Design of CPU

VLSI ARCHITECTURE (MELZG642) ASSIGNMENT-3

ANAND S, 2021HT80003 | VLSI Architecture | 24-04-2022

Contents

Table of Figures

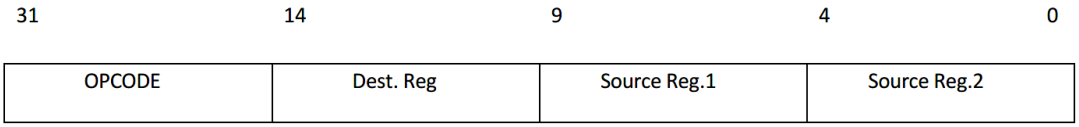
1. Problem statement

Given that the sequence of instructions to be executed by the processor is guaranteed to be free from pipeline hazards, design a 4 – stage (Instruction Fetch; Decode and read operand; execute; write back) pipelined RISC processor that can execute following register to – register instructions with a throughput of one instruction per clock –cycle: MUL , Barrel shifter XOR, NOR.The multiplier and barrel shifter, ALU designed in assignment-1 and 2 should be used here.

STEP 0: Draw the detailed architecture level diagram of the processor, naming and depicting the various architectural blocks (e.g. register file, instruction memory, ALU, pipeline registers, PC and combinational functional blocks etc).
STEP 1: Create Verilog/ VHDL behavioral models for each of the architectural blocks. (Register file, Instruction memory, ALU, Pipeline registers, PC etc)
STEP 2: Build the top level **structural model** of the processor by instantiating and interconnecting the architectural blocks created in step 1.
STEP 3: Initialize the instruction memory with a program consisting of 4 instructions:
0000 MUL reg3, reg2, reg1
0004 SHIFT reg6, reg5, reg4 0008 XOR reg9, reg8, reg7 0012 NOR reg 13, reg11, reg10

Initialize the register file with the following data
reg2 = 60 reg1 = 40
reg5 = 40 reg4 = 60
reg7 = FFFF856D reg8 = EEEE3721
reg10 = 1FFF756F reg11 = FFFF765E
Initialize PC with 0000 address.

The instruction format is:



Opcode MUL 0---------- 001

SHIFT 0----------010

XOR 0---------- 011

NOR 0 ---------- 100

The register file has 32 number of 32 bit registers. The data in registers is stored as integers in 2's complement binary representation. Assume that register file has two 32-bit read ports: A and B for data reading and one 32 bit write port: C for data writing. At the rising edge of the clock the read ports A and B output the data from the registers whose addresses are present at port A address and port B address respectively if the enable read port A and enable read port B signals are true. At the falling edge of a clock, data is written to the register whose address is present at Port C address if write enable Port C is true. Design the pipeline registers such that they are latched at the rising edge of the clock. Specify the size and format of all pipeline registers including the fields holding the decoded control signals as well as data.
The instruction memory is of size 128 bytes. A read operation from the memory outputs 4 consecutive bytes of information (starting from the byte address provided to the memory) at the positive edge of clock if the read enable instruction memory signal is true. Byte addresses must be aligned and to be provided to memory must be either 0 or multiples of 4.

2. Design

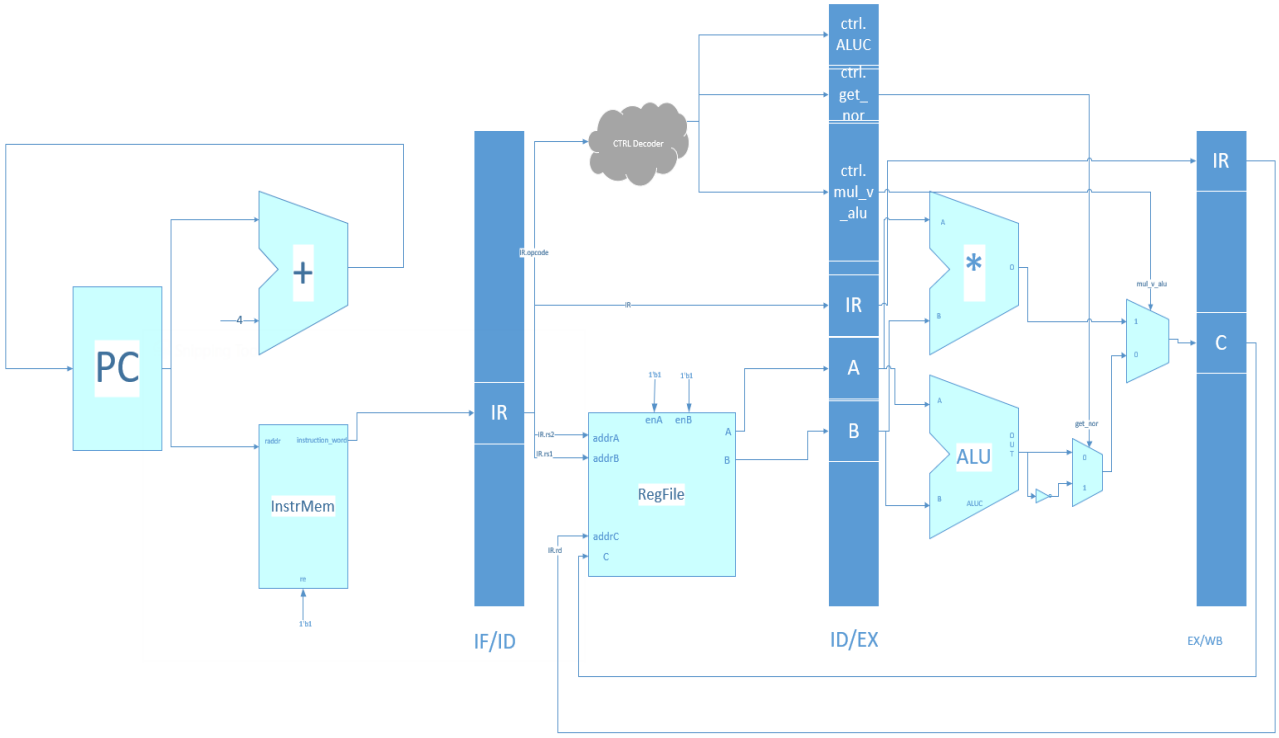


Figure 1 Architecture of CPU with pipelining

As shown, in Figure 1, the CPU consists of the structural implementation of PC, Instruction memory, register file, ALU, multiplier and pipeline blocks for each pipeline stage boundary.

The Multiplier and ALU are taken as is from Assignment 1 and Assignment 2 respectively. The multiplier used is a 16x9 multiplier. The ALU takes in 32 bit A and B inputs and provides 32 bit output. For shift operations, only 0 to 31 values are valid in B.

The CPU design is adapted from standard MIPS pipeline architecture, but is optimized to run only the provided instructions. Since there is no memory write back required, the design is actually only a 4 stage pipeline with IF, ID, EX and WB stages.

The entire design is implemented in system-verilog IEEE 1800-2009 standard. The use of SV structures are made to increase the readability of the HDL code. As the ALU and Multiplier code is same as in assignment-1, they are not shown in section “HDL code”.

3. HDL code

3.1 CPU_PKG.SV

The below code consists of SV package that encapsulates the

```
// ------------------------------------------------------
// VLSI Arch Assignment - 3
// Author: Anand S
// BITSID: 2021HT80003
// ------------------------------------------------------

package cpu_pkg;

    // ------------------------------------------------
    // Parameters
    // ------------------------------------------------
    localparam INSTR_MEM_DEPTH = 128/4; // 128 bytes in words
    localparam REG_FILE_DEPTH  = 32;
    localparam MD_WD   = 16;
    localparam MR_WD   = 9;
    localparam MDMR_WD = MD_WD+MR_WD;
    localparam DATA_WDTH = 32;

    // ------------------------------------------------
    // Typedefs
    // ------------------------------------------------
    typedef enum logic [16:0] {
        NA    = 17'b0_0000_0000_0000_0000,
        MUL   = 17'b0_0000_0000_0000_0001,
        SHIFT = 17'b0_0000_0000_0000_0010,
        XOR   = 17'b0_0000_0000_0000_0011,
        NOR   = 17'b0_0000_0000_0000_0100
    } opcode_t;

    typedef logic [$clog2(REG_FILE_DEPTH)-1:0] reg_t;

    typedef struct packed {
        opcode_t    opcode; // [31:15]
        reg_t       rd;     // [14:10]
        reg_t       rs1;    // [9:5]
        reg_t       rs2;    // [4:0]
    } instruction_t;

    typedef instruction_t instr_array_t [INSTR_MEM_DEPTH];

    typedef logic [$clog2(INSTR_MEM_DEPTH)-1:0] instr_mem_addr_t;

    typedef logic signed [31:0] sint32_t;

    typedef logic [31:0] uint32_t;

    typedef struct {
        logic       en;
        sint32_t    data; // write data
```

```
        reg_t       addr;
    } port_t;

    typedef struct {
        logic       en;
        reg_t       addr;
    } port_in_t;

    typedef struct {
        instruction_t IR;
    } IF_ID_pipe_t;

    typedef struct {
        logic mul_v_alu;
        logic get_nor;
        logic [3:0] aluc;
    } control_t;

    typedef struct {
        instruction_t IR;
        sint32_t      A;
        sint32_t      B;
        control_t ctrl;
    } ID_EX_pipe_t;

    typedef struct {
        instruction_t   IR;
        sint32_t        C;
    } EX_WB_pipe_t;

endpackage: cpu_pkg
```

3.2 INSTR_MEM.SV

The instruction memory is coded as per specification. As there is no requirement in the current specification to fill the memory through explicit writes, the HDL code is similar to that of a ROM. The initialization of the memory has been done directly in the RTL instead of using readmemh Verilog directives for simplicity and better illustration.

The specification also points that the read needs to be on a positive edge of the clock – This can be enforced in two ways :-

1. By external constraints: - The memory read port can be made an asynchronous read port, but the "raddr" port needs to be ensured to change only on posedge of the clk.
2. By internal design: - The memory should have a registered read port that changes value on posedge of the clk. This matches the specification, but it adds additional latency in the pipeline, and correspondingly all the pipelines need to be held to meet the required timing.

```
// ------------------------------------------------------
// VLSI Arch Assignment - 3
// Author: Anand S
// BITSID: 2021HT80003
// ------------------------------------------------------

module instr_mem
import cpu_pkg::*;
(
    input   logic               clk, resetn, re,
    input   instr_mem_addr_t    raddr,
    output  instruction_t       instruction_word
);

    // Instr mem can be implemented as Read-Only for this application
    instr_array_t instr_mem_array;
    logic [($clog2(INSTR_MEM_DEPTH/4))-1:0] raddr_int;

    assign raddr_int = raddr >> 2; // Ensures word alignment

    always_comb  begin: mem_init
        foreach (instr_mem_array[i]) begin
            instr_mem_array[i] = '{opcode: NA, default: 0};
            // synopsys translate off
            $display ("instr_mem_array[%0d] = %p", i,
instr_mem_array[i]);
            // synopsys translate on
        end
        instr_mem_array[0] = '{opcode: MUL,   rd: 3, rs1: 2, rs2: 1};
        instr_mem_array[1] = '{opcode: SHIFT, rd: 6, rs1: 5, rs2: 4};
        instr_mem_array[2] = '{opcode: XOR,   rd: 9, rs1: 8, rs2: 7};
        instr_mem_array[3] = '{opcode: NOR,   rd:13, rs1:11, rs2:10};
    end: mem_init

`ifdef RD_STALL_ALLOWED
    always_ff @(posedge clk, negedge resetn) begin
        if (!resetn) begin
            instruction_word <= '{opcode:NA, default:0};
        end else begin
            if (re) instruction_word <= instr_mem_array[raddr_int];
        end
    end
`else
    // asynch read port
    assign instruction_word = instr_mem_array[raddr_int];
`endif

endmodule: instr_mem
```

3.3 REG_FILE.SV

The register file is also coded in two different ways :- One with the asynchronous read port, and the other with registered read port. The selection of the RTL code is based on a macro "RD_STALL_ALLOWED" (same as in instr_mem.sv). The initialization of the memory has been done directly in the RTL instead of using readmemh Verilog directives for simplicity and better illustration.

The specification points that the read needs to be on a positive edge of the clock – This can be enforced in two ways :-

1. By external constraints: - The memory read port can be made an asynchronous read port, but the "addrA" and "addrB" ports needs to be ensured to change only on posedge of the clk. If the "enA" and "enB" ports are not set, the memory provides "0", otherwise it provides the data at the pointed address.
2. By internal design: - The memory should have a registered read port that changes value on posedge of the clk. This matches the specification, but it adds additional latency in the pipeline, and correspondingly all the pipelines need to be held to meet the required timing. If the "enA" and "enB" ports are not set, the previous value is stored, otherwise the read value is updated on the next posedge.

NOTE: As the "SHIFT" operation is done on reg5 and reg4, the value of 0x60 is much greater 32, the maximum allowed shift value in the ALU design as coded in assignment-2. If the reg4 content is maintained at 0x60,we cannot see any shifting happening (as shift value is 0). Thus the reg4 value has been changed to a smaller value during the initialization phase.

```
// ------------------------------------------------------
// VLSI Arch Assignment - 3
// Author: Anand S
// BITSID: 2021HT80003
// ------------------------------------------------------

module reg_file
import cpu_pkg::*;
(
    input   logic       clk, resetn,
    input   logic       enA, enB, enC,
    input   reg_t       addrA, addrB, addrC,
    input   sint32_t    C,          // write data port
    output  sint32_t    A, B        // read data ports
);

    sint32_t rf_array [REG_FILE_DEPTH];

    always_ff @(negedge clk, negedge resetn) begin
        if (!resetn) begin: init
            foreach (rf_array[i]) begin
                rf_array[i]    <= '{default: 0};
```

```
            end
            rf_array[1] <= 'h40;
            rf_array[2] <= 'h60;
            rf_array[4] <= 'h02;
            rf_array[5] <= 'h40;
            rf_array[7] <= 'hFFFF856D;
            rf_array[8] <= 'hEEEE3721;
            rf_array[10]<= 'h1FFF756F;
            rf_array[11]<= 'hFFFF765E;
        end: init
        else begin
            if (enC) rf_array[addrC] <= C;
        end
    end

`ifdef RD_STALL_ALLOWED
    always_ff @(posedge clk, negedge resetn) begin
        if (!resetn) begin
            A <= 0;
            B <= 0;
        end else begin
            if (enA) A <= rf_array[addrA];
            if (enB) B <= rf_array[addrB];
        end
    end
`else
    assign A = (enA) ? rf_array[addrA] : 1'b0;
    assign B = (enB) ? rf_array[addrB] : 1'b0;
`endif

endmodule: reg_file
```

3.4 IF_ID_PIPE.SV

The if_id_pipe is the pipeline between the Instruction Fetch (IF) and Instruction decode (ID) stage. This stage pipelines the Instruction word (IR) from the Instruction fetch stage.

```
// VLSI Arch Assignment - 3
// Author: Anand S
// BITSID: 2021HT80003
// ------------------------------------------------------

module if_id_pipe
import cpu_pkg::*;
(
    input  logic clk, resetn,
    input  instruction_t IR_next,
    output IF_ID_pipe_t if_id_pipe
);

    always_ff @(posedge clk, negedge resetn) begin
```

```
        if (!resetn)
            if_id_pipe <= '{IR: '{opcode:NA, default:0}, default:0};
        else
            if_id_pipe <= '{IR: IR_next};
    end


endmodule: if_id_pipe
```

3.5 ID_EX_PIPE.SV

The id_ex_pipe is the pipeline between the Instruction Decode (ID) and Instruction Execute (EX) stage. This stage pipelines the following items from the Instruction Decode stage : -

1. Instruction word (IR)
2. ALU/MUL input 1 (A)
3. ALU/MUL input 2 (B)
4. control signals decoded in the ID stage by the control word decoder (ctrl) which consists of :-
    1. select signal to choose between the output of multiplier or ALU based on the instruction type (whether it is a “MUL” or anything else) (mul_v_alu).
    2. select signal to choose between the normal ALU output or the inverted value. Since the ALU does not support a NOR instruction, when a “NOR” opcode is seen, the decoder passes this signal to ensure the ALU does an “OR” operation internally, and the ALU output is then sent through and inverter to get the actual NOR output (get_nor).
    3. The actual ALU control signals that need to be driven to the ALUC ports of the ALU (ALUC).

```
// ------------------------------------------------------
// VLSI Arch Assignment - 3
// Author: Anand S
// BITSID: 2021HT80003
// ------------------------------------------------------

module id_ex_pipe
import cpu_pkg::*;
(
    input logic clk, resetn,
    input instruction_t IR_next,
    input sint32_t A_next, B_next,
    input mul_v_alu_next, get_nor_next,
    input logic [3:0] aluc_next,
```

```
    output ID_EX_pipe_t id_ex_pipe
);


    always_ff @(posedge clk, negedge resetn) begin
        if (!resetn)
            id_ex_pipe <= '{IR: '{opcode:NA, default:0}, default:0};
        else
            id_ex_pipe <= '{IR: IR_next,
                            A: A_next,
                            B: B_next,
                            ctrl: '{mul_v_alu: mul_v_alu_next,
                                    get_nor: get_nor_next,
                                    aluc: aluc_next
                                   }
                           };
    end


endmodule: id_ex_pipe
```

3.6 EX_WB_PIPE.SV

The ex_wb_pipe is the pipeline between the Instruction Execute (EX) and Write back (WB) stage. This stage pipelines the following items from the Instruction Execution (EX) stage: -

1. Instruction word (IR) – This is required to write back the output of the execution stage back to the register file.
2. The ALU/MUL output (C) – This is the final result of the instruction execution that is written back to the register file.

```
// ------------------------------------------------------
// VLSI Arch Assignment - 3
// Author: Anand S
// BITSID: 2021HT80003
// ------------------------------------------------------

module ex_wb_pipe
import cpu_pkg::*;
(
    input logic clk, resetn,
    input instruction_t IR_next,
    input sint32_t C_next,

    output EX_WB_pipe_t ex_wb_pipe
);


    always_ff @(posedge clk, negedge resetn) begin
        if (!resetn)
            ex_wb_pipe <= '{IR: '{opcode:NA, default:0}, default:0};
```

```
        else
            ex_wb_pipe <= '{IR:IR_next, C: C_next};
    end

endmodule: ex_wb_pipe
```

3.7 ALU.V,… AND MULTIPLIER.SV,…

As these are taken as is from the specification in Assignment, no change was done to these blocks, and they are being used as is.

3.8 CPU.SV

The cpu module is a structural RTL which instantiates the sub-blocks mentioned above as well as contains the behavioral code for the PC incrementing functionality and the control word decode functionality. If the memories are required to be registered read ports, additional stalling code is added under the macro “RD_STALL_ALLOWED”.

```
// ------------------------------------------------------
// VLSI Arch Assignment - 3
// Author: Anand S
// BITSID: 2021HT80003
// ------------------------------------------------------

module cpu
import cpu_pkg::*;
(
    input   logic   clk, resetn
);
    instr_mem_addr_t    pc;
    IF_ID_pipe_t        if_id_pipe_s;
    ID_EX_pipe_t        id_ex_pipe_s;
    EX_WB_pipe_t        ex_wb_pipe_s;
    sint32_t            A, B, C, ex_out, alu_out;
    logic [24:0]        mult_out;
    instruction_t       instruction_word;
    logic               mul_v_alu;
    logic               carry;
    logic [3:0]         ALUC;
    logic               get_nor;

    // --------------------------
    // PC
    // --------------------------
    always_ff @(posedge clk, negedge resetn) begin
        if (!resetn)        pc <= 0;
        else if (pc >='hc)  pc <= 0;
        else                pc <= pc + 4;
    end

    // --------------------------
    // IF stage
```

```
    // --------------------------
    instr_mem
    u_instr_mem (
                 // Interfaces
                 .raddr                 (pc),
                 .instruction_word      (instruction_word),
                 // Inputs
                 .clk                   (clk),
                 .resetn                (resetn),
                 .re                    (1'b1)); // Always enabled

    if_id_pipe
    u_if_id_pipe (
                  // Interfaces
                  .IR_next              (instruction_word),
                  // Outputs
                  .if_id_pipe           (if_id_pipe_s),
                  // Inputs
                  .clk                  (clk),
                  .resetn               (resetn));

    // --------------------------
    // ID stage
    // --------------------------
    reg_file
    u_reg_file (
                // Interfaces
                .addrA                  (if_id_pipe_s.IR.rs2),
                .addrB                  (if_id_pipe_s.IR.rs1),
                .addrC                  (ex_wb_pipe_s.IR.rd),
`ifdef RD_STALL_ALLOWED
                .C                      (ex_wb_pipe_ss.C),
`else
                .C                      (ex_wb_pipe_s.C),
`endif
                .A                      (A),
                .B                      (B),
                // Inputs
                .clk                    (clk),
                .resetn                 (resetn),
                .enA                    (1'b1),
                .enB                    (1'b1),
                .enC                    (1'b1));

`ifdef RD_STALL_ALLOWED
    // Stall the controls before reaching the pipeline
    always_ff @(posedge clk, negedge resetn) begin:aluc_dec
        if (!resetn) begin
            get_nor   <= 0;
            mul_v_alu <= 0;
            ALUC      <= 4'bxxxx;
        end  else begin
            get_nor   <= 0;
            mul_v_alu <= 0;
            ALUC      <= 4'bxxxx;
            case (if_id_pipe_s.IR.opcode)
                MUL: begin
```

```
                    mul_v_alu <= 1;
                end
                SHIFT: begin
                    ALUC <= 4'b0011; // Assuming SHIFT means sll
                end
                XOR: begin
                    ALUC <= 4'b0010;
                end
                NOR: begin
                    ALUC <= 4'b0101; // Use OR of ALU and negate
externally
                    get_nor <= 1;
                end
            endcase
        end
    end:aluc_dec
`else
    always_comb begin: aluc_dec
        mul_v_alu = 0;
        ALUC =  4'bxxxx;
        get_nor = 0;
        case (if_id_pipe_s.IR.opcode)
            MUL:    mul_v_alu = 1;
            SHIFT:  ALUC = 4'b0011;
            XOR:    ALUC = 4'b0010;
            NOR: begin
                    ALUC = 4'b0101;
                    get_nor = 1;
            end
        endcase
    end: aluc_dec
`endif

    id_ex_pipe
    u_id_ex_pipe (
                // Interfaces
                .IR_next                (if_id_pipe_s.IR),
                .A_next                 (A),
                .B_next                 (B),
                // Outputs
                .id_ex_pipe             (id_ex_pipe_s),
                // Inputs
                .clk                    (clk),
                .resetn                 (resetn),
                .aluc_next              (ALUC),
                .get_nor_next           (get_nor),
                .mul_v_alu_next         (mul_v_alu));

    // --------------------------
    // EX stage
    // --------------------------
    mult_xy
    u_mult (
            // Outputs
            .O                          (mult_out[MDMR_WD-1:0]),
            // Inputs
            .A                          (id_ex_pipe_s.A[MD_WD-1:0]),
```

```
            .B                          (id_ex_pipe_s.B[MR_WD-1:0]));

    alu
    u_alu (
           // Outputs
           .OUT                         (alu_out[DATA_WDTH-1:0]),
           .CARRY                       (carry),
           // Inputs
           .A                           (id_ex_pipe_s.A[DATA_WDTH-
1:0]),
           .B                           (id_ex_pipe_s.B[DATA_WDTH-
1:0]),
           .ALUC                        (id_ex_pipe_s.ctrl.aluc[3:0]));

    assign ex_out = (id_ex_pipe_s.ctrl.mul_v_alu)?
                        signed'({7'b0, mult_out[24:0]}):
                        ((id_ex_pipe_s.ctrl.get_nor)?
                            ~(signed'(alu_out)):
                            signed'(alu_out)
                        );

    ex_wb_pipe
    u_ex_wb_pipe (
    /*AUTOINST*/
                  // Interfaces
                  .IR_next              (id_ex_pipe_s.IR),
                  .C_next               (ex_out),
                  .ex_wb_pipe           (ex_wb_pipe_s),
                  // Inputs
                  .clk                  (clk),
                  .resetn               (resetn));

    // --------------------------
    // WB stage
    // --------------------------
`ifdef RD_STALL_ALLOWED
    EX_WB_pipe_t ex_wb_pipe_ss; // stalled version for timing
    always_ff @(posedge clk, negedge resetn) begin
        if (!resetn) begin
            ex_wb_pipe_ss <= '{IR: '{opcode:NA, default:0}, default:0};
        end else begin
            ex_wb_pipe_ss <= ex_wb_pipe_s;
        end
    end
`endif
endmodule: cpu
```

3.9 FLIST.V

The flist.v file consist of the include statements for all the files that need to be compiled. This consists of all the ALU and Multiplier files from previous assignments. As in assignment 2, the ALU has the additional capability to use Adder based on a single stage carry look-ahead adder for all 32 bits, or 8 stage, 4 bit carry look-ahead adder.

```
// ------------------------------------------------------------
// VLSI Arch Assignment - 2
// Author: Anand S
// BITSID: 2021HT80003
// ------------------------------------------------------------
// file list include
// ------------------------------------------------------------

`ifndef __INCLUDE_FILES__
`define __INCLUDE_FILES__
    // ALU files
    `include "alu_lib.v"
    `ifdef ALU_4bit_CLA
        `include "cla_4bit.v"
        `include "cla_top.v"
    `else
        `include "cla.v"
    `endif
    `include "mux2x1.v"
    `include "mux4x1.v"
    `include "shifter.v"
    `include "alu.v"
    // MUL files
    `include "mult16x9behav.sv"
    `include "ppg.sv"
    `include "fulladder.sv"
    `include "csa.sv"
    `include "cpa.sv"
    `include "mult_xy.sv"
    // CPU files
    `include "cpu_pkg.sv"
    `include "ex_wb_pipe.sv"
    `include "id_ex_pipe.sv"
    `include "if_id_pipe.sv"
    `include "instr_mem.sv"
    `include "reg_file.sv"
    `include "cpu.sv"
`endif // __INCLUDE_FILES__
```

3.10 TESTBENCH: TOP.SV

The testbench is a basic top-level file which has a clock generator and a reset assertion sequence. The behaviour of the CPU is tested only through waveform reviews.

```
// ------------------------------------------------------------
// VLSI Arch Assignment - 3
// Author: Anand S
// BITSID: 2021HT80003
// ------------------------------------------------------------

`include "flist.v"
module top;
    logic clk;
```

```
    logic resetn;

    // ---------------------
    // DUT
    // ---------------------
    cpu
    u_cpu (
    /*AUTOINST*/
           // Inputs
           .clk                          (clk),
           .resetn                       (resetn));

    initial begin
        clk <= 0;
        forever #5 clk = ~ clk;
    end

    initial begin: seq
        resetn = 0;
        #10;
        resetn = 1;
    end: seq
    initial begin: finish
        #200;
        $finish;
    end: finish
endmodule: top
```

The below Makefile was used to compile and run simulation:-

```
BSUB = bsub -Ip
IRUN = $(BSUB) irun
GUI  = 1
ifeq ($(GUI),1)
    RUNOPT += -gui
endif
RUNOPT += -access +rwc

all:
        $(IRUN) top.sv $(RUNOPT)
```

4. Simulation

The CPU design was simulated using cadence incisive "irun" command (version 15.20-s042)

Design needs IEEE 1800-2009 std to be enabled. It was only tested on a UNIX environment.

Simulation was done with default timescale of 1ns/100ps

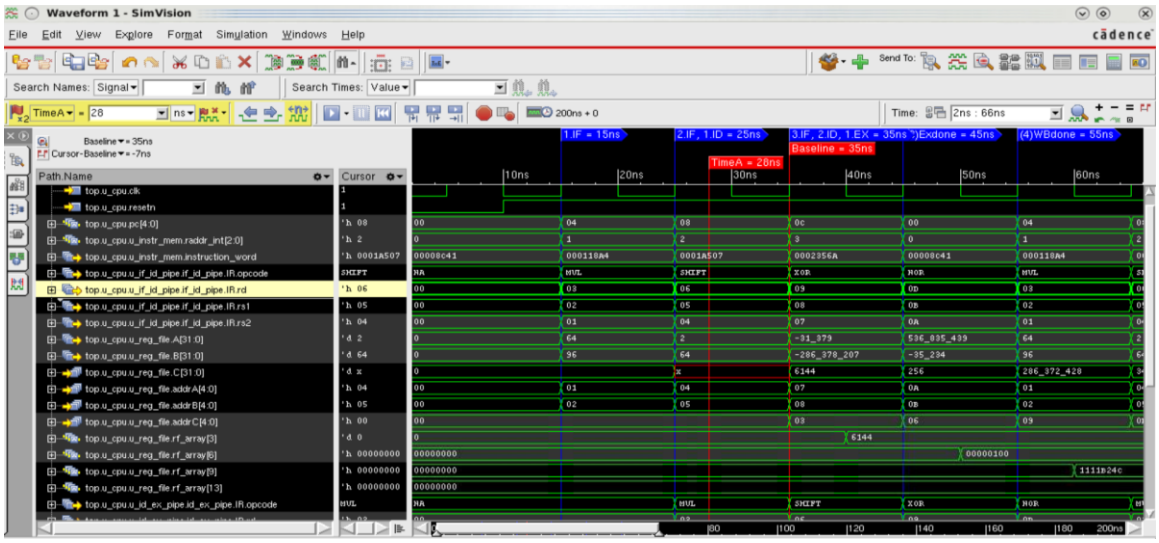
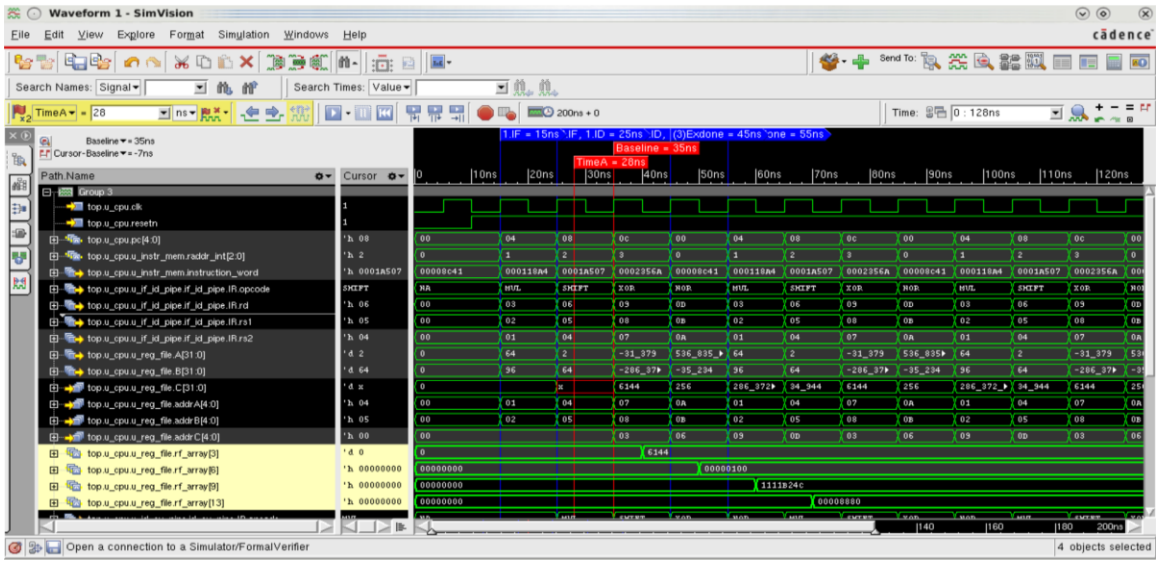


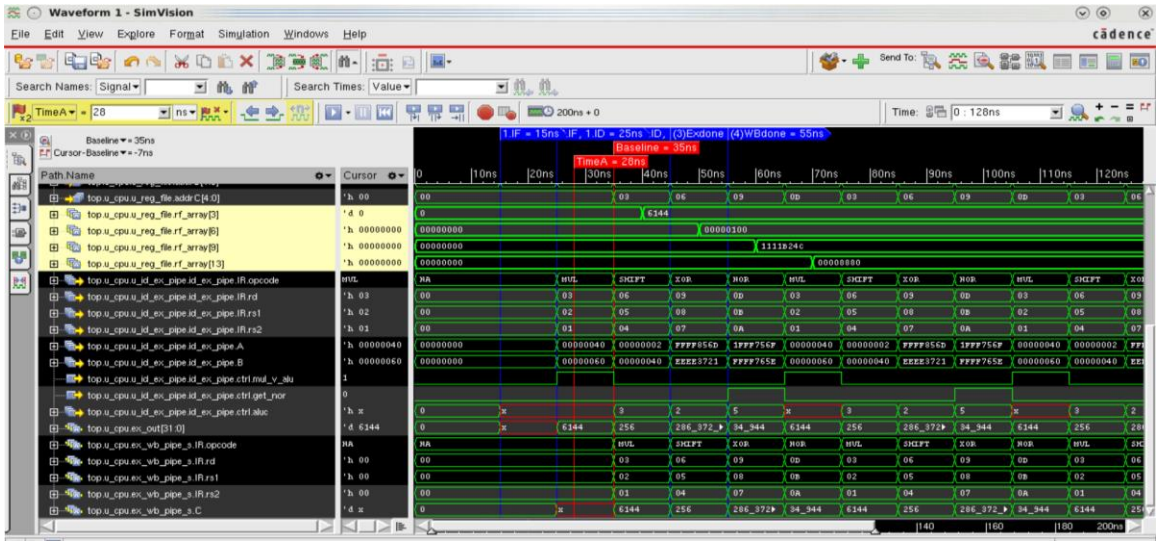
Figure 2 Simulation result of CPU.

As can be seen in Figure 2 Simulation result of CPU. The result of ALU/MUL units in the execution stage are written back in the next half-cycle during the write back stage.

The CPU is made to loop back the PC to 0 after pc == `h0C. Thus, the CPU keeps executing the same instruction until the end of simulation as shown: -



Below are some additional signals which shows the values of the fields in each pipeline : -



5. Conclusion

The CPU design, ALU using CLA, Barrel-shifter and multiplexor components was designed and simulated and found to match with specification and was verified to work using Cadence Incisive suit of simulation tools with the help of IEEE 1800-2009 SystemVerilog assertion and coding constructs.