Introduction and Implementation of Computer System in Behavioral and Gate Level Model and Test Results

Claire Lin
Computer Science Dept., San Jose State University
Shao-yu.lin @sjsu.edu

I. INTRODUCTION

The goal is to simulate the mix of behavioral and gate level model of a bare minimum computer system, DaVinci v1.0, and its instruction set, cs147DV.

II. REQUIREMENT FOR SYSTEM

A. Register File

The Register File is made up of 32 registers (with number 0 - 31) each with a size of 32-bit. The register file resets on a -ve edge of RST signal, and other operations are switched and executed on a positive edge of the clock.

The Register File has two read ports, and each read port has an address port and data port, RF_ADDR_R1, RF_DATA_R1, RF_ADDR_R2, and RF_DATA_R2. The register file has one write port that has an address port and data port, RF__ADDR_W, RF_DATA_W. The read and write ports are separated so there is no need for bi-directional mechanism as implemented for memory. Read operation is executed when signal READ, WRITE = 1,0. Write operation is executed when signal READ, WRITE = 01. When READ, WRITE = 00 or 11, the Register File does nothing and holds the previously read or write data.

B. Memory

The Memory size is 256MB and double-word addressable (total 64M address). Reset is done at –ve edge of the RST signal, and other operations and switched and executed on +ve edge of the Clock. The Memory uses one port to write or write. As the result, there is a need to implement the bi-directional read-write mechanism, which is discussed in detailed in the Design and Implementation of Memory section. Read operation is executed when signal READ, WRITE = 1,0. Write operation is executed when signal READ, WRITE = 01.

C. Arithmetic Logic Unit (ALU)

The Control Unit commands the ALU the operations to be executed. All ALU operations take two operands, one operation code (opcode), and produce one output. The operands are 32-bit, the opcode is 6-bit, and the output is 32-bit. An additional ZERO output is set to 0 when an operation result output is 1, 0 otherwise. The operations the corresponding Operation Code (opcode) are listed below.

- Add (opcode: 000001)
- Sub (opcode: 000002)
- Mul (opcode: 000003)
- Shift Right (opcode: 000004)
- Shift Left (opcode: 000005)
- Bitwise AND (opcode: 000006)
- Bitwise OR (opcode: 000007)
- Bitwise NOR (opcode: 000008)
- Set Less Than (slt) (opcode: 000009)

D. Control Unit

The Control Unit implements a five-stage state machine. The stages are ‘PROC_FETCH, ‘PROC_DECODE, ‘PRCO_EXE, ‘PROC_MEM, ‘PROC_WP. The machine stage switches at +ve of the clock.

The Control Unit also implements the execution of CS147DV INSTRUCTION SET and synchronize all operations between Memory, Clock, Register File, and ALU.

E. Clock

The Clock generates a 1-bit signal that oscillated between 0 and 1, which serves as a measure of time for the entire DaVinci v1.0 computer system.

F. CS147DV Instruction Set

The CS147DV Instruction Set consists of 3 types of instructions, J-type, I-type, and R-type. All instruction widths are 32-bit.

- R-type instructions consist of an opcode field (6-bit), three register address fields (each 6-bit), a shamt field (5-bit shift amount), and a funct field (6-bit function code).
- I-type instructions consist of an opcode field (6-bit), two register address fields (each 6-bit), and an immediate field (16-bit).
- J-type instructions consist of an opcode field (6-bit) and an address field (26 bit)

III. DESIGN AND IMPLEMENTATION OF PROCCESSOR

The Processor consists of Register File, ALU, and Control Unit.

A. Register File Design and Implementation

The Register File storage is a 32x32 two-dimensional array, with each 32 registers being 32 bits.

- *Read-Write Mechanism*

When READ, WRITE = 1,0 (read operation), the two separate read data ports, DATA_R1, DATA_R2, each read from their respective register addressed. DATA_R1 is set to register file content at address ADDR_R1, and register file at ADDR_R2 is set to content DATA_R2. When READ, WRITE = 0,1(write operation), the register with address ADDR_W is set to DATA_W, the write only data port. The Register File does not handle read, write = 00 or 11. For such configuration, Register File hold the previously read or written data. Code for read and write operation is shown in Fig.1.

```
if((READ===1'b1) && (WRITE===1'b0))
begin
    DATA_R1 = reg_32x32[ADDR_R1];
    DATA_R2 = reg_32x32[ADDR_R2];
end
else if ((READ===1'b0) && (WRITE===1'b1))
    reg_32x32[ADDR_W] = DATA_W;
```

Fig. 1

- *Reset Operation*

The Register File resets at the –ve edge of RST port signal and esets all contents of Register File into 32-bit 0s. The code for reset operation is shown in Fig. 2.

```
if (RST === 1'b0)
begin
    for (i=0; i<=`REG_INDEX_LIMIT; i = i+1)
        reg_32x32[i] = {`DATA_WIDTH{1'b0} };
end
```

Fig. 2

- *Gate Level Design and Implementation*

The implementation of Register File at gate level requires implementing each smaller components of Register File at gate level. The necessary components are listed below.

a. 32-bit Register
   The 32-bit register is made up of smaller gate level components, as listed below.
   1. SR Latch
   2. D Latch
   3. Flip Flop that is triggered by –ve of Clock
   4. 1-bit Register

b. 32-bit 32x1 mux
   The 32-bit 32x1 reuses smaller mux components implemented during ALU gate-level implementations.

c. 5x32 line decoder
   The 5x32 line decoder is made up of smaller gate level components, as listed below.
   1. 4x16 line decoder
   2. 3x8 line decoder
   3. 2x4 line decoder

The gate-level Register File instantiates 32 32-bit registers. The register file also instantiates a 5x32 line decoder to select which register to conduct write operation. Each of the 32 signal outputs from the decoder is, then, AND with the write signal to determine ultimately if there is a write operation and which register to conduct write operations. Such signals are connected to each of the 32-bit registers, as shown in Fig.3.

```
DECODER_5x32 decode_inst1 (decode_out, ADDR_W);
genvar i;
generate
    for(i =0; i<32; i = i+1)
    begin : reg32_gen_loop
        and and_inst (and_out[i], decode_out[i], WRITE);
    end
endgenerate

genvar j;
generate
    for(j =0; j<32; j = j+1)
    begin : re32_gen_loop
        REG32 reg_inst (qout[j], DATA_W, and_out[j] , CLK, RST);
    end
endgenerate
```

Fig.3

For read operation, each of the 32 32-bit register is connected 32x1 32-bit multiplexer to select which registers to read from. Because there are two read data ports, 2 32x1 32-bit multiplexer are instantiated. To decide if there is a read operation, the 2 32x1 32-bit multiplexer are each connected with a 2x1 32-bit multiplexer with a control signal as the read signal, as shown in the figure.4.

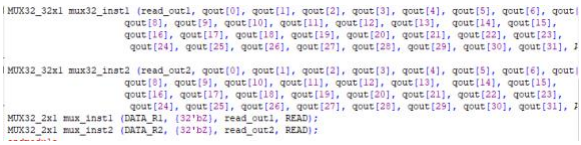

Fig 4.

B. ALU Design and Implementation

The implementation of ALU at gate level requires implementing each component of an arithmetic operations supported by ALU at gate level. The necessary components are listed below.

a. Add (opcode: 000001), Sub (opcode: 000002)

The following components are implemented at gate level to support Add and Sub.

1. Half Adder
2. Full Adder
3. 32-bit Binary Ripple Carry Adder
4. 32-bit Binary Ripple Carry Adder/Subtractor

b. Mul (opcode: 000003)

The following components are implemented at gate level to support Mul.

1. 32-bit Binary Ripple Carry Adder
2. 32-bit And gate
3. 32-bit unsigned Multiplier
4. 32-bit 2's compliment
5. 64-bit 2's compliment
6. 32-bit 2x1 mux

c. Shift Right (opcode: 000004), Shift Left (opcode: 000005)

The following components are implemented at gate level to support Shift Right.

1. 32-bit left shifter
2. 32-bit right shifter
3. 32-bit barrel shifter

d. Bitwise AND (opcode: 000006)

The following components are implemented at gate level to support Bitwise AND.

1. 32-bit AND gate

e. Bitwise OR (opcode: 000007)

The following components are implemented at gate level to support Bitwise OR.

1. 32-bit OR gate

f. Bitwise NOR (opcode: 000008)

The following components are implemented at gate level to support Bitwise NOR.

1. 32-bit NOR gate

g. Set Less Than (slt) (opcode: 000009)

Set Less Than uses the gate level 32-bit Binary Ripple Carry Adder/Subtractor. The result is indicated by taking the 32th bit result of 32-bit Binary Ripple Carry Adder/Subtractor

h. ZERO output Flag

If the output of any of the nine operations is 0, ZERO is set to 1. ZERO is set to 0, otherwise. If the operation code does not match any of the operation code of the nine operations listed above. The ALU returns 32 bit of HiZ.

The gate-level ALU instantiates necessary gate-level components for each of the nine operations, and uses the 32-bit 16x1 multiplexer to select between operations to performed based on 6-bit opcode, as shown in Fig.5.

```
RC_ADD_SUB_32 addsub1 (addsub_out, coNoNeed, OP1, OP2, SnA);
MULT32 mul_inst1 (mulHiNoNeed, mul_out, OP1, OP2);
SHIFT32 sh_inst1 (shift_out,OP1,OP2, LnR);
AND32_2x1 and_inst1(and_out,OP1,OP2);
OR32_2x1 or_inst1 (or_out,OP1,OP2);
NOR32_2x1 nor_inst1 (nor_out,OP1,OP2);

MUX32_16x1 mu_inst1 (OUT, NoResult,addsub_out ,addsub_out, mul_out, shift_out, shift_out, and_out
          {{31'b0},addsub_out[31]},NoResult , NoResult, NoResult, NoResult, NoResult,
```

Fig.5.

The ZERO flag is implemented by calling 1-bit OR on every bit of the ALU output result and inverting the final result. Such implementation is equivalent to calling NOR on every bit of ALU output result, as shown in Fig.6.

```
or inst_or11 (or_result[0], OUT[0], OUT[1]);
genvar i;
generate
    for(i =2; i<=31; i = i+1)
    begin : or30_gen_loop
        or inst_or12 (or_result[i-1], OUT[i], or_result[i-2]);
    end
endgenerate

not not_inst1 (ZERO,or_result[30]);
```

Fig.6.

IV. TEST STRETEGY AND TEST IMPLEMENTATION

A. Test Of ALU

The gate-level implementation of ALU is made up of smaller gate-level components, as discussed earlier in ALU Design and Implementation. Each smaller gate-level component has its own test, and the results are listed below.

a. Half Adder

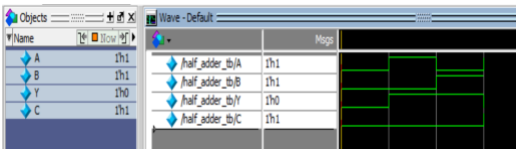

Fig. Test waveform of half adder. All test cases pass.

b. Full Adder

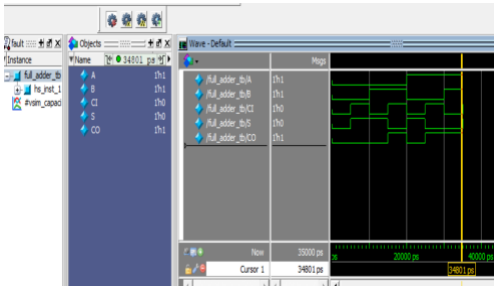

Fig. Test waveform of full adder. All test cases pass.

c. 32-bit and 64-bit Binary Ripple Carry Adder/ Substractor

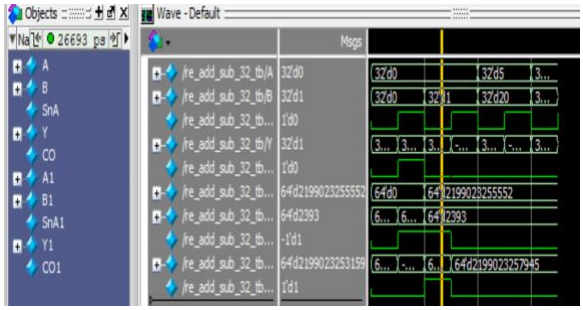

Fig. Test waveform of 32-bit and 64-bit ripple carry adder/substractor. All test cases pass.

d. 32-bit invertor

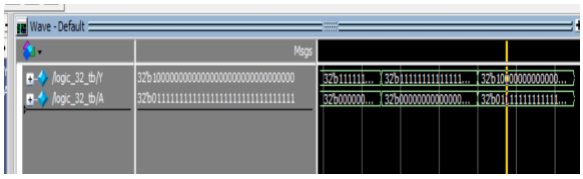

Fig. Test waveform of 32-bit invertor. All test cases pass.

e. 1-bit 2x1 MUX

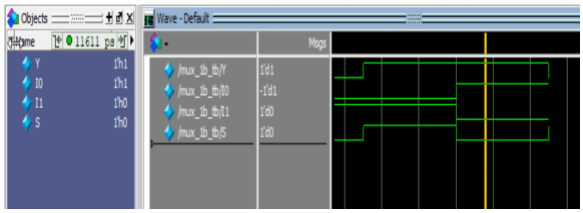

Fig. Test waveform of 1-bit 2x1 MUX. All test cases pass.

f. 32-bit 2x1 MUX

Fig. Test waveform of 32-bit 2x1 MUX. All test cases pass.

g. 32-bit 4x1 MUX

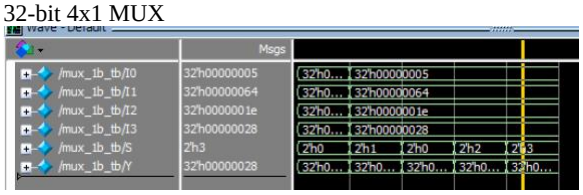

Fig. Test waveform of 32-bit 4x1 MUX. All test cases pass.

h. 32-bit 8x1 MUX

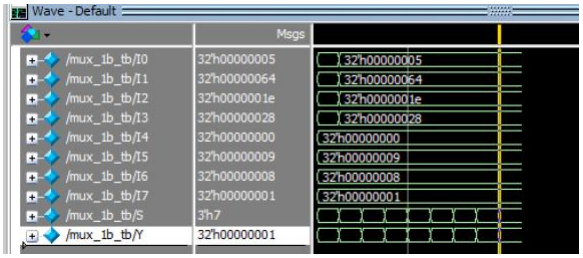

Fig. Test waveform of 32-bit 8x1 MUX. All test cases pass.

i. 32-bit 16x1 MUX

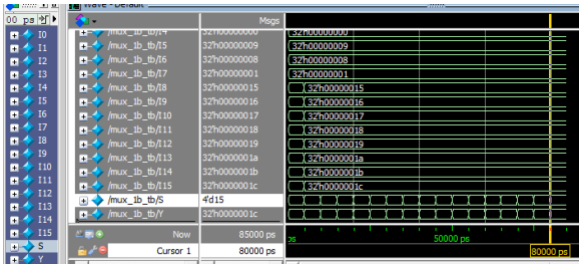

Fig. Test waveform of 32-bit 16x1 MUX. All test cases pass.

j. 32-bit and 64-bit 2's complement

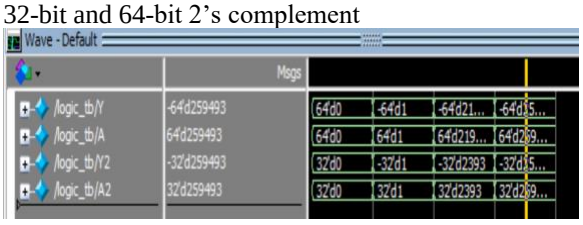

Fig. Test waveform of 32-bit and 64-bit 2's complement. All test cases pass.

k. 32-bit unsigned multiplier and signed multiplier

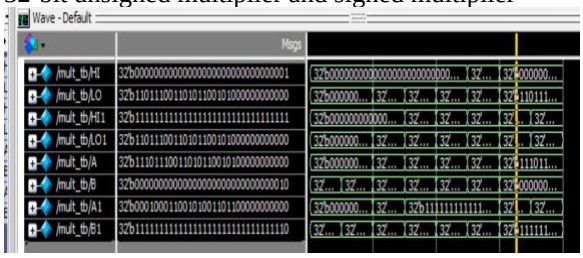

Fig. Test waveform of 32-bit unsigned and signed multiplier. All test cases pass.

l. 32-bit left shifter, right shifter, barrel shifter with 5-bit shift input

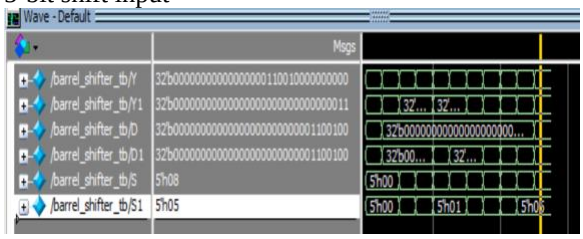

Fig. Test waveform of 32-bit left, right, and barrel shifter with 5-bit shift input. All test cases pass.

m. 32-bit barrel shifter with 32-bit shift input

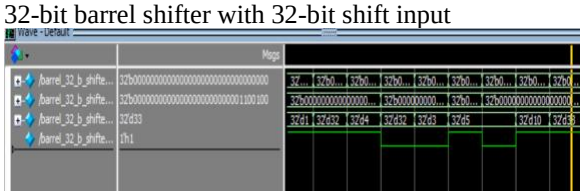

Fig. Test waveform of 32-bit barrel shifter with 32-bit shift input. All test cases pass.

n. Testing the ALU as a whole

The test bench file for testing ALU as a whole is retrieved from the Project 1 test bench file. The result generated from ALU implementation is compared with correct results of predefined test cases. If the two values equal to each other, the test case passes. The test case fails, otherwise. The correctness of ZERO flag implementation can be observed in waveform.

```
# Expect performance to be adversely affected.
VSIM 102> run -all
# [TEST] 15 + 3 = 18 , got 18 ... [PASSED]
# [TEST] 15 - 5 = 10 , got 10 ... [PASSED]
# [TEST] 15 * 5 = 75 , got 75 ... [PASSED]
# [TEST] 15 * 0 = 0 , got 0 ... [PASSED]
# [TEST] 15 >> 5 = 0 , got 0 ... [PASSED]
# [TEST] 15 << 5 = 480 , got 480 ... [PASSED]
# [TEST] 15 & 5 = 5 , got 5 ... [PASSED]
# [TEST] 15 | 5 = 15 , got 15 ... [PASSED]
# [TEST] 15 ~| 5 = 4294967280 , got 4294967280 ... [PASSED]
# [TEST] 15 < 5 = 0 , got 0 ... [PASSED]
# [TEST] 5 < 15 = 1 , got 1 ... [PASSED]
#
#       Total number of tests         11
#       Total number of pass          11
#
# ** Note: $stop    : C:/Users/Claire/Desktop/cs147/alb_tb.v(111)
#    Time: 115 ns  Iteration: 0  Instance: /alu_tb
```

Fig. 12 ALU test case transcript screenshot from ModelSim

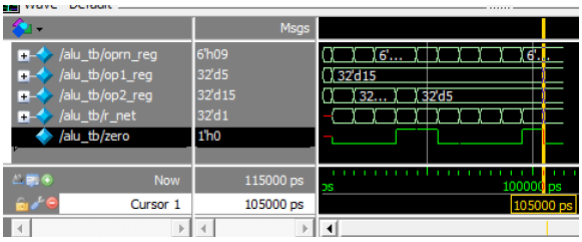

Fig. 13 ALU test case waveform screenshot from ModelSim. The waves verify the test case results. ZERO flag behaves correctly.

B. Test of Register File

The gate-level implementation of Register File is made up of smaller gate-level components, as discussed earlier in Register File Design and Implementation. Each smaller gate-level component has its own test, and the results are listed below.

a. 2x4 line decoder

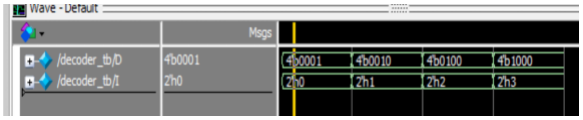

Fig. Test waveform of 2x4 line decoder. All test cases pass.

b. 3x8 decoder

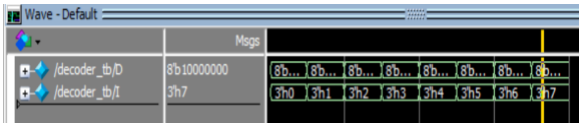

Fig. Test waveform of 3x8 line decoder. All test cases pass.

c. 4x16 decoder

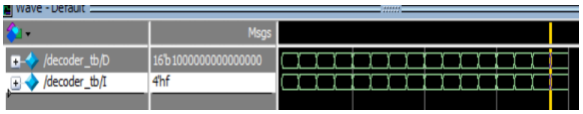

Fig. Test waveform of4x16 line decoder. All test cases pass.

d. 5x32 decoder

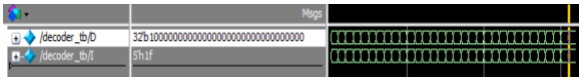

Fig. Test waveform of 5x32 line decoder. All test cases pass.

e. 1-bit FlipFlop

The 1-bit FlipFlop is made up of 1 D-Latch and 1 SR-Latch. Test cases are written and conducted at FlipFlop level.

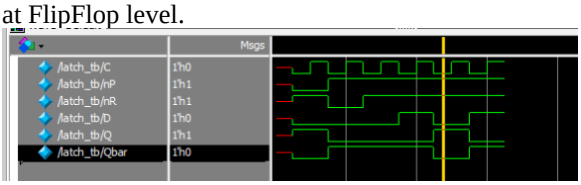

Fig. Test waveform of 1-bit FlipFlop. All test cases pass.

f. 32-bit 32x1 mux

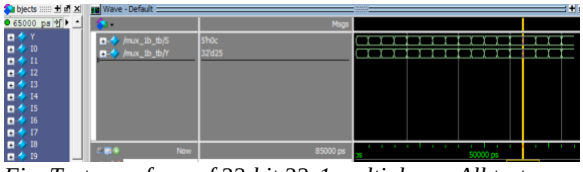

Fig. Test waveform of 32-bit 32x1 multiplexer. All test cases pass.

g. Testing 1-bit Register, 32-bit Register and 32x32 Register File as a whole

The test bench file for testing 32x32 Register File as a whole is provided in source code folder, in the file named RF_tb.v. The result generated from the Register File implementation is compared with correct results of predefined test cases. The test case fails, otherwise. The 1-bit and 32-bit register did not have their separated test bench, but was tested in Register File as a whole.

Register File testing involved writing values into registers and reading from registers later. If the value that is write to the register is not the value read from the register, the test fails. If the two read ports returns different values at the same register address, the test also fails. The test passes, otherwise.

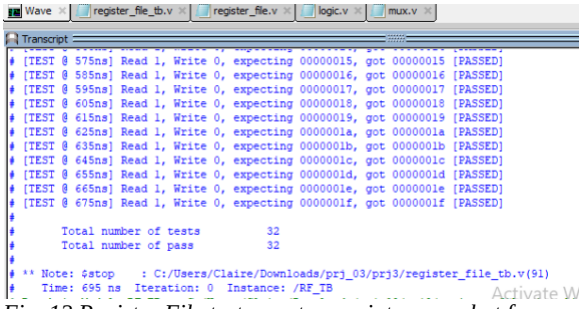

Fig. 12 Register File test case transcript screenshot from ModelSim

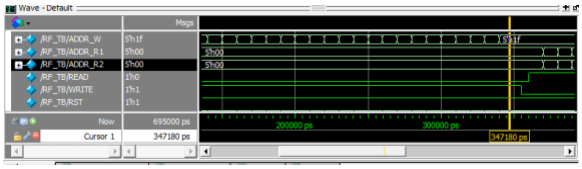

Fig.13 Register File test case waveform and transcript screenshots from ModelSim. All test cases pass.

V. CONCLUSION

The process of implementing the ALU and Register File of DaVinci v.1.0 computer system at gate level has dramatically improved my understanding of how and what circuit components are used in building a computer system.

The DaVinci v.1.0 compuster system also offers an in-dept. hands-on experience on writing code in Verilog with ModelSim. The building of many smaller gate-level components requires intensive testing, which has improved my ability and knowledge on debugging with simulation waveforms.