Design and Implementation of 32 bit RISC microprocessor by pipelining method

Bachelor of Technology

Electronics and Communication Enginnering

Course - Hardware Description Language [HDL]

By

Ritvik Thakur - BT18ECE018

Shashvat Vats - BT18ECE033

Mranal Shroff - BT18ECE034

Guided By

Assistant Prof. Vipin Kamble

Department of Electronics and Communication Engineering

Indian Institute of Information Technology, Nagpur

May, 2020

Acknowledgements

We would like to thank **Assistant Prof. Vipin Kamble** for project guidance and constant support. All the meetings within our group were highly useful in understanding the topic in depth.

Contents

Chapter 1

Abstract

These RISC or Reduced Instruction Set Computer is a design philosophy that has become a mainstream in Scientific and engineering applications. A set of instructions (program) or group of programmes (software) are written to the microprocessor, to perform the task and compute the output. A physical set of hardware modules accomplish the said purpose. he primarily used such modules are the Arithmetic Logic Unit (ALU), Control Unit, Registers and Instruction Execution Unit.The design of an efficient hardware architecture involves the capability to operate with maximum performance even while consuming lower power and reduced silicon area.

The main objective of this paper is to design and implement a 32 – bit RISC (Reduced Instruction Set Computer) processor using XILINX Tool for embedded and portable applications. The design will help to improve the speed of the processor, and to give the higher performance of the processor. The most important feature of the RISC processor is that this processor is very simple and supports load/store architecture. We mainly focus on developing processor using Verilog programming language along with method of pipelining.

Chapter 2

Background and Related Work

2.1 Architecture

The logic circuit for the microprocessor can be divided into two parts: the datapath and the control unit, as shown in Figure 2.1. Figure 2.2 shows the details inside the control unit and the datapath. The datapath is responsible for the actual execution of all data operations performed by the microprocessor, such as the addition of two numbers inside the arithmetic logic unit (ALU). The datapath also includes registers for the temporary storage of your data. The

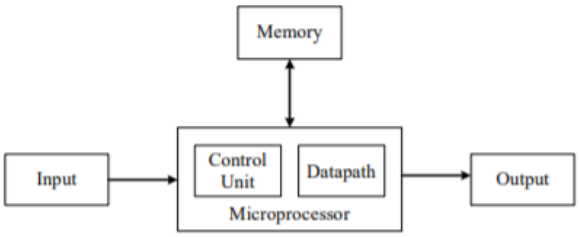

Figure 2.1: Von Neumann model of computer

functional units inside the datapath, which in our example includes the ALU and the register, are connected together with multiplexers and data signal lines. The data signal lines are for transferring data between two functional units. Data signal lines in the circuit diagram are represented by lines connecting two functional units. Sometimes, several data signal lines are grouped together to form a bus. In the sample circuit, a 2-to-1 multiplexer is used to select between the input data and the constant '0' to go to the left operand of the ALU. The output of the ALU is connected to the input of the register.

Even though the datapath is capable of performing all of the data operations of the microprocessor, it cannot, however, do it on its own. In order for the datapath to execute the operations automatically, the control unit is

required. The control unit, also known as the controller, controls all of the operations of the datapath, and therefore, the operations of the entire microprocessor. The control unit is a finite state machine (FSM) because it is a machine that executes by going from one state to another and that there are only a finite number of states for the machine to go to. The control unit is made up of three parts: the next-state logic, the state memory, and the output logic. The purpose of the state memory is to remember the current state that the FSM is in. The next-state logic is the circuit for determining what the next state should be for the machine. And the output logic is the circuit for generating the actual control signals for controlling the datapath.

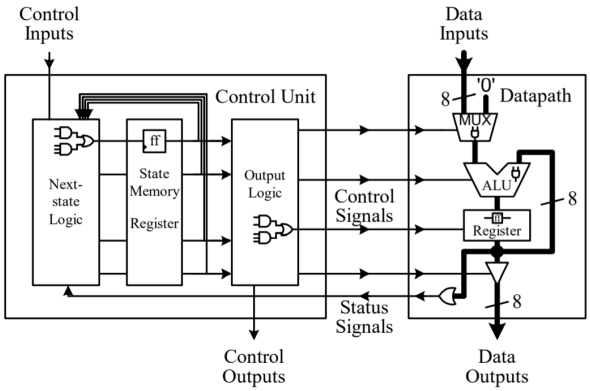

Figure 2.2: Internal architecture of microprocessor

2.2 Instruction Set Architecture

The instruction set, also called ISA (instruction set architecture), is part of a computer that pertains to programming, which is basically machine language. The instruction set provides commands to the processor, to tell it what it needs to do. The instruction set consists of addressing modes, instructions, native data types, registers, memory architecture, interrupt, and exception handling, and external I/O.

2.2.1 Registers and Addressing Modes

32, 32-bit general purpose registers(GPRs) R0 to R31. Register R0 contains a constant 0; cannot be written. A special purpose 32-bit program counter(PC). It points to the next instruction in memory to be fetched and executed. No flag registers i.e., zero, carry, sign, etc.

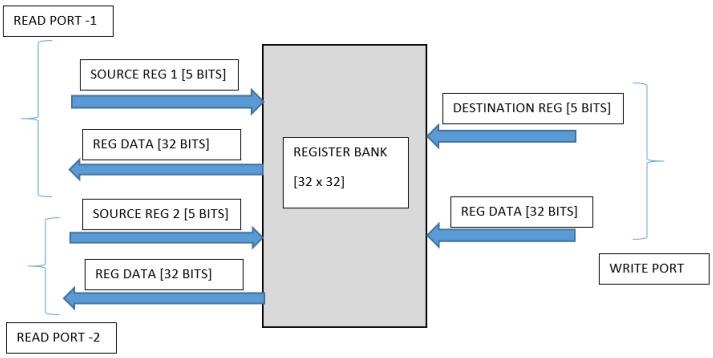

Figure 2.3: Internal Register Bank

Addressing Modes:

Register Mode : In register addressing the operand is placed in one of 8 bit or 16 bit general purpose registers. The data is in the register that is specified by the instruction. Here one register reference is required to access the data.

Immediate Mode : In this mode data is present in the address field of instruction Designed like one address instruction format. Limitation in the immediate mode is that the range of constants are restricted by size of address field.

Register Indexed : In this addressing the operand's offset is placed in any one of the registers as specified in the instruction. The effective address of the data is in the base register or an index register that is specified by the instruction.Here two register references are required to access the data.

2.2.2 Instruction Encoding

2.2.2.1 R-type Instruction Encoding

- Have op(opcode) : (6 bits)
- Here an instruction can use up to three register operands.(Two source and one destination).
- rs: 1st register operand (register source) (5 bits)
- rt: 2nd register operand (5 bits)

- rd: register destination (5 bits)
- shamt: shift amount (0 when N/A) (5 bits)
- funct: function code (identifies the specific R-format instruction) (6 bits)

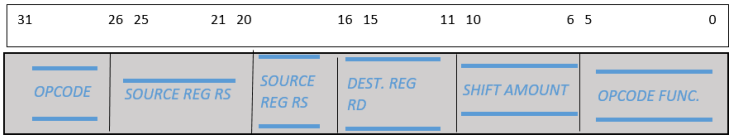

Figure 2.4: R-type instruction

2.2.2.2 I-type Instruction Encoding

- Have a constant value immediately present in the instruction.
- Contains a 16-bit immediate data field.
- Supports one source and one destination register.
- rs: register containing base address (5 bits)
- rt: register destination/source (5 bits)
- immediate: value or offset (16 bits)

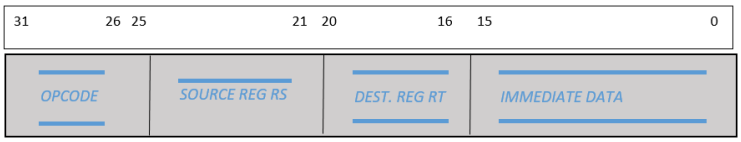

Figure 2.5: I-type instruction

2.2.2.3 J-type Instruction Encoding

- Contains a 26-bit jump address field.
- Extended to 28 bits by padding two 0's on the right.

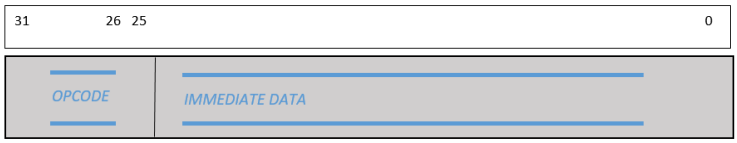

Figure 2.6: J-type instruction

2.2.2.4 Addressing Modes

- Register Addressing : Eg: ADD R1, R2, R3
- Immediate Addressing : Eg: ADDI R1, R2, 200
- Base Addressing : Content of a register is added to a base value to get the
- operand address. Eg : LW R5, 150(R7)
- PC relative addressing : 16 bit offset is added to get the target address. Eg : BEQZ R3, Label.
- Pseudo-direct Addressing : 26 bit offset is added to PC to get the target address.

2.3 Instruction Cycle

Instruction Cycle consist of five steps:

- IF : Instruction fetch
- ID : Instruction Decode / Register Fetch
- EX : Execution / effective Address Calculation
- MEM : Memory access / Branch Completion
- WB : Register write-back.

2.3.1 Instruction Fetch

- Here the instruction pointed to by PC is fetched from memory, and also the next value of PC is computed.
- Every MIPS32 instruction is of 32 bits.
- Every memory word is of 32 bits and has a unique address.

- For a branch instruction, a new value of the PC may be the target address. So the PC is not updated at this stage; new value is stored in a register NPC.

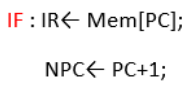

Figure 2.7: Instruction Fetch

2.3.2 Instruction Decode/Register Fetch

- The instruction already fetched in IR is decoded.
- Opcode is 6-bits (bits 31:26).
- First source operand rs (bits 25:21), second source operand rt (bits 20:16)
- 16 bit immediate data (bits 15:0)
- 26 bit immediate data (bits 25:0)
- Decoding is done in parallel with reading the register operands rs and rt. Possible because these fields are in a fixed location in the instruction format.
- In a similar way, the immediate data are sign-extended.

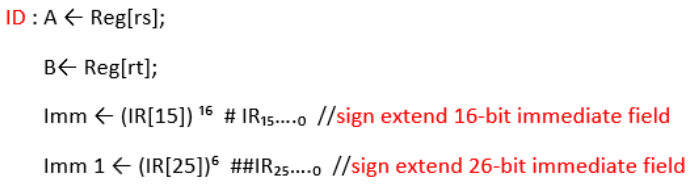

Figure 2.8: Instruction Decode

2.3.3 Execution/Effective Address Calculation

- In this step, the ALU is used to perform some calculation.
- The exact operation depends on the instruction that is already decoded.
- The ALU operates on operands that have been already made ready in the previous cycle

Memory Reference: ALUOut <- A +Imm;

Register-Register ALU Instruction: ALUOut <- A func B;

Register-Immediate ALU Instruction: ALUOut <- A func Imm;

Branch: ALUOut <- NPC + Imm; and cond <- (A op 0);

Figure 2.9: Execution/Address Calculation

2.3.4 Memory Address/Branch Completion

- The only instructions that make use of this step are loads, stores, and branches.
- The load and store instructions access the memory.
- The branch instruction updates PC depending upon the outcome of the branch condition.

Load Instruction: PC <- NPC; LMD <- Mem[ALUOut];

Store Instruction: PC <- NPC; Mem[ALUOut] <- B;

Branch Instruction: if(cond) PC <- ALUOut; else PC <- NPC;

Figure 2.10: Memory Address/Branch Completion

2.3.5 Register Write Back

- In this step, the result is written back into the register file.
- Result may come from the ALU.
- Result may come from the memory system (viz. A LOAD instruction).

- The position of the destination register in the instruction depends on the instruction.

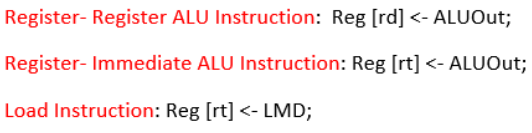

Figure 2.11: Register Write Back

Chapter 3

Concept of Pipelining

Pipelining is a mechanism for overlapped execution of several input sets by partitioning some computation into a set of k sub-computations (or stages). In most of the cases we create a pipeline by dividing a complex operation into simpler operations. We can also say that instead of taking a bulk thing and processing it at once, we break it into smaller pieces and process it one after another.

3.1 Design Of a Basic Pipeline

Basic requirements for pipelining are:-

- We should be able to start a new instruction every clock cycle.
- Each of the five stages(IF,ID,EX,MEM,WB) becomes a pipeline stage.
- Each stage must finish its execution within one cycle.

3.2 Execution of Instructions in Pipelined Processor

Convention used:-

- Most of the temporary registers required in data path are included as part of the inter-stage latches.
- IF ID : - denotes the latch stage between the IF and ID stages.
- ID EX : - denotes the latch stage between the ID and EX stages.
- EX MEM :- denotes the latch stage between the EX and MEM stages.

- MEM WB:- denotes the latch stage between the MEM and WB stages.

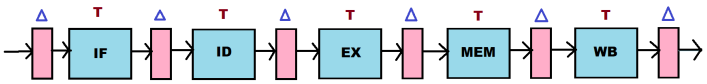

Figure 3.1: Pipeline of Instructions with Latches

- Microinstructions[microoperations] for Pipeline Stage IF : -

```
IF_ID _IR <- Mem [PC];
IF_ID_NPC, PC <- [ if ( EX_MEM_IR[opcode] == branch) & (EX_MEM_cond)]
                    { EX_MEM_ALUOut}
                    Else
                    { PC +1 } ;
```

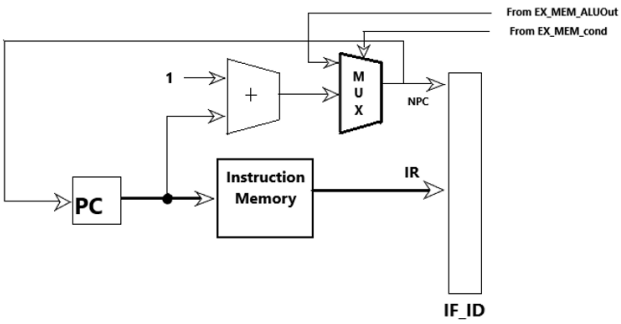

Figure 3.2: Pipeline Stage IF

- Microinstructions[microoperations] for Pipeline Stage ID : -

```
ID_EX_A <- Reg [IF_ID_IR [rs] ];
ID_EX_B <- Reg [IF_ID_IR [rt] ];
ID_EX_NPC <- IF_ID_NPC;
ID_EX_IR <- IF_ ID_IR;
ID_EX_Imm <- sign extend [ IF_ID_IR{15.....0} ] ;
```

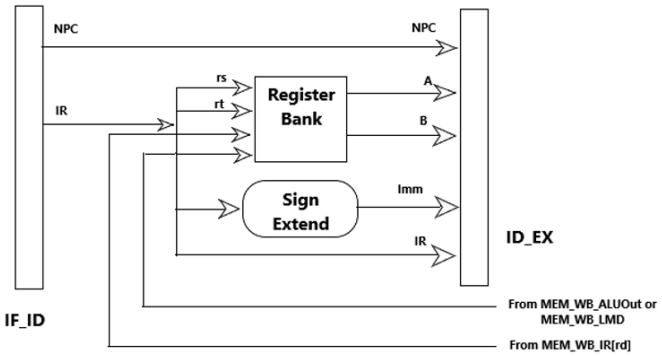

Figure 3.3: Pipeline Stage ID

- Microinstructions[microoperations] for Pipeline Stage EX : -

R – R – ALU :-

EX_MEM_IR <- ID_EX_IR;

EX_MEM_ALUOut <- ID_EX_A func ID_EX_B;

R-M ALU:-

EX_MEM_IR <- ID_EX_IR;

EX_MEM_ALUOut <- ID_EX_A func ID_EX_IMM;

Load/Store:-

EX_MEM_IR <- ID_EX_IR;

EX_MEM_ALUOut <- ID_EX_A + ID_EX_IMM;

EX_MEM_B <- ID_EX_B;

Branch:-

EX_MEM_ALUOut <- ID_EX_NPC + ID_EX_IMM;

EX_MEM_cond <- (ID_EX_A == 0);

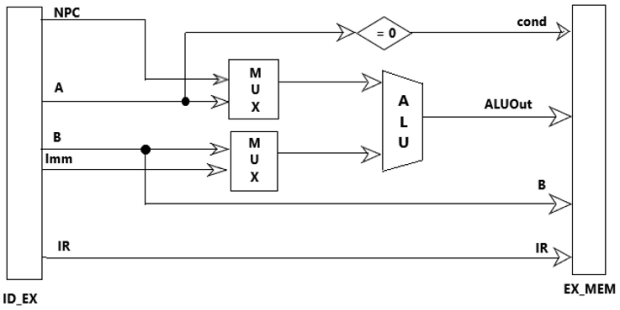

Figure 3.4: Pipeline Stage EX

- Microinstructions[microoperations] for Pipeline Stage MEM : -

ALU Operation

MEM_WB_IR <- EX_MEM_IR;

MEM_WB_ALUOut <- EX_MEM_ALUOut;

Load Operation

MEM_WB_ IR <- EX_MEM_IR;

MEM_WB_LMD <- Mem [EX_MEM_ALUOut];

Store Operation

MEM_WB_IR <- EX_MEM_IR;

MEM [EX_MEM_ALUOut] <- EX_MEM_B;

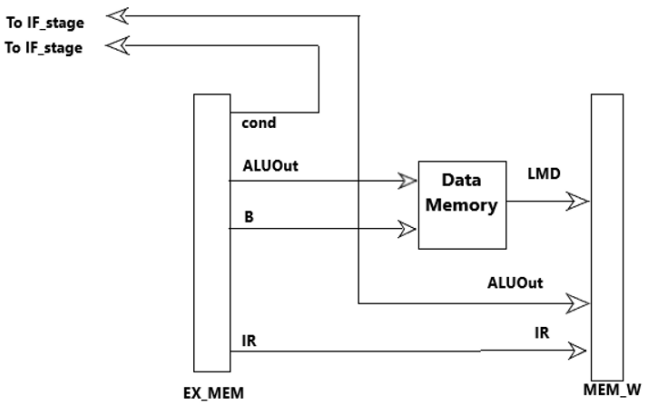

Figure 3.5: Pipeline Stage MEM

- Microinstructions[microoperations] for Pipeline Stage WB : -

R-R ALU: -

Reg [MEM_WB_IR [rd]] <- MEM_WB_ALUOut;

R-M ALU:-

Reg [MEM_WB_IR [rt]] <- MEM_WB_ALUOut;

Load:-

Reg [MEM_WB_IR [rt]] <- MEM_WB_LMD;

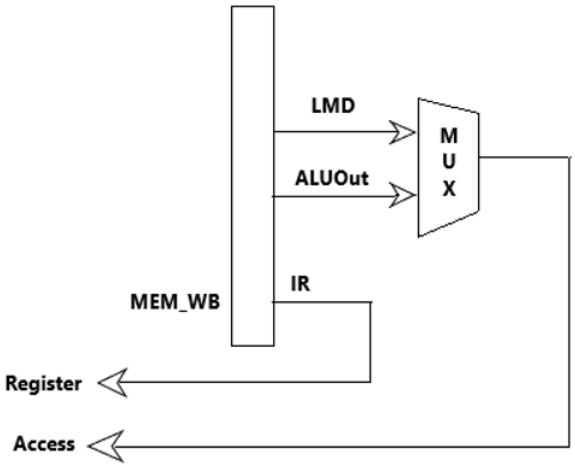

Figure 3.6: Pipeline Stage WB

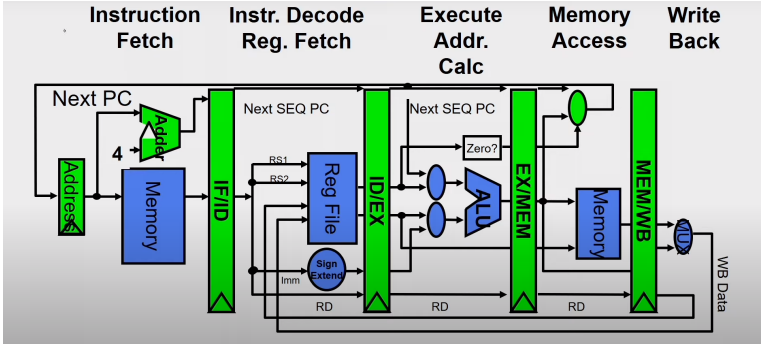

Figure 3.7: Pipelined architecture of RISC Microprocessor

3.3 Performance of Pipelined Processor

- Consider a 'k' segment pipeline with clock cycle time as 'Tp'. Let there be 'n' tasks to be completed in the pipelined processor. Now, the first instruction is going to take 'k' cycles to come out of the pipeline but the other 'n – 1' instructions will take only '1' cycle each, i.e, a total of 'n – 1' cycles. So, time taken to execute 'n' instructions in a pipelined processor: ETpipeline = k + n – 1 cycles = (k + n – 1) Tp
- In the same case, for a non-pipelined processor, execution time of 'n' instructions will be: ETnon-pipeline = n * k * Tp
- So, speedup (S) of the pipelined processor over non-pipelined processor, when 'n' tasks are executed on the same processor is: S = Performance of pipelined processor / Performance of Non-pipelined processor
- As the performance of a processor is inversely proportional to the execution time, we have, S = ETnon-pipeline / ETpipeline S = [n * k * Tp] / [(k + n – 1) * Tp] S = [n * k] / [k + n – 1]
- When the number of tasks 'n' are significantly larger than k, that is, n ¿¿ k S = n * k / n S = k

  where 'k' are the number of stages in the pipeline. Also, Efficiency = Given speed up / Max speed up = S / Smax We know that, Smax = k So, Efficiency = S / k
- Throughput = Number of instructions / Total time to complete the instructions So, Throughput = n / (k + n – 1) * Tp

Chapter 4

Pipeline Implementation of Processor

4.1 Verilog Code

```
module MIPS_32(clk1,clk2);

input clk1,clk2; //2 phase clock
reg[31:0] PC, IF_ID_IR, IF_ID_NPC;
reg[31:0] ID_EX_IR, ID_EX_NPC, ID_EX_A, ID_EX_B, ID_EX_Imm;
reg[2:0] ID_EX_type, EX_MEM_type, MEM_WB_type;
reg[31:0] EX_MEM_IR, EX_MEM_ALUOut, EX_MEM_B;
reg EX_MEM_cond;
reg[31:0] MEM_WB_IR, MEM_WB_ALUOut, MEM_WB_LMD;

reg [31:0] Reg[0:31];     //32x32 register
reg [31:0] Mem[0:1023];    //Memory Size

parameter ADD=6'b000000, SUB=6'b000001, AND=6'b000010,
                                    OR=6'b000011, SLT=6'b000100, MUL=6'b000101,
                                    HLT=6'b111111, LW=6'b001000, SW=6'b001001,
                                    ADDI=6'b001010, SUBI=6'b001011, SLTI=6'b001100,
                                    BNEQZ=6'b001101, BEQZ=6'b001110;

parameter RR_ALU=3'b000, RM_ALU=3'b001, LOAD=3'b010,
```

```
        STORE=3'b011, BRANCH=3'b100, HALT=3'b101;

reg HALTED;
reg TAKEN_BRANCH;

//Instruction Fetch Stage

always @(posedge clk1)
        if (HALTED ==0)
        begin
                if(((EX_MEM_IR[31:26] == BEQZ) && (EX_MEM_cond == 1))||
                 ((EX_MEM_IR[31:26] == BNEQZ)&& (EX_MEM_cond == 0)))
                        begin
                                IF_ID_IR <= #2 Mem[EX_MEM_ALUOut];
                                TAKEN_BRANCH <= #2 1'b1;
                                IF_ID_NPC <= #2 EX_MEM_ALUOut + 1;
                                PC <=   #2 EX_MEM_ALUOut + 1;
                                end
                        else
                                begin
                                        IF_ID_IR <= #2  Mem[PC];
                                        IF_ID_NPC <= #2 PC + 1;
                                        PC <= #2 PC + 1;
                                        end
                        end

// Instruction Decode
always @(posedge clk2)
 if(HALTED == 0)
 begin
        if(IF_ID_IR[25:21] == 5'b00000)
                ID_EX_A <= 0;
        else
                ID_EX_A <= #2 Reg[IF_ID_IR[25:21]]; //"rs"
```

```
        if(IF_ID_IR[20:16] == 5'b00000)
                ID_EX_B <= 0;
        else
                ID_EX_B <= #2 Reg[IF_ID_IR[20:16]];   //"rt"

        ID_EX_NPC <= #2 IF_ID_NPC;
        ID_EX_IR <= #2 IF_ID_IR;
        ID_EX_Imm <= #2 {{16{IF_ID_IR[15]}} , {IF_ID_IR[15:0]}};


        case (IF_ID_IR[31:26])
                ADD,SUB,AND,OR,SLT,MUL : ID_EX_type <= #2 RR_ALU;
                ADDI, SUBI, SLTI: ID_EX_type <= #2 RM_ALU;
                LW: ID_EX_type <=#2 LOAD;
                SW: ID_EX_type <= #2 STORE;
                BNEQZ,BEQZ: ID_EX_type <= #2 BRANCH;
                HLT: ID_EX_type <= #2 HALT;
                default: ID_EX_type <= #2 HALT;

                endcase
        end

//Execution Stage

always @(posedge clk1)
        if(HALTED == 0)
        begin
                EX_MEM_type <= ID_EX_type;
                EX_MEM_IR <= ID_EX_IR;
                TAKEN_BRANCH <= 0;

                case (ID_EX_type)
                        RR_ALU: begin
```

```
            case (ID_EX_IR[31:26])
            ADD: EX_MEM_ALUOut <= #2 ID_EX_A + ID_EX_B;
            SUB: EX_MEM_ALUOut <= #2 ID_EX_A - ID_EX_B;
            AND: EX_MEM_ALUOut <= #2 ID_EX_A & ID_EX_B;
            OR:  EX_MEM_ALUOut <= #2 ID_EX_A | ID_EX_B;
            SLT: EX_MEM_ALUOut <= #2 ID_EX_A < ID_EX_B;
            MUL: EX_MEM_ALUOut <= #2 ID_EX_A * ID_EX_B;
            default: EX_MEM_ALUOut <= #2 32'hxxxxxxxx;
            endcase
            end

RM_ALU:     begin
            case (ID_EX_IR[31:26])
            ADDI: EX_MEM_ALUOut <= #2 ID_EX_A + ID_EX_Imm;
            SUBI: EX_MEM_ALUOut <= #2 ID_EX_A - ID_EX_Imm;
            SLTI: EX_MEM_ALUOut <= #2 ID_EX_A < ID_EX_Imm;
            default: EX_MEM_ALUOut <= #2 32'hxxxxxxxx;
            endcase
            end

LOAD, STORE:
            begin
            EX_MEM_ALUOut <= #2 ID_EX_A + ID_EX_Imm;
            EX_MEM_B <= #2 ID_EX_B;
            end

BRANCH:     begin
            EX_MEM_ALUOut <= #2 ID_EX_NPC + ID_EX_Imm;
            EX_MEM_cond <= #2 (ID_EX_A == 0);

            end
            endcase
            end
```

```
//MEMORY STAGE

always @(posedge clk2)
        if(HALTED == 0)
        begin
                MEM_WB_type <=#2 EX_MEM_type;
                MEM_WB_IR <= #2 EX_MEM_IR;

                case(EX_MEM_type)
                        RR_ALU, RM_ALU:
                        MEM_WB_ALUOut <= #2 EX_MEM_ALUOut;

                        LOAD:
                         MEM_WB_LMD  <= #2  Mem[EX_MEM_ALUOut];

                        STORE:     if(TAKEN_BRANCH == 0)
                        Mem[EX_MEM_ALUOut] <= #2 EX_MEM_B;
                endcase
        end

//WRITE BACK

always @(posedge clk1)

        begin
                if(TAKEN_BRANCH == 0)

                case (MEM_WB_type)

                RR_ALU : Reg[MEM_WB_IR[15:11]] <= #2 MEM_WB_ALUOut;

                RM_ALU : Reg[MEM_WB_IR[20:16]] <= #2 MEM_WB_ALUOut;
```

```
                LOAD : Reg[MEM_WB_IR[20:16]] <= #2 MEM_WB_LMD;

                HALT : HALTED <= #2 1'b1;

                endcase
        end
endmodule
//End Of Code...
```

- In our code we have parametrized opcodes of various microoperations.
- ADD :- 6'b000000
- SUB :- 6'b000001
- AND :- 6'b000010
- OR :- 6'b000011
- SLT :- 6'b000100
- MUL :- 6'b000101
- HLT :- 6'b111111
- LW :- 6'b001000
- SW :- 6'b001001
- ADDI :- 6'b001010
- SUBI :- 6'b001011
- SLTI :- 6'b001100
- BNEQZ :- 6'b001101
- BEQZ :- 6'b001110

4.2 TestBench

- Test Benches can be written for verifying the operation of the processor model.
- Load a program from a specific memory address (say, 0);
- Initialize PC with starting address of the program.
- The program starts executing and will continue to do so until the HLT instruction is encountered.
- We can print the results from the memory locations or registers to verify the operation.

We are performing following example for the verification of our code:-

- Add three numbers 10,20 and 30 stored in processor registers.
- Initialize R1 with 10.
- Initialize R2 with 20.
- Initialize R3 with 25.
- Add the three numbers and store the sum in R4,R5.

ASSEMBLY LANGUAGE PROGRAM	MACHINE CODE [IN BINARY]
ADDI R1, R0, 10	001010 00000 00001 0000000000001010
ADDI R2, R0, 20	001010 00000 00010 0000000000010100
ADDI R3, R0, 25	001010 00000 00011 0000000000011001
ADD R4, R1, R2	000000 00001 00010 00100 00000 000000
ADD R5, R4, R3	000000 00100 00011 00101 00000 000000
HLT	111111 00000 00000 00000 00000 000000

Figure 4.1: Machine Code Of Instructions

```
‘timescale 1ns / 1ps
module test;

        // Inputs
```

```
reg clk1;
reg clk2;
integer k;

// Instantiate the Unit Under Test (UUT)
MIPS_32 mips (clk1 , clk2);

initial begin
        // Initialize Inputs
        clk1 = 0;
        clk2 = 0;

        repeat(20)     //Generating Two Phase Clock...
                begin
                 #5 clk1 = 1;
                 #5 clk1 = 0;
                 #5 clk2 = 1;
                 #5 clk2 = 0;
                end
        end

        initial
                begin
                        for (k=0; k<31; k=k+1)
                                mips.Reg[k] = k;

                        mips.Mem[0] = 32'h2801000a; // ADDI R1,R0,10
                        mips.Mem[1] = 32'h28020014; // ADDI R2,R0,20
                        mips.Mem[2] = 32'h28030019; // ADDI R3,R0,25
                        mips.Mem[3] = 32'h0ce77800; // OR R7,R7,R7--dummy instruc.
                        mips.Mem[4] = 32'h0ce77800; // OR R7,R7,R7--dummy instruc.
                        mips.Mem[5] = 32'h00222000; // ADD R4,R1,R2
                        mips.Mem[6] = 32'h0ce77800; // OR R7,R7,R7--dummy instruc.
                        mips.Mem[7] = 32'h00832800; // ADD R5,R4,R3
```

```
            mips.Mem[8] = 32'hfc000000;// HLT

            mips.HALTED = 0;
            mips.PC = 0;
            mips.TAKEN_BRANCH = 0;

        #280;

        for(k=0; k<6; k=k+1)
            $display("R%1d - %2d", k, mips.Reg[k]);
        end

        initial
            begin
                $dumpfile("mips.vcd");
                //$dumpvars(0, test);
                $dumpvars(0,mips);
                #300 $finish;

            end
endmodule
```

- We are using a dummy instruction to consume one cycle [OR R7,R7,R7].

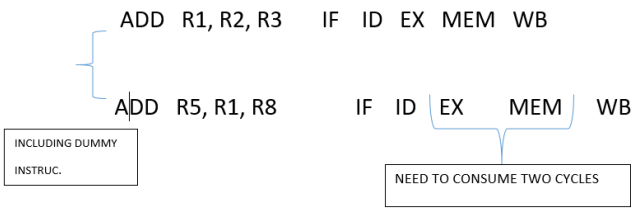

Figure 4.2: Dummy Instructions

4.3 Output with waveforms

We have got simulation output as follows:-

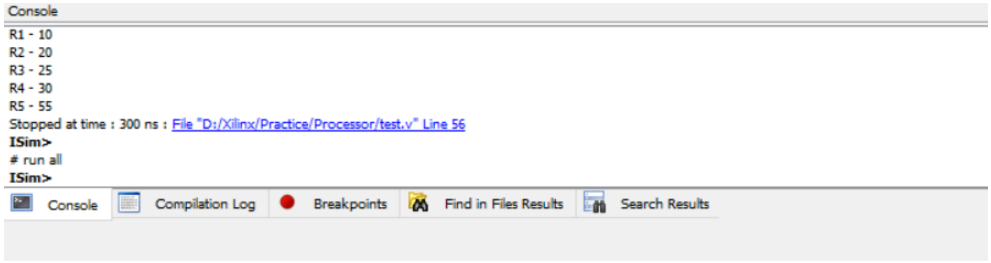

Figure 4.3: Simulation Output

- R1 :- 10
- R2 :- 20
- R3 :- 25
- R4 :- 30
- R5 :- 55

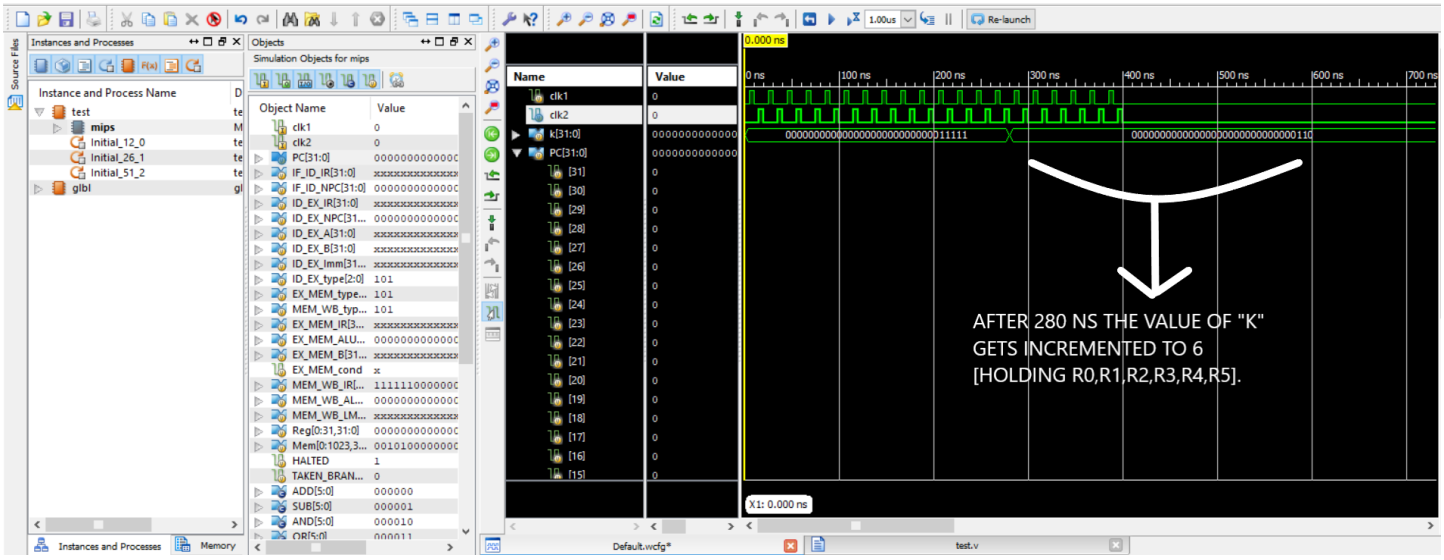

Figure 4.4: Waveforms

Chapter 5

Conclusion and Future Work

- In our project we have observed that process of executing the instructions can be speedup by using the concept of pipelining.
- A more efficient and reliable design can be made by using pipelining to avoid any unwanted changes.
- More complex instructions can be performed by using branch instruction.
- For future scope, many control signals can also be generated for execution of instructions in the datapath.

5.0.1 References

- Design of a 16 Bit RISC Processor - K. Vishnuvardhan Rao*, A. Anita Angeline and V. S. Kanchana Bhaaskaran
- Digital Logic and Microprocessor Design With VHDL - Enoch O. Hwang La Sierra University, Riverside .
- COMPUTER PRINCIPLES AND DESIGN IN VERILOG HDL - Yamin Li Hosei University, Japan
- Processor Design System-on-Chip Computing for ASICs and FPGA - Jari Nurmi Tampere University of Technology, Finland.
- Hardware Modelling using Verilog - NPTEL Course By Prof.Indranil Sengupta,IIT Kharagpur