Behavioral Model of Davinci v1.0

SHUANG PAN

San Jose State University

shuang.pan@sjsu.edu

***Abstract*—The project is to use ModelSim Student Edition to implement to write behavioral model of a bare minimum computer system Davinci v1.0, which it supports the instruction set cs 147DV that is taught by Professor. Patra in the class.. The followings are the objective of this project.**

1) To implement a behavioral model of a computer system with 32-bits processor and 256Mb memory.
    1. Implement the memory model.
    2. Implement the ALU.
    3. Implement the register file.
    4. Implement the control unit which can support the cs147sec05 instruction set.
    5. Combine ALU, register file, and the control unit as the processor.
    6. Combine the processor and the memory system as Davinci v1.0.
2) To implement test for implemented system Davinci v1.0.
    1. Create the testbench for the memory system.
    2. Create the testbench for the ALU.
    3. Create the testbench for the register file.
    4. Create the testbench for Davinci v1.0.

1. Implement the memory model

The implementation of the memory system is given by the professor, and the following is the interface of the memory model (figure 1.1).

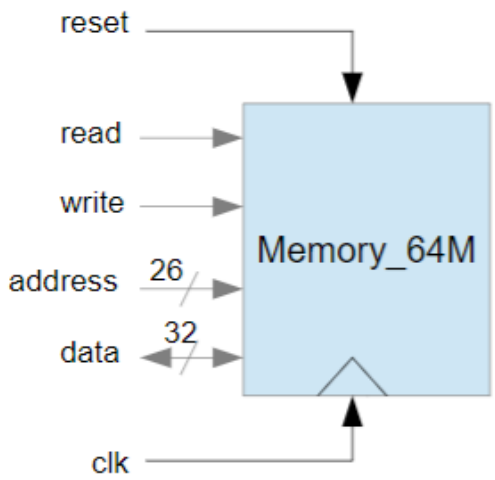

Figure 1.1 Interface of memory model

The memory model has 64MB with 26 address port. Each spot in the memory has 32 bits. It has four signals including READ, WRITE, CLK, and RST. Figure 1.2 shows how it is designed in physical model. At every negative of RST and positive edge of CLK, the memory will be initialized with 32bits zero if the RST is 0. Otherwise, if READ signal is 1

and WRITE signal is 0, the data will be read out from the certain address. If READ signal is 0 and WRITE signal is 1, the data will be written at the certain address. The data will be at high Z for all other cases. The following is the interface of the memory interface.

```
module MEMORY_64MB(DATA, READ, WRITE, ADDR, CLK, RST);
// Parameter for the memory initialization file name
parameter mem_init_file = "mem_content_01.dat";
// input ports
input READ, WRITE, CLK, RST;
input [`ADDRESS_INDEX_LIMIT:0] ADDR;
// inout ports
inout [`DATA_INDEX_LIMIT:0] DATA;

// memory bank
reg [`DATA_INDEX_LIMIT:0] sram_32x64m [0:`MEM_INDEX_LIMIT]; // memory storage
integer i; // index for reset operation

reg [`DATA_INDEX_LIMIT:0] data_ret; // return data register

assign DATA = ((READ===1'b1)&&(WRITE===1'b0))?data_ret:{`DATA_WIDTH{1'bz} };

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

Figure 1.2 Memory model implementation

2. Implement the ALU

The ALU module is similar with the one in the project 1. It only has a tiny difference. The following is the interface of the ALU. In the project 2, we add one bit ZERO flag for some operations in the instruction set.

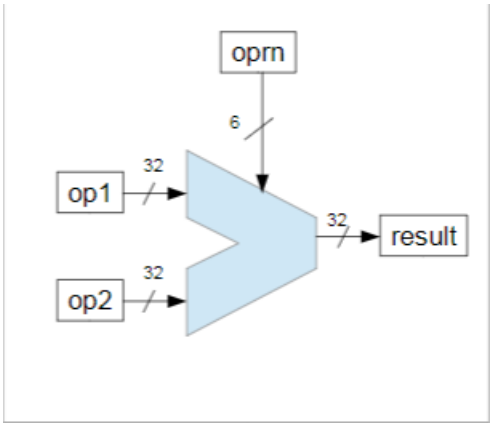

Figure 2.1 Interface of ALU

It has two 32-bits input, one 6-bits operation code, and one 32-btis result. The implementation of ALU is the following figure 2.2.

```
module ALU(OUT, ZERO, OP1, OP2, OPRN);
// input list
input [`DATA_INDEX_LIMIT:0] OP1; // operand 1
input [`DATA_INDEX_LIMIT:0] OP2; // operand 2
input [`ALU_OPRN_INDEX_LIMIT:0] OPRN; // operation code

// output list
output [`DATA_INDEX_LIMIT:0] OUT; // result of the operation.
output ZERO;

//simulator registers
reg [`DATA_INDEX_LIMIT:0] OUT;
reg ZERO;

always @(OP1 or OP2 or OPRN)
begin

    case (OPRN)
        `ALU_OPRN_WIDTH'h01 : OUT = OP1 + OP2; // addition
        `ALU_OPRN_WIDTH'h02 : OUT = OP1 - OP2; // subtraction
        `ALU_OPRN_WIDTH'h03 : OUT = OP1 * OP2; // multiplication
        `ALU_OPRN_WIDTH'h04 : OUT = OP1 >> OP2; // shift_right
        `ALU_OPRN_WIDTH'h05 : OUT = OP1 << OP2; // shift_left
        `ALU_OPRN_WIDTH'h06 : OUT = OP1 & OP2; // bitwise and
        `ALU_OPRN_WIDTH'h07 : OUT = OP1 | OP2; // bitwise or
        `ALU_OPRN_WIDTH'h08 : OUT = ~(OP1 | OP2); // bitwise nor
        `ALU_OPRN_WIDTH'h09 : OUT = (OP1 < OP2)?1:0; // set less than
        default: OUT = `DATA_WIDTH'hxxxxxxxx;

    endcase

    ZERO = (OUT === 1'b0) ? 1'b1 : 1'b0;

end
```

Figure 2.2 ALU implementation

The ALU support the following operation:

- Addition
- Subtraction
- Multiplication
- Shift left
- Shift right
- Bitwise AND
- Bitwise OR
- Bitwise NOR
- Set less than

Finally, it will set the ZERO flag. If two operands are equal, it will set the ZERO flag to 1; otherwise, the ZERO flag will be set to 0. The ALU result will be used in the control unit for further operations.

3. Implement the register file

The register file is similar with the memory model. The figure 3.1 describes the register file. It has four signals including READ, WRITE, CLK, and RST. The register has two address port for reading and one address port for writing. It also has one data port for the data which will be written in the writing address. Besides, the data will be returned from the register after reading the input address R1 or R2. The following figure 3.2 is about how the register works.

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

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

// output reg
reg [`DATA_INDEX_LIMIT:0] data_ret_R1; // return R1 data register
reg [`DATA_INDEX_LIMIT:0] data_ret_R2; // return R2 data register

reg [`DATA_INDEX_LIMIT:0] reg_32x32 [0:`REG_INDEX_LIMIT]; // register memory storag
integer i; // index for reset operation

assign DATA_R1 = ((READ===1'b1)&&(WRITE===1'b0))?data_ret_R1:{`DATA_WIDTH{1'bz} };
assign DATA_R2 = ((READ===1'b1)&&(WRITE===1'b0))?data_ret_R2:{`DATA_WIDTH{1'bz} };
```

Figure 3.1 Description of the register

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

always @ (negedge RST or posedge CLK)
begin
if (RST === 1'b0)
begin
for(i = 0;i <= `REG_INDEX_LIMIT; i = i + 1)
    reg_32x32[i] = { `DATA_WIDTH{1'b0} };
end
else
begin
 if ((READ===1'b1)&&(WRITE===1'b0)) // read operation
 begin
        data_ret_R1 =  reg_32x32[ADDR_R1];
        data_ret_R2 =  reg_32x32[ADDR_R2];
 end
 else if ((READ===1'b0)&&(WRITE===1'b1)) // write operation
        reg_32x32[ADDR_W] = DATA_W;
end
end
endmodule
```

Figure 3.2 How the register works

When the RST signal is zero. It will initialize the register with all zeros. At either the negative edge of RST and positive edge of CLK, the data will be returned from certain addresses if the READ signal is 1 and the WRITE signal is 0. If the READ signal is 0 and the WRITE signal is 1, the data will be written at the given address.

4. Implement the Control Unit for the *instruction set*

After building memory, ALU and register, the control unit is designed for controlling different signal for data flow. It will implement different thing for different instructions in the ***cs147sec05 instruction set.***

The control unit has five stages, and they are PROC_FETCH, PROC_DECODE, PROC_EXE, PROC_MEM and PROC_WB. Different operations will be performed at the different stages. The figure 4.1 is the implementation of the state machine.

```
input CLK, RST;
// list of outputs
output [2:0] STATE;
reg [2:0] state;
reg [2:0] next_state;

assign STATE = state;

initial
begin
  state = 3'bxx;
  next_state = `PROC_FETCH;
end

always @ (negedge RST)
begin
  state = 3'bxx;
  next_state = `PROC_FETCH;
end

always @ (posedge CLK)
begin
  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
  endcase

end

endmodule;
```

Figure 4.1 State machine

Two internal registers will be used to represent the current state and next state of the state machine. At every positive edge of CLK, the next state will be assigned to the current state and the next state will has a new state according to the logic.

At the PROC_FETCH, the figure 4.1 shows the interface of this process.

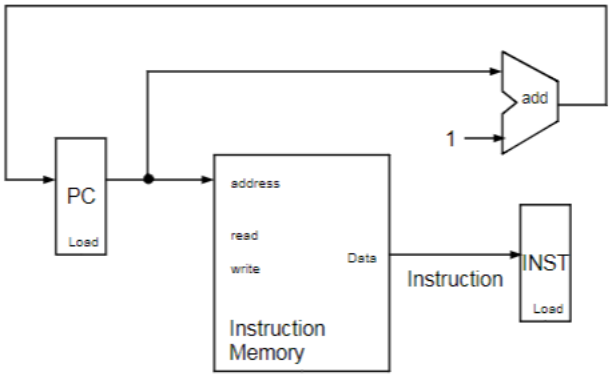

Figure 4.1 Interface of Fetch

The figure 4.2 shows what will happen at this stage. PC_REG is the instruction start address. It is assigned to the memory address to get the instruction code from the instruction memory. The READ signal for memory will be set to 1 and WRITE signal for

memory will be set to 0. The READ and WRITE signal for the register file will be set to 0.

```
if(proc_state === `PROC_FETCH)
begin
  MEM_ADDR_SET = PC_REG;
  MEM_READ_SET = 1'b1; MEM_WRITE_SET = 1'b0;
  RF_READ_SET = 1'b0; RF_WRITE_SET = 1'b0;
end
```

Figure 4.2 PROC_FETCH

At the PROC_DECODE, the figure 4.3 shows what will happen at this stage.

```
else if(proc_state === `PROC_DECODE)
begin
  INST_REG = MEM_DATA;
  // parse the instruction
  // R-type
  {opcode, rs, rt, rd, shamt, funct} = INST_REG;
  // I-type
  {opcode, rs, rt, immediate} = INST_REG;
  // J-type
  {opcode, address} = INST_REG;
  sign_extension = {{16{immediate[15]}}, immediate};
  zero_extension = {{16{1'b0}}, immediate};
  LUI = {immediate, {16{1'b0}}};
  jump_address = {{6{1'b0}}, address};
  RF_ADDR_R1_SET = rs;
  RF_ADDR_R2_SET = rt;
  RF_READ_SET = 1'b1;
end
```

Figure 4.3 PROC_DECODE

In the decoding stage, the instruction code will be returned from the memory. According to the cs147DV instruction set, it has three types of instruction including R type, I type and J type. R type instructions includes opcode, rs, rt, and rd registers, shamt (shift amount), and the funct code. I type instructions include opcode, rs and rt registers and immediate. R type has opcde and address. For some specific values like sign_extension and zero_extension will be evaluated according to the following formula.

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

The ALUI will be evaluated according to the following formula.

LUI = {imm, 16'b0}

The jump address will be evaluated according to the following formula.

Jump address = {6{1'b0}, address}

After that, the R1 register address will be rs, and the R2 register address will be rt. The READ signal will be set to 1.

At the PROC_EXE, it is related to the ALU which was designed above. The following figures show the different operations for R type, I type and J type.

```
case(opcode)
      //R type
      6'h00:
      begin
        ALU_OP1_SET = RF_DATA_R1;
        ALU_OP2_SET = RF_DATA_R2;
        case(funct)
          6'h20: ALU_OPRN_SET = 'h01;
          6'h22: ALU_OPRN_SET = 'h02;
          6'h2c: ALU_OPRN_SET = 'h03;
          6'h24: ALU_OPRN_SET = 'h06;
          6'h25: ALU_OPRN_SET = 'h07;
          6'h27: ALU_OPRN_SET = 'h08;
          6'h2a: ALU_OPRN_SET = 'h09;
          6'h01: begin ALU_OPRN_SET = 'h04; ALU_OP2_SET = shamt; end
          6'h02: begin ALU_OPRN_SET = 'h05; ALU_OP2_SET = shamt; end
          6'h08: PC_REG = RF_DATA_R1;
          default: $write("There is no such r type instruction.");
        endcase
      end
```

Figure 4.4 R type Execution

According to the different funct code:

Addition: R[rd] = R[rs] + R[rt];
Subtraction: R[rd] = R[rs] - R[rt];
Multiplication: R[rd] = R[rs] * R[rt];
Logical AND: R[rd] = R[rs] & R[rt];
Logical OR: R[rd] = R[rs] | R[rt];
Logical NOR: R[rd] = ~(R[rs] | R[rt]);
Set Less Than: R[rd] = (R[rs] < R[rt])?1:0;

For the above seven instructions, the register data R1 will be set to the operand one and the register data R2 will be set to the operand two.

For the shift left logical and shift right logical:
R[rd] = R[rs] << shamt;
R[rd] = R[rs] >> shamt;
the operand two will be set to shamt, and the operand one will keep the same.

For the jump Register:
PC = R[rs];
The PC will be assigned to the register data R1.

For all instructions, it will be used with different operation codes.

For I type instructions, the following figures show how it is implemented.

```
//I type
6'h08:
begin
  ALU_OP1_SET = RF_DATA_R1;
  ALU_OP2_SET = sign_extension;
  ALU_OPRN_SET = 'h01;
end
6'h1d:
begin
  ALU_OP1_SET = RF_DATA_R1;
  ALU_OP2_SET = sign_extension;
  ALU_OPRN_SET = 'h03;
end
6'h0c:
begin
  ALU_OP1_SET = RF_DATA_R1;
  ALU_OP2_SET = zero_extension;
  ALU_OPRN_SET = 'h06;
end
6'h0d:
begin
  ALU_OP1_SET = RF_DATA_R1;
  ALU_OP2_SET = zero_extension;
  ALU_OPRN_SET = 'h07;
end
```

Figure 4.5 I type

For the addi and muli operation,
Add imme: R[rt] = R[rs] + SignExImm
Mul imme: R[rt] = R[rs] * SignExImm
The above instructions will set the register data R1 to the operand one and the sign-extended immediate to the operand two.

For the andi and ori operation,
andi: R[rt] = R[rs] & ZeroExImm
ori: R[rt] = R[rs] | ZeroExImm
The above instructions will set the register data R1 to the operand one and the zero-extended immediate to the operand two.

```
6'h0f:
begin
end
```

Figure 4.6 LUI

When the opcode is 6'h0f, it is lui instruction. It doesn't need the ALU to finish the task, so it will be done at the write back stage.

```
6'h0a:
begin
  ALU_OP1_SET = RF_DATA_R1;
  ALU_OP2_SET = sign_extension;
  ALU_OPRN_SET = 'h09;
end
6'h04:
// for ZERO flag
begin
  ALU_OP1_SET = RF_DATA_R1;
  ALU_OP2_SET = RF_DATA_R2;
  ALU_OPRN_SET = 'h02;
end
6'h05:
begin
  ALU_OP1_SET = RF_DATA_R1;
  ALU_OP2_SET = RF_DATA_R2;
  ALU_OPRN_SET = 'h02;
end
6'h23:
begin
  ALU_OP1_SET = RF_DATA_R1;
  ALU_OP2_SET = sign_extension;
  ALU_OPRN_SET = 'h01;
end
6'h2d:
begin
  ALU_OP1_SET = RF_DATA_R1;
  ALU_OP2_SET = sign_extension;
  ALU_OPRN_SET = 'h01;
end
```

Figure 4.7 I type part 2

When opcode is 6'h0a, it is
slti: R[rt] = (R[s] < SignExItmm)?1:0
The above instruction will set the register data R1 to operand one and the sign-extended immediate to the operand two.

When the opcode is 6'h04 or 6'h05, it will check the zero flag and do more operations in the write back stage.
beq: if(R[rs] == R[rt])
PC = PC + 1 + BranchAddress
bne: if(R[rs] != R[rt])
PC = PC + 1 + BranchAddress
The above instruction will set the register data R1 to operand one and the register data R2 to the operand two.

When the opcode is 6'h23 and 6'h2d, it will get the address for further operations in the write back stage.
lw: R[rt] = M[R[rs] + SignExtImm]

sw: M[R[rs] + SignExItmm] = R[rt]
The above instruction will set the register data R1 to operand one and the register data R2 to the operand two.

For J type instructions, the following figures show how it is implemented.

```
//J type
6'h02:
begin
end
6'h03:
begin
end
6'h1b:
begin
  RF_ADDR_R1_SET = 0;
  ALU_OP1_SET = SP_REG;
  ALU_OP2_SET = 1;
  ALU_OPRN_SET = 'h02;
end
6'h1c:
begin
  RF_ADDR_W_SET = 0;
  ALU_OP1_SET = SP_REG;
  ALU_OP2_SET = 1;
  ALU_OPRN_SET = 'h01;
end
```

Figure 4.8 J type instructions

When opcode is 6'h02 or 6'h03, they are jmp and jal instructions and they will be done in the write back stage.
When opcode is 6'h1b or 6'h1c, it is push or pop instruction.
push: M[$sp] = R[0] $sp = $sp - 1
pop: $sp = $sp + 1 R[0] = M[$sp]
Both of them will set the operand one to SP_REG which is stack point for push operation and the operand two to 1 for other push operations.
Push operation will set the register address r1 to zero as reading address and pop operation will set the memory write address to 0 as writing address.

At the PROC_MEM stage, it has four operations with the memory model. Those are 'lw', 'sw'. 'push' and 'pop' instructions. The following figure 4.9shows how it is implemented.

```
else if(proc_state === `PROC_MEM)
begin
  MEM_READ_SET = 1'b0;
  MEM_WRITE_SET = 1'b0;
  case(opcode)
  6'h23:
  begin
    MEM_READ_SET = 1'b1;
    MEM_WRITE_SET = 1'b0;
    MEM_ADDR_SET = ALU_RESULT;
  end
  6'h2b:
  begin
    MEM_READ_SET = 1'b0;
    MEM_WRITE_SET = 1'b1;
    MEM_ADDR_SET = ALU_RESULT;
    MEM_DATA_SET = RF_DATA_R2;
  end
  6'h1b:
  begin
    MEM_READ_SET = 1'b0;
    MEM_WRITE_SET = 1'b1;
    MEM_ADDR_SET = SP_REG;
    MEM_DATA_SET = RF_DATA_R1;
    SP_REG = ALU_RESULT;
  end
  6'h1c:
  begin
    MEM_READ_SET = 1'b1;
    MEM_WRITE_SET = 1'b0;
    SP_REG = ALU_RESULT;
    MEM_ADDR_SET = SP_REG;
  end
  endcase
end
```

Figure 4.9 PROC_MEM stage

At the beginning, it will initialize with the READ and WRITE signal to 0 for memory. When the opcode is 6'h23, it will set the READ signal to 1 and the WRITE signal to 0 and set the alu result to the memory address for reading. It will read the data from the memory for operation in the write back stage. When the opcode is 6'h2b, it will set the READ signal to 0 and the WRITE signal to 1 and set the alu result to memory address for writing, the register data r2 will be written at this address for 'sw' instruction. When cpcode is 6'h1b, it will set the memory address to SP_REG which is stack pointer for pushing and the data which is pushed is register data R1. The new SP_REG will be set to alu result. For pop instruction, the opcode is 6'h1c. It will set the SP_REG to alu result and memory address to SP_REG.

At the PROC_WB stage, it will write the data back to the register file for different instructions. At the

beginning, it will increase PC_REG by one for next cycle operation. It will also set the register as read mode and memory as high Z mode.

The following figures show how it is implemented.

```
6'h00:
begin
  if(funct != 6'h08)
  begin
    RF_ADDR_W_SET = rd;
    RF_DATA_W_SET = ALU_RESULT;
  end
end
    6'h08:
    begin
      RF_ADDR_W_SET = rt;
      RF_DATA_W_SET = ALU_RESULT;
    end
    6'h1d:
    begin
      RF_ADDR_W_SET = rt;
      RF_DATA_W_SET = ALU_RESULT;
    end
    6'h0c:
    begin
      RF_ADDR_W_SET = rt;
      RF_DATA_W_SET = ALU_RESULT;
    end
    6'h0d:
    begin
      RF_ADDR_W_SET = rt;
      RF_DATA_W_SET = ALU_RESULT;
    end
```

Figure 4.10 Write back part1

When the opcode is 6'h00, it is R type operation.
When the funct is 6'h08, it is 'jr' operation. It has been done at the PROC_EXE stage. For other operations, it will write the alu result to the at the rd address. For all other operations in the above figure, it will write the alu result to the rt address.

```
6'h0f:
begin
  RF_ADDR_W_SET = rt;
  RF_DATA_W_SET = LUI;
end
6'h0a:
begin
  RF_ADDR_W_SET = rt;
  RF_DATA_W_SET = ALU_RESULT;
end
6'h04:
begin
  if(ZERO === 1)
  begin
    PC_REG = sign_extension + PC_REG;
  end
end
6'h05:
begin
  if(ZERO === 0)
    begin
      PC_REG = sign_extension + PC_REG;
    end
end
```

Figure 4.11 Write back part 2

When the opcode is 6'h08, it is LUI instruction.
lui: R[rt] = {Imm, 16'b0};
It will write the LUI value to the rt address. When the opcode is 6'h04 or 6'h05, it is beq and bne instruction. ZERO flag which is returned from the ALU will be check. For beq, if the ZERO flag is 1, PC_REG will be set to a new value. For bne, if the ZERO flag is 0, PC_REG will be set to a new value.
PC = PC + 1 + BranchAddress

```
6'h02:
begin
  PC_REG = jump_address;
end
6'h03:
begin
  RF_ADDR_W_SET = 31;
  RF_DATA_W_SET = PC_REG;
  PC_REG = jump_address;
end
6'h23:
begin
  RF_ADDR_W_SET = rt;
  RF_DATA_W_SET = MEM_DATA;
end
6'h1c:
begin
  RF_DATA_W_SET = MEM_DATA;
end
endcase
end
```

Figure 4.12 Write back part 3

When the opcode is 6'h02, it is jmp instruction.
PC = JumpAddress
The PC_REG will be set to the jump address.
When the opcode is 6'h03, it is 'jal' instruction.
jal: R[31] = PC + 1; PC = JumpAddress
The PC_REG will be written at the 31th register and the new PC_REG is the jump address.
When the opcode is 6'h23, it is push operation, the memory data will be written at the rt address. When the opcode is 6'h1c, the memory data will be written at the SP_REG address which we set at the PRO_MEM stage.

5. Combine ALU, register file, and the control unit as the processor

The processor consists of ALU, register file, and the control unit. It is implemented at the following figure 5.1.

```
`include "prj_definition.v"
module PROC_CS147_SEC05(DATA, ADDR, READ, WRITE, CLK, RST);
// output list
output [`ADDRESS_INDEX_LIMIT:0] ADDR;
output READ, WRITE;
// input list
input  CLK, RST;
// inout list
inout [`DATA_INDEX_LIMIT:0] DATA;

// net section
wire [`DATA_INDEX_LIMIT:0] rf_data_w, rf_data_r1, rf_data_r2, alu_op1, alu_op2, alu_result;
wire [`REG_ADDR_INDEX_LIMIT:0] rf_addr_w,  rf_addr_r1, rf_addr_r2;
wire [`ALU_OPRN_INDEX_LIMIT:0] alu_oprn;
wire rf_read, rf_write;
wire zero;

// instantiation section
// Control unit
CONTROL_UNIT cu_inst (.MEM_DATA(DATA),       .RF_DATA_W(rf_data_w),   .RF_ADDR_W(rf_addr_w),   .RF_ADDR_R1(rf_addr_r1),
                      .RF_ADDR_R2(rf_addr_r2), .RF_READ(rf_read),      .RF_WRITE(rf_write),     .ALU_OP1(alu_op1),
                      .ALU_OP2(alu_op2),      .ALU_OPRN(alu_oprn),     .MEM_ADDR(ADDR),         .MEM_READ(READ),
                      .MEM_WRITE(WRITE),      .RF_DATA_R1(rf_data_r1), .RF_DATA_R2(rf_data_r2), .ALU_RESULT(alu_result),
                      .ZERO(zero),            .CLK(CLK),               .RST(RST));
// register file
REGISTER_FILE_32x32 rf_inst (.DATA_R1(rf_data_r1), .DATA_R2(rf_data_r2), .ADDR_R1(rf_addr_r1), .ADDR_R2(rf_addr_r2),
                             .DATA_W(rf_data_w),   .ADDR_W(rf_addr_w),   .READ(rf_read),      .WRITE(rf_write),
                             .CLK(CLK),            .RST(RST));
// alu
ALU alu_inst (.OUT(alu_result), .ZERO(zero), .OP1(alu_op1), .OP2(alu_op2), .OPRN(alu_oprn));

endmodule;
```

Figure 5.1 Processor

The processor has the instance of the control, the register file and the ALU. It has all wires to connect different components. Besides, it has CLK and RST input signal. Also, it has one data bus and READ, WRITE, and ADDR as output.

6. Create Davinci v1.0

The Davinci v 1.0 is created by combining the processor and the memory system. The following figure 6.1 show the interface of Davinci v 1.0.

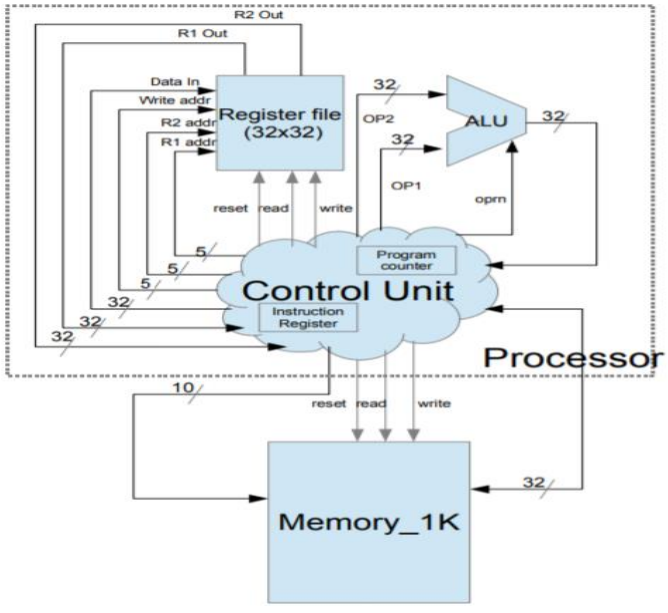

Figure 6.1 Davinci v1.0

The following figure 6.2 is the implementation of the Davinci v 1.0. It creates the instances of processor and the memory system. Besides, it has a parameter about the file reading and writing.

```
`include "prj_definition.v"
module DA_VINCI (DATA, ADDR, READ, WRITE, CLK, RST);
// Parameter for the memory initialization file name
parameter mem_init_file = "mem_content_01.dat";
// output list
output [`ADDRESS_INDEX_LIMIT:0] ADDR;
output READ, WRITE;
// input list
input  CLK, RST;
// inout list
inout [`DATA_INDEX_LIMIT:0] DATA;

// Instance section
// Processor instance
PROC_CS147_SEC05 processor_inst(.DATA(DATA),    .ADDR(ADDR), .READ(READ),
                                .WRITE(WRITE), .CLK(CLK),    .RST(RST));
// memory instance
defparam memory_inst.mem_init_file = mem_init_file;
MEMORY_64MB memory_inst(.DATA(DATA), .READ(READ), .WRITE(WRITE),
                        .ADDR(ADDR), .CLK(CLK),    .RST(RST));
endmodule;
```

Figure 6.2 Implementation of Davinci v1.0

The followings will be all test benches for memory model, ALU, register file, and Davinci v1.0.

7. Memory Testbench

```
// Start the operation
#10     RST=1'b0;
#10     RST=1'b1;
// Write cycle
for(i=1;i<10; i = i + 1)
begin
#10     DATA_REG=i; READ=1'b0; WRITE=1'b1; ADDR = i;
end

// Read Cycle
#10   READ=1'b0; WRITE=1'b0;
#5    no_of_test = no_of_test + 1;
      if (DATA !== {`DATA_WIDTH{1'bz}})
        $write("[TEST] Read %1b, Write %1b, expecting 32'hzzzzzzzz, got %8h [FAILED]\n", READ, WRITE, DATA);
      else
        no_of_pass  = no_of_pass + 1;

// test of write data
for(i=0;i<10; i = i + 1)
begin
#5      READ=1'b1; WRITE=1'b0; ADDR = i;
#5      no_of_test = no_of_test + 1;
        if (DATA !== i)
            $write("[TEST] Read %1b, Write %1b, expecting %8h, got %8h [FAILED]\n", READ, WRITE, i, DATA);
        else
            no_of_pass  = no_of_pass + 1;

end
```

Figure 7.1 Initialization and test read cycle and write data

To implement the test bench for memory (figure 7.1), the memory was initialized by a for loop and some data are written into the memory with corresponding address. At the read cycle, it will set both of WRITE and READ to 0, thus the DATA should be at high Z and the result will be 32bits z. After that, it will test the data which is written in the memory. In the process of initialization, it makes the address and data are equal. This process will check whether they are equal.

```
// test for the initialize data
for(i='h001000; i<'h001010; i = i + 1)
begin
#5      READ=1'b1; WRITE=1'b0; ADDR = i;
#5      no_of_test = no_of_test + 1;
        if (DATA !== load_data)
            $write("[TEST] Read %1b, Write %1b, Addr %7h, expecting %8h, got %8h [FAILED]\n",
                                                    READ, WRITE, ADDR, load_data, DATA);
        else
            no_of_pass  = no_of_pass + 1;
        load_data = load_data + 1;
end
#10    READ=1'b0; WRITE=1'b0; // No op

#10 $write("\n");
    $write("\tTotal number of tests %d\n", no_of_test);
    $write("\tTotal number of pass  %d\n", no_of_pass);
    $write("\n");
    $writememh("mem_dump_01.dat", mem_inst.sram_32x64m, 'h0000000, 'h000000f);
    $writememh("mem_dump_02.dat", mem_inst.sram_32x64m, 'h0001000, 'h000100f);
    $stop;

end
endmodule;
```

Figure 7.2 Test for the initialize data and write file

After that, in the figure 7.2, it will test the initialize data fort the data file. It starts at address ‘h0001000 and the first data is ‘h00414020 which is the load data. At every time of the loop, the load data will be increased by 1 to match the data in the file which is written in the memory. After simulating the testbench, the following waves (figure 7.1 and 7.2) show the process of read cycle, test for write data, and test for initialization data.

Figure 7.2 Waveform

```
VSIM 27> run -all
#
#       Total number of tests         27
#       Total number of pass          27
#
# ** Note: $stop    : C:/Users/pansh/Downloads/prj_02/mem_64MB_tb.v(107)
#    Time: 405 ns  Iteration: 0  Instance: /MEM_64MB_TB
# Break in Module MEM_64MB_TB at C:/Users/pansh/Downloads/prj_02/mem_64MB_tb.v line 107
```

Figure 7.3 Text output

At the same time, the text shows all the tests are passed. One for read cycle, ten for the first loop about testing the write data, and ten for the second loop for testing the initialization data.
The following figure 7.4 shows the result after running the memory testbench.

mem_dump_02 - Notepad

```
File  Edit  Format  View  Help
// memory data file (do not edit the following line - required for mem load use)
// instance=/MEM_64MB_TB/mem_inst/sram_32x64m
// format=hex addressradix=h dataradix=h version=1.0 wordsperline=1 noaddress
00414020
00414021
00414022
00414023
00414024
00414025
0041402a
0041402b
0041402c
0041402d
0041402e
0041402f
```

mem_dump_01 - Notepad

```
File  Edit  Format  View  Help
// memory data file (do not edit the following line - required for mem load use)
// instance=/MEM_64MB_TB/mem_inst/sram_32x64m
// format=hex addressradix=h dataradix=h version=1.0 wordsperline=1 noaddress
00000000
00000001
00000002
00000003
00000004
00000005
00000006
00000007
00000008
00000009
00000000
00000000
00000000
00000000
00000000
00000000
```

Figure 7.4 Output files

mem_dump_01 shows the initialization loop, 0 to 9 are written to the memory, and all others are zeors because the loop only run ten times. Besides, mem_dump_02 show the input file is read and the corresponding data is written into the memory. The initialization address start from ‘h0001000 and the first data is 00414020. The loop will run 16 times, so all data will be written to the memory. The following figure 7.5 is the input file mem_content_01.dat.

mem_content_01 - Notepad

```
File  Edit  Format  View  Help
@0001000
00414020 00414021 00414022 00414023
00414024 00414025 00414026 00414027
00414028 00414029 0041402a 0041402b
0041402c 0041402d 0041402e 0041402f

@002f00a
00514020 00514021 00514022 00514023
00514024 00514025 00514026 00514027
00514028 00514029 0051402a 0051402b
0051402c 0051402d 0051402e 0051402f
```

Figure 7.5 Input file

The input file is used to write the mem_dump_02 file.

8. ALU Testbench

The ALU Testbench is almost the same with the project one, but it has a test for the ZERO flag.

The following figure 8.1 shows the implementation of the ALU testbench.

```
// Instantiation of ALU
ALU ALU_INST_01(.OUT(r_net), .ZERO(zero), .OP1(op1_reg),
                .OP2(op2_reg), .OPRN(oprn_reg));

// Drive the test patterns and test
initial
begin
op1_reg=0;
op2_reg=0;
oprn_reg=0;

total_test = 0;
pass_test = 0;

// test 15 + 3 = 18
#5  op1_reg=15;
    op2_reg=3;
    oprn_reg=`ALU_OPRN_WIDTH'h01;
#5  test_and_count(total_test, pass_test,
                   test_golden(op1_reg,op2_reg,oprn_reg,r_net, zero));

//test 14 - 5 = 9
#5  op1_reg=14;
    op2_reg=5;
    oprn_reg=`ALU_OPRN_WIDTH'h02;
#5  test_and_count(total_test, pass_test,
                   test_golden(op1_reg,op2_reg,oprn_reg,r_net, zero));

//test 15 - 15 = 0
#5  op1_reg=15;
    op2_reg=15;
    oprn_reg=`ALU_OPRN_WIDTH'h02;
#5  test_and_count(total_test, pass_test,
                   test_golden(op1_reg,op2_reg,oprn_reg,r_net, zero));

//test 10 * 9 = 90
#5  op1_reg=10;
    op2_reg=9;
    oprn_reg=`ALU_OPRN_WIDTH'h03;
#5  test_and_count(total_test, pass_test,
                   test_golden(op1_reg,op2_reg,oprn_reg,r_net, zero));

//test 16 >> 2 = 4
#5  op1_reg=16;
    op2_reg=2;
    oprn_reg=`ALU_OPRN_WIDTH'h04;
#5  test_and_count(total_test, pass_test,
                   test_golden(op1_reg,op2_reg,oprn_reg,r_net, zero));

//test 7 << 3 = 56
#5  op1_reg=7;
    op2_reg=3;
    oprn_reg=`ALU_OPRN_WIDTH'h05;
#5  test_and_count(total_test, pass_test,
                   test_golden(op1_reg,op2_reg,oprn_reg,r_net, zero));

//test 3 & 5 = 1
#5  op1_reg=3;
    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, zero));

//test 10 | 6 = 12
#5  op1_reg=10;
    op2_reg=6;
    oprn_reg=`ALU_OPRN_WIDTH'h07;
#5  test_and_count(total_test, pass_test,
                   test_golden(op1_reg,op2_reg,oprn_reg,r_net, zero));

//test ~(1 | 4) = 4294967290
#5  op1_reg=1;
    op2_reg=4;
    oprn_reg=`ALU_OPRN_WIDTH'h08;
#5  test_and_count(total_test, pass_test,
                   test_golden(op1_reg,op2_reg,oprn_reg,r_net, zero));

//test 6 > 5 = 0
#5  op1_reg=6;
    op2_reg=5;
    oprn_reg=`ALU_OPRN_WIDTH'h09;
#5  test_and_count(total_test, pass_test,
                   test_golden(op1_reg,op2_reg,oprn_reg,r_net, zero));

#5  $write("\n");
    $write("\tTotal number of tests %d\n", total_test);
    $write("\tTotal number of pass  %d\n", pass_test);
    $write("\n");
    $stop; // stop simulation here
end
```

Figure 8.1 All test cases

At the beginning, we initialize the instance of the ALU. All of above are the test cases for the ALU, and they are different instructions that the ALU can implement.

```
task test_and_count;
inout total_test;
inout pass_test;
input test_status;

integer total_test;
integer pass_test;
begin
    total_test = total_test + 1;
    if (test_status)
    begin
        pass_test = pass_test + 1;
    end
end
endtask
```

Figure 8.2 Record passed tests

Whiling testing each case, we will use task test_and_count to record all the tests and all passed tests. It will use the result of the function test_golden which is at following figure 8.2. test_golden will be either true or false which is gotten from comparing the golden and the result from the ALU. It will also write whether the current test is failed or passed.

```
function test_golden;
input [`DATA_INDEX_LIMIT:0] op1;
input [`DATA_INDEX_LIMIT:0] op2;
input [`ALU_OPRN_INDEX_LIMIT:0] oprn;
input [`DATA_INDEX_LIMIT:0] res;
input [0:0] zero;

reg [`DATA_INDEX_LIMIT:0] golden; // expected result
begin
    $write("[TEST] %0d ", op1);
    case(oprn)
        `ALU_OPRN_WIDTH'h01 : begin $write("+ "); golden = op1 + op2; end
        `ALU_OPRN_WIDTH'h02 : begin $write("- "); golden = op1 - op2; end
        `ALU_OPRN_WIDTH'h03 : begin $write("* "); golden = op1 * op2; end
        `ALU_OPRN_WIDTH'h04 : begin $write(">> "); golden = op1 >> op2; end
        `ALU_OPRN_WIDTH'h05 : begin $write("<< "); golden = op1 << op2; end
        `ALU_OPRN_WIDTH'h06 : begin $write("& "); golden = op1 & op2; end
        `ALU_OPRN_WIDTH'h07 : begin $write("| "); golden = op1 | op2; end
        `ALU_OPRN_WIDTH'h08 : begin $write("~| "); golden = ~(op1 | op2); end
        `ALU_OPRN_WIDTH'h09 : begin $write("< "); golden = op1 < op2?1:0; end
        default: begin $write("? "); golden = `DATA_WIDTH'hx; end
    endcase
    $write("%0d = %0d , got %0d, zero_flag = %0d... ", op2, golden, res, zero);

    test_golden = (res === golden)?1'b1:1'b0; // case equality
    if (test_golden)
        $write("[PASSED]");
    else
        $write("[FAILED]");
    $write("\n");
end
endfunction

endmodule
```

Figure 8.3 test_golden

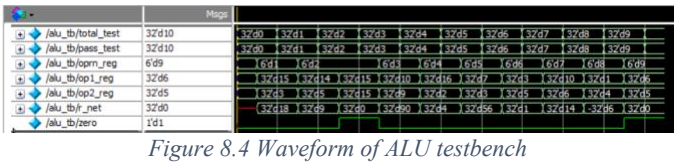

Figure 8.4 Waveform of ALU testbench

The above figure 8.4 show the waveform of this testbench. It shows all the operands, operation codes and their corresponding outputs.

```
VSIM 30> run -all
# [TEST] 15 + 3 = 18 , got 18, zero_flag = 0... [PASSED]
# [TEST] 14 - 5 = 9 , got 9, zero_flag = 0... [PASSED]
# [TEST] 15 - 15 = 0 , got 0, zero_flag = 1... [PASSED]
# [TEST] 10 * 9 = 90 , got 90, zero_flag = 0... [PASSED]
# [TEST] 16 >> 2 = 4 , got 4, zero_flag = 0... [PASSED]
# [TEST] 7 << 3 = 56 , got 56, zero_flag = 0... [PASSED]
# [TEST] 3 & 5 = 1 , got 1, zero_flag = 0... [PASSED]
# [TEST] 10 | 6 = 14 , got 14, zero_flag = 0... [PASSED]
# [TEST] 1 ~| 4 = 4294967290 , got 4294967290, zero_flag = 0... [PASSED]
# [TEST] 6 < 5 = 0 , got 0, zero_flag = 1... [PASSED]
#
#       Total number of tests          10
#       Total number of pass           10
#
# ** Note: $stop    : C:/Users/pansh/Downloads/prj_02/alu_tb.v(127)
#    Time: 105 ns  Iteration: 0  Instance: /alu_tb
# Break in Module alu_tb at C:/Users/pansh/Downloads/prj_02/alu_tb.v line 127
```

Figure 8.5 Final result

While simulating the testbench, test results will be displayed in the transcript window. We can see how many tests total and how many passed tests.

9. Register File Testbench

The register file testbench is similar with the memory testbench. The difference is that there is a file test in the memory testbench

```
// Register Instance
REGISTER_FILE_32x32 reg_inst(.DATA_R1(DATA_R1), .DATA_R2(DATA_R2), .ADDR_R1(ADDR_R1), .ADDR_R2(ADDR_R2),
                             .DATA_W(DATA_W), .ADDR_W(ADDR_W), .READ(READ), .WRITE(WRITE), .CLK(CLK), .RST(RST));

initial
begin
RST = 1'b1;
READ = 1'b0;
WRITE = 1'b0;
no_of_test = 0;
no_of_pass = 0;

// Start the operation
#10    RST = 1'b0;
#10    RST = 1'b1;
// Write cycle
for(i = 1;i < 10; i = i + 1)
begin
#10   DATA_W = i; READ = 1'b0; WRITE = 1'b1; ADDR_W = i;
end
```

Figure 9.1 Initialization block

At the beginning, we initialize the READ and WRITE signal to 0 and all tests and passed tests to 0. In the initialization block, data is written at the specific address by using a loop. Data is from one to ten and address is also from one to ten.

```
//read cycle
#10   READ=1'b0; WRITE=1'b0;
#5    no_of_test = no_of_test + 1;
      if (DATA_R1 !== {`DATA_WIDTH{1'bz}} && DATA_R2 !== {`DATA_WIDTH{1'bz}})
        $write("[TEST] Read %1b, Write %1b, expecting 32'hzzzzzzzz, got %8h [FAILED]\n", READ, WRITE, DATA_R1);
      else
        no_of_pass  = no_of_pass + 1;

//test write data from address R1
for(i = 1; i < 10; i = i + 1)
begin
#5      READ=1'b1; WRITE=1'b0; ADDR_R1 = i;
#5      no_of_test = no_of_test + 1;
        if (DATA_R1 !== i)
            $write("[TEST] Read %1b, Write %1b, expecting %8h, got %8h [FAILED]\n", READ, WRITE, i, DATA_R1);
        else
            no_of_pass  = no_of_pass + 1;
end
#20
//test write data from address R2
for(i = 1; i < 10; i = i + 1)
begin
#5      READ = 1'b1; WRITE = 1'b0; ADDR_R2 = i;
#5      no_of_test = no_of_test + 1;
        if (DATA_R2 !== i)
            $write("[TEST] Read %1b, Write %1b, expecting %8h, got %8h [FAILED]\n", READ, WRITE, i, DATA_R2);
        else
            no_of_pass  = no_of_pass + 1;
end

#10    READ = 1'b0; WRITE = 1'b0; // No op

#10 $write("\n");
    $write("\tTotal number of tests %d\n", no_of_test);
    $write("\tTotal number of pass  %d\n", no_of_pass);
    $write("\n");
    $stop;
end
endmodule;
```

Figure 9.2 Tests block

The test cases (figure 9.2) include three parts. The first part is read cycle. It is to test the high Z when both READ and WRITE signal are zeros. After that, the first for loop is to test data which is written in the register from read address R1. The second loop is to test data which is written in the register from read address R2.

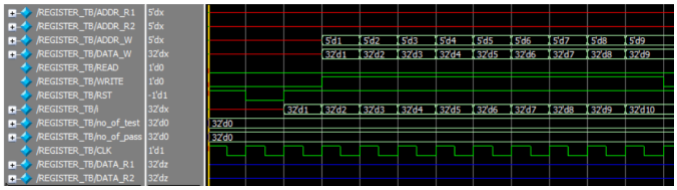

Figure 9.3 Waveform of initialization block

The figure 9.3 show the integer i is from 1 to 10. The write address and written data are from 1 to 9. It is the initialization block.

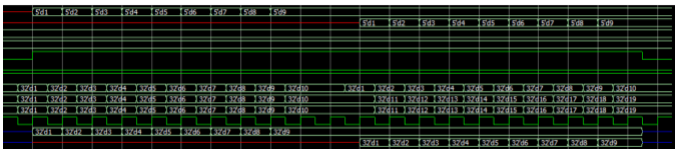

Figure 9.4 Waveform of the test block

In the figure 9.4, the blue line shows that the DATA_R1 and DATA_R2 are at high Z state which means the first part is passed. Then both of the ARRD_R1 and DATA_R1 are from 1 to 9. Integer i is from one to 10. Since the loop only run 9 times, the second part passed. After that, both of the ARRD_R2 and DATA_Rw are from 1 to 9. Integer i is from one to 10. Since the loop only run 9 times, the third part passed.

```
VSIM 3> run -all
#
#       Total number of tests          19
#       Total number of pass           19
#
# ** Note: $stop    : C:/Users/pansh/Downloads/prj_02/register_tb.v(105)
#    Time: 345 ns  Iteration: 0  Instance: /REGISTER_TB
# Break in Module REGISTER_TB at C:/Users/pansh/Downloads/prj_02/register_tb.v line 105
```

Figure 9.5 Total tests and passed tests

The above figure 9.5 show total number of tests and total number of passed tests. There is one test for high Z in the if statement. In two for loops, total number of tests = 9 + 9 = 18. Thus, all tests are passed.

10. Davinci v1.0 Testbench

To test the whole system, three files are created and passed to the system. By comparing the dump

file and the golden file, we can see whether the Davinci v1.0 works correctly.

```
`include "prj_definition.v"
module DA_VINCI_TB;
 // output list
 wire [`ADDRESS_INDEX_LIMIT:0] ADDR;
 wire READ, WRITE, CLK;
 // inout list
 wire [`DATA_INDEX_LIMIT:0] DATA;

 // reset
 reg RST;

 // Clock generator instance
 CLK_GENERATOR clk_gen_inst(.CLK(CLK));

 // DA_VINCI v1.0 instance
 //defparam da_vinci_inst.mem_init_file = "fibonacci.dat";
 //defparam da_vinci_inst.mem_init_file = "RevFib.dat";
 defparam da_vinci_inst.mem_init_file = "tests.dat";
 DA_VINCI da_vinci_inst(.DATA(DATA), .ADDR(ADDR), .READ(READ),
                        .WRITE(WRITE), .CLK(CLK), .RST(RST));

 initial
 begin
 RST=1'b1;
 #5 RST=1'b0;
 #5 RST=1'b1;

 // TBD: rest of the test code goes here.

 //# 20 $stop;
 #5000  //$writememh("RevFib_mem_dump.dat", da_vinci_inst.memory_inst.sram_32x64m, 'h03fffff0, 'h03ffffff);
        //$writememh("fibonacci_mem_dump.dat", da_vinci_inst.memory_inst.sram_32x64m, 'h01000000, 'h0100000f);
        $writememh("tests_mem_dump.dat", da_vinci_inst.memory_inst.sram_32x64m, 'h01000000, 'h0100000f);
        $stop;

 end
 endmodule;
```

Figure 10.1 Davinci v1.0 testbench

The figure 10.1 is the Davinci v1.0 testbench. It will create the instance of Davinci v1.0. There are three files total, so the file is passed to the system one by one through memory model. While execute the instruction in the file, some data will be written to the dump file.

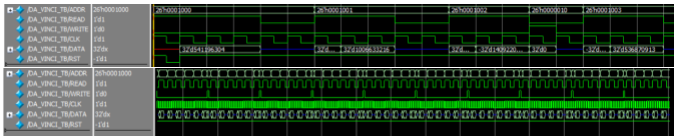

Figure 8.2 Waveform of testbench

In the figure 8.2, it is the waveform when the tests.dat was passed into the system. The second part is the whole waveform. The first part is the one after zooming in. The first part of the figure shows the address in increasing by 1 and the data is changing all the time as the instructions are implemented.

```
VSIM 6> run -all
# ** Note: $stop    : C:/Users/pansh/Downloads/prj_02/da_vinci_tb.v(56)
#    Time: 5010 ns  Iteration: 0  Instance: /DA_VINCI_TB
# Break in Module DA_VINCI_TB at C:/Users/pansh/Downloads/prj_02/da_vinci_tb.v line 56
```

Figure 10.2 Text output of the testbench

There is no text output in this testbench because all data are written in the .dat files.

fibonacci - Notepad

File Edit Format View Help

```
@0001000
20420001 //          addi r[2], r[2], 0x0001;
3C000100 //          lui  r[0], 0x0100;
AC010000 //          sw   r[1], r[0], 0x0000;
20000001 // loop:    addi r[0], r[0], 0x0001;
AC020000 //          sw   r[2], r[0], 0x0000;
20430000 //          addi r[3], r[2], 0x0000;
00411020 //          add  r[2], r[2], r[1];
20610000 //          addi r[1], r[3], 0x0000;
08001003 //          jmp  loop;
```

fibonacci_mem_dump - Notepad

File Edit Format View Help

```
// 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
```

fibonacci_mem_dump.golden - Notepad

File Edit Format View Help

```
00000000
00000001
00000001
00000002
00000003
00000005
00000008
0000000d
00000015
00000022
00000037
00000059
00000090
000000e9
00000179
00000262
```

```
// 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
00000002
00000003
00000005
00000008
0000000d
00000015
00000022
00000037
00000059
00000090
000000e9
00000179
00000262
```

Figure 10.3 fibonacci

The first file is the one who calculate the Fibonacci. According to the assembly code, they are converted to the machine code. The program is to calculate the fibonacci numbers. After passing the file to the system, the result will be written into the fibonacci.mem_dump.dat. Compared with the golden file, we can see it is the same. Thus, the test is passed.

tests - Notepad

File Edit Format View Help

```
@0001000
20420010 //          addi r[2], r[2], 0x0010;
3C000100 //          lui  r[0], 0x0100;
AC010000 //          sw   r[1], r[0], 0x0000;
20000001 // loop:    addi r[0], r[0], 0x0001;
AC020000 //          sw   r[2], r[0], 0x0000;
20430000 //          addi r[3], r[2], 0x0000;
00411022 //          sub  r[2], r[2], r[1];
20610000 //          addi r[1], r[3], 0x0000;
08001003 //          jmp  loop;
```

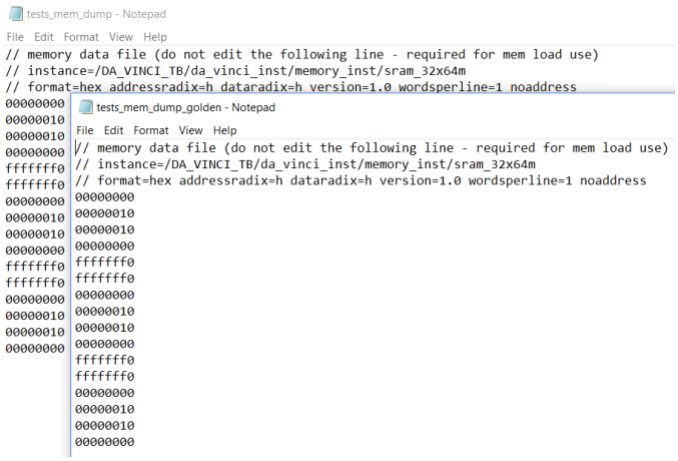

Figure 10.4 tests

The figure 10.4, we set the first data to 00000010; after that, the subtraction will be implemented in the for loop. The program will produce a circular pattern. The result is written into the test_mem_dump.dat. Compared with the test_mem_dump.dat, it is the same. Thus, it is passed.

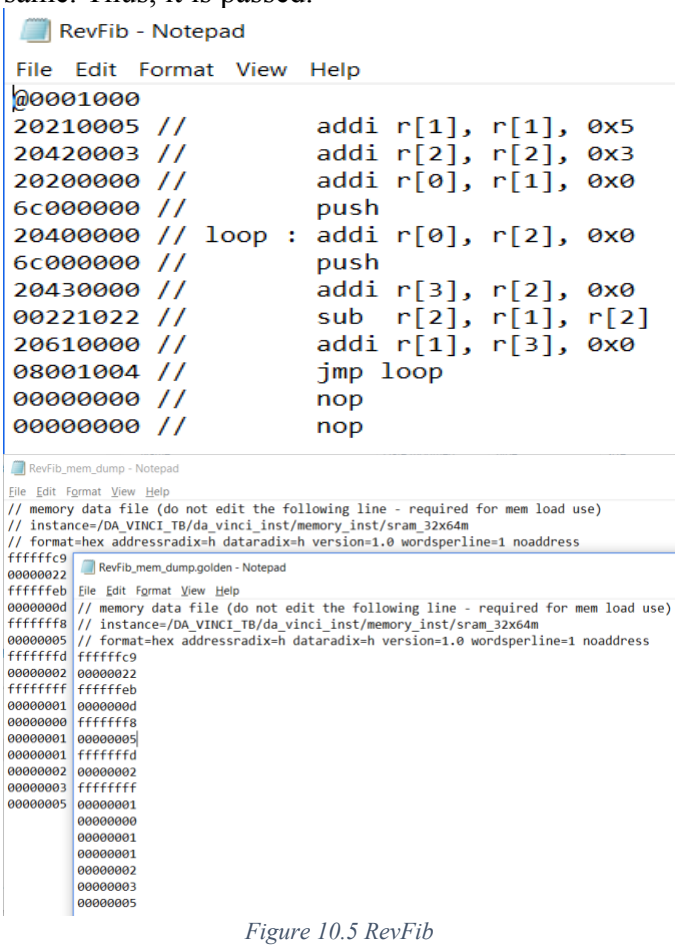

Figure 10.5 RevFib

The first test is almost the same with the fibonnacci numbers. The only difference is that the initialization numbers. The result is written to the Revfib_mem_dump.dat. Compared with the golden file, it is the same. Thus, it is passed.

11. Conclusion

In project2, I have a better understanding on the Verilog programming language in the process of designing and implemented the Davinci v1.0. The most important thing that I learned is that how the computer works in the machine level. I learned different function of different component including memory, register, control unit, and ALU. I know how the data is transferred from one component to the other component. The control unit is key to handle the different signal and data path. I know what operations will be done in different stage by using state machine. Besides, I have more deeper comprehension about how hardware works exactly.