = 18 , got 18 ... [PASSED]
# [TEST] 15 - 5 = 10 , got 10 ... [PASSED]
# [TEST] 15 + 5 = 20 , got 20 ... [PASSED]
# [TEST] 15 * 5 = 75 , got 75 ... [PASSED]
# [TEST] 15 << 5 = 480 , got 480 ... [PASSED]
# [TEST] 15 >> 5 = 0 , got 0 ... [PASSED]
# [TEST] 15 & 5 = 5 , got 5 ... [PASSED]
# [TEST] 15 | 5 = 15 , got 15 ... [PASSED]
# [TEST] 15 ~| 5 = 4294967280 , got 4294967280 ... [PASSED]
```

```
#5  test_and_count(total_test, pass_test,
    test_golden(op1_reg,op2_reg,oprn_reg,r_net));
#5  op1_reg=15;
    op2_reg=5;
    oprn_reg=`ALU_OPRN_WIDTH'h03;
#5  test_and_count(total_test, pass_test,
    test_golden(op1_reg,op2_reg,oprn_reg,r_net));
#5  op1_reg=15;
    op2_reg=5;
    oprn_reg=`ALU_OPRN_WIDTH'h04;
#5  test_and_count(total_test, pass_test,
    test_golden(op1_reg,op2_reg,oprn_reg,r_net));
#5  op1_reg=15;
    op2_reg=5;
    oprn_reg=`ALU_OPRN_WIDTH'h05;
#5  test_and_count(total_test, pass_test,
    test_golden(op1_reg,op2_reg,oprn_reg,r_net));
#5  op1_reg=15;
    op2_reg=5;
    oprn_reg=`ALU_OPRN_WIDTH'h06;
#5  test_and_count(total_test, pass_test,
    test_golden(op1_reg,op2_reg,oprn_reg,r_net));
#5  op1_reg=15;
    op2_reg=5;
    oprn_reg=`ALU_OPRN_WIDTH'h07;
#5  test_and_count(total_test, pass_test,
    test_golden(op1_reg,op2_reg,oprn_reg,r_net));
#5  op1_reg=15;
    op2_reg=5;
    oprn_reg=`ALU_OPRN_WIDTH'h08;
#5  test_and_count(total_test, pass_test,
    test_golden(op1_reg,op2_reg,oprn_reg,r_net));
```

No ALU operation is as effective to test bench in terminal output than in wave form.

V. Conclusion.

This was an incredibly intense programing assignment, but also incredibly rewarding. Time was too short and my workload too heavy for me to truly and extensively manipulate this System as I would have wished.

Our known test bed using the Fib and RevFib operations were unsatisfyingly short and far from extensive enough to test all the operations in even our tiny instruction set. Because of time constraints other signals were calculated but not tested.

The use of Logisim as well was very enticing and a learned many new applications, including custom circuits and automatic clocks as I attempted to model particularly vexing modules.

I would have enjoyed seeing a larger system modeled in GUI but fear the XML that drives in Logism would fail over that number of connections.

The problem of registers is particularly vexing to me, but owning to lack of time I have not been able to solve it. In small manual testbeds written by our professor they pass swimmingly. All 32-bit registers also work in other applications. But the 32x 32-bit RF does not run and the 2 minute delay on the full system test, even for a single instruction made diagnostics thus far ineffective.