SMIPS 5 STAGE PIPELINED PROCESSOR

EE 619 : VLSI SYSTEM DESIGN

GROUP MEMBERS

AMISH GOEL(10327071)

NITISH KUMAR SRIVASTAV(10464)

RITURAJ(10327605)

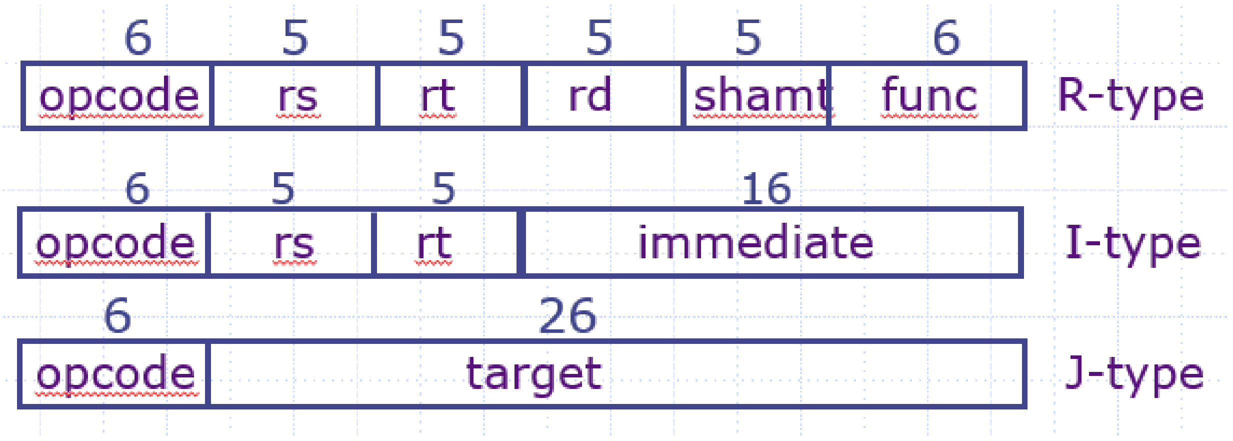
INTRODUCTION

SMIPS is a 32 bit Instruction Set Architecture (ISA) with Integer operations only. In this project we have designed a 5 stage pipelined processor based on SMIPS ISA. The processor is implemented using Harvard Architecture consisting of separate Instruction and Data Memories. The main motivation behind pipelining the processor is to increase the throughput. However, this adds additional complexity to the control unit design that was not present in the non-pipelined version. Our design efficiently handles all possible data and control hazards which arise as a result of pipelining the processor. We have implemented this architecture in Verilog and synthesized it for Virtex-II PRO XILINX FPGA board. FPGA implementation proves the feasibility of the designed processor.

SMIPS ISA

It is the version of the MIPS ISA. SMIPS stands for Simple MIPS since it is a subset of full MIPS ISA. The MIPS architecture is based on RISC (Reduced Instruction Set Computer) philosophy.

All the instructions can be grouped into following three categories.



R Type Instructions:

Computational instruction – rd ← (rs) func (rt)

I Type Instructions:

Computational instruction – rt ← (rs) op immediate

Load Instruction – rt ← Mem [(rs) + immediate]

Store Instruction – Mem [(rs) + immediate] ← (rt)

Control Instruction – Branch target address = immediate + PC + 1

J Type Instructions:

Control Instruction – Jump target address = immediate

Implemented Processor Microarchitecture

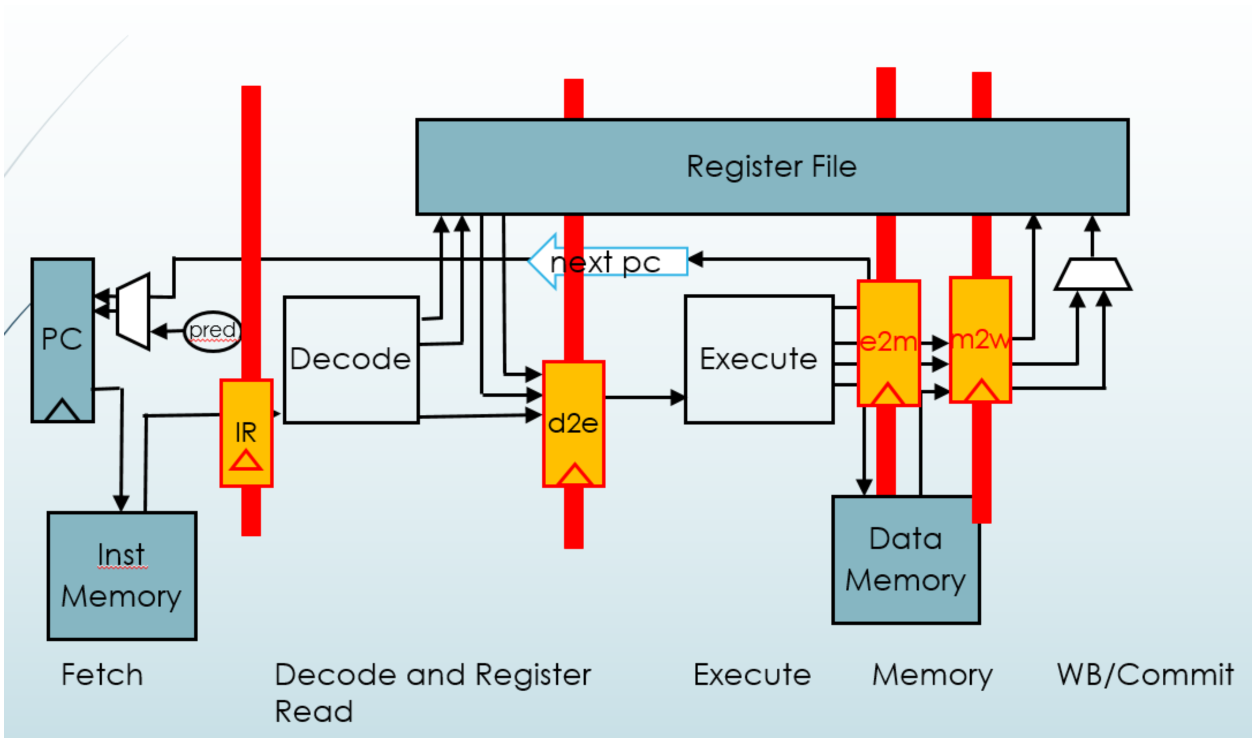


Fig 1- 5 stage pipelined processor

The processor is pipelined into following 5 stages.

- Fetch Stage
- Decode and Register Read Stage
- Execute Stage
- Memory Operation
- Writeback / Commit

FETCH- Fetches instructions from the instruction memory. It also sends the epoch value with each instruction to the decode stage (required to handle mispredictions).

DECODE AND REGISTER READ – Decodes the fetched instructions and reads the value of the source registers for the R and I Type instructions, to be sent for execute stage. In case, an instruction's source register is same as destination register of some other instruction ahead in the pipeline, then it stalls the pipeline until that instruction is committed. This takes care of any potential data hazard.

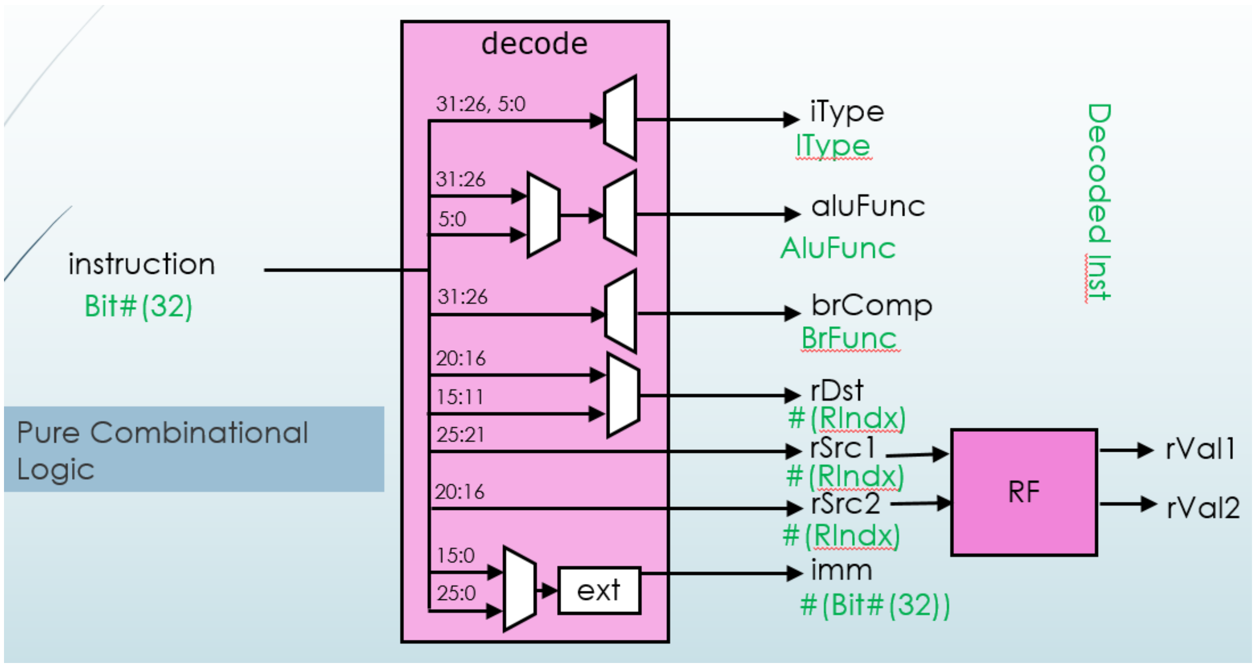


Fig.2- Decode Stage

EXECUTE –It performs the Arithmetic and logical operations and also branch address calculations. It updates the pc value in each cycle. In case of misprediction, it toggles the global epoch register, corrects the PC value and poisons the wrong path instructions in the earlier stages of the pipeline.

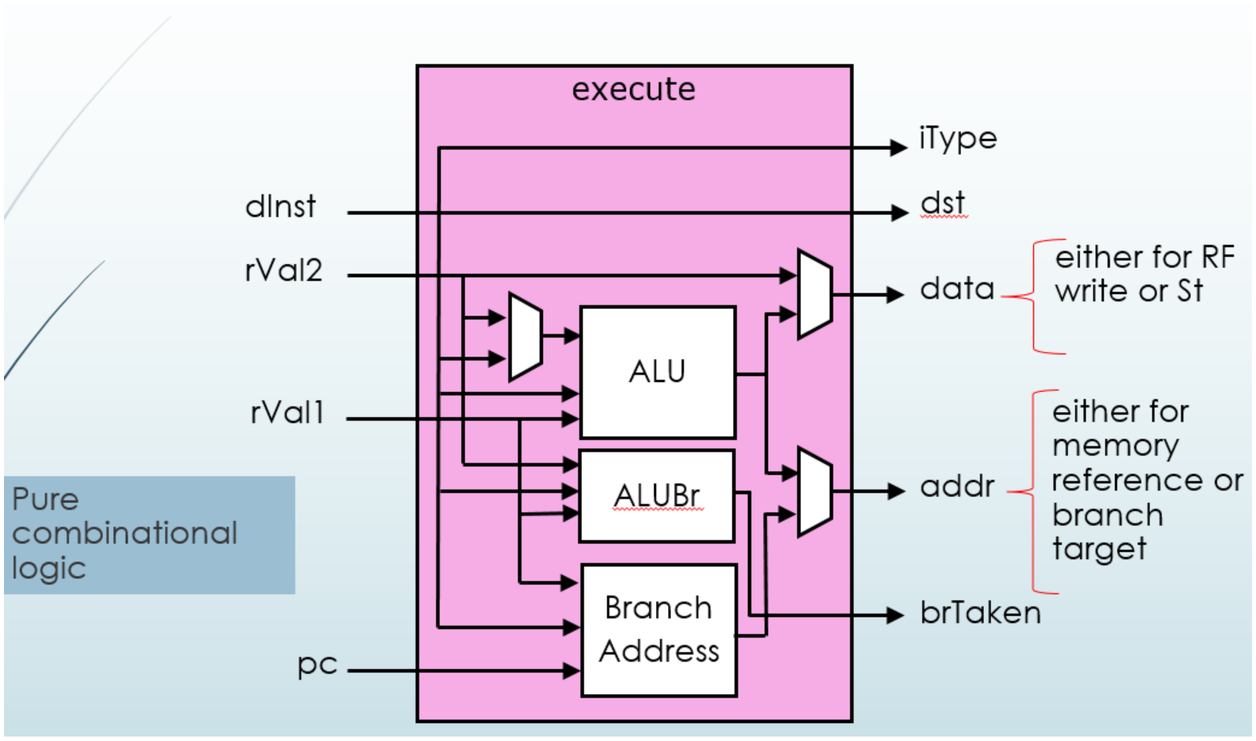


Fig.3- Execute Stage

MEMORY – It performs Load and Store operations on the data memory for load and store instructions and forwards the instruction to writeback stage.

WRITEBACK – It does register writes (if required) and is the last stage of the pipeline.

<u>ARCHITECTURAL ELEMENTS -</u>

Memory –

Instruction and data memories have one read port and one write port. It is implemented as 1024X32 bit memory.

Register File – Has two read ports and one write port. There are 31 general purpose 32-bit registers r1-r31. Register r0 is reserved and cannot be used.

Taking Care of Hazards

1. Control Hazards

Control hazard: Inst_{i+1} is not known until Inst_i is executed(since there can be a branch or jump instruction). So wrong instructions get fetched in the pipeline implementation. These instructions have to be poisoned so that they don't get executed.

To poison the wrong path instruction the following steps have been adopted.

- An epoch register has been added in the processor state. It is a 1 bit register.
- The Execute stage toggles the epoch whenever the pc prediction is wrong and sets the pc to the correct value.
- The Fetch stage associates the current epoch with every instruction when it is fetched.
- The epoch of the instruction is examined when it is ready to execute. If the processor epoch has changed the instruction is thrown away

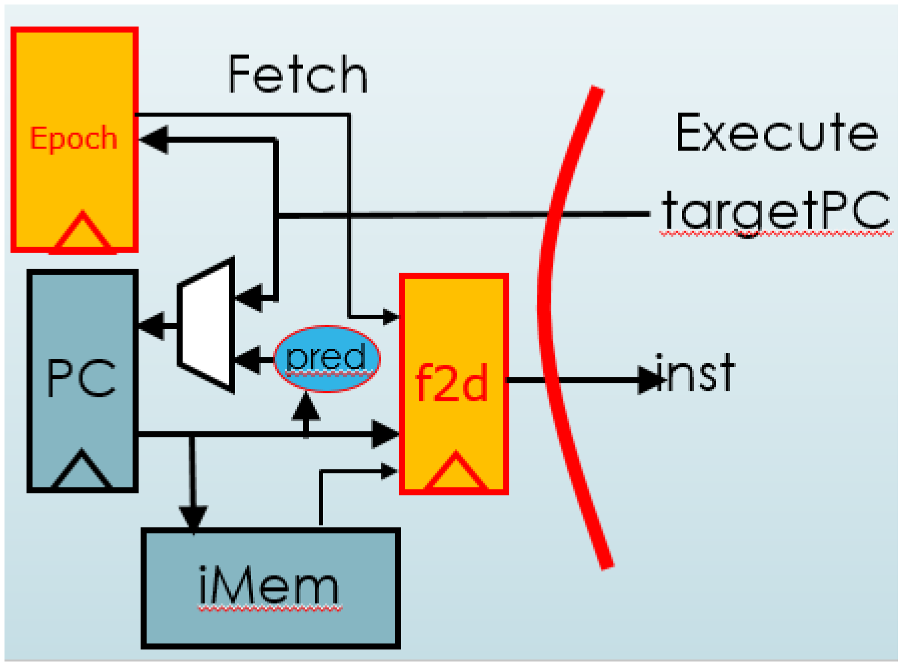


Fig.4- Epoch based solution for Control Hazard

2. **Data Hazard**

- It occurs in the pipeline if the register values to be fetched are stale, i.e., will be modified by some older instruction still in the pipeline. This condition is referred to as a **Read-After-Write** (RAW) hazard.
- Data hazard depends upon the match between the source registers of the fetched instruction and the destination register of an instruction already in the pipeline
- We have implemented a detection logic to **Stall** the Fetch from dispatching the instruction as long as RAW hazard prevails.
- RAW hazard will disappear as the pipeline drains.

RESULTS

The post synthesis result of various architectural elements synthesized are summarized in Table1.

Clock Details:

Maximum Frequency	:	165.223 MHz
Fanout	:	1767
Net Skew	:	0.278

Testbenches used – Array search

– Fibonacci Series Generator

Post Synthesis Result of the processor design in Verilog

HDL Synthesis Report	
Macro Statistics	
# RAMs	2
1024x32-bit dual-port RAM	1
1024x32-bit single-port RAM	1
# ROMs	1
16x4-bit ROM	1
# Adders/Subtractors	4
10-bit adder	2
32-bit adder	1
32-bit subtractor	1
# Registers	54
1-bit register	3
10-bit register	4
3-bit register	4
32-bit register	39
4-bit register	1
5-bit register	3
# Comparators	12
32-bit comparator equal	1
32-bit comparator greatequal	1
32-bit comparator greater	1
32-bit comparator less	1
32-bit comparator lessequal	1
32-bit comparator not equal	1
5-bit comparator equal	6
# Multiplexers	2
32-bit 32-to-1 multiplexer	2
# Logic shifters	2
32-bit shifter logical left	1
32-bit shifter logical right	1
# Xors	2
1-bit xor2	1
32-bit xor2	1

Verilog Codes

Top Level Module :   CPU

```
module cpu(input clk, input reset,input wI_en, input[9:0] wI_addr, input [31:0]wI_data, output reg[31:0]
res_out);

        parameter Ld=3'b 001;
        parameter St=3'b 010;
        parameter Alu=3'b 000;
        parameter NOP=3'b111;
        parameter NotALU=4'b 1111;
        parameter NT=3'b 111;

        reg[31:0] Imem[1023:0],Dmem[1023:0],f2dIR;
        reg[9:0] PC,f2dPC,d2ePC;
        reg[31:0] d2eOp1,d2eOp2,d2e_data,e2mres,m2wbres;
        reg[2:0] d2eitype,e2mitype,m2wbitype;
        reg[4:0] d2erd,e2mrd,m2wbrd;
        reg[3:0] Alufunc;
        reg[2:0] Brfunc;
        reg[9:0] Memaddr;
        reg gepoch;
        reg f2depoch;
        reg d2e_epoch;

        wire[9:0] next_pc;


        wire[31:0] d2eOp1_w,d2eOp2_w,d2e_data_w,e2mres_w;//m2wbres_w;
        wire[2:0] d2eitype_w,e2mitype_w;//m2wbitype_w;
        wire[4:0] d2erd_w,e2mrd_w;//m2wbrd_w;
        wire[3:0] Alufunc_w;
        wire[2:0] Brfunc_w;
        wire[9:0] Memaddr_w;
        reg w_en;
        wire[31:0] w_data;
        wire[4:0] w_addr;
        wire stall,d2e_epoch_w;
        wire mispred;
        wire[9:0] d2ePC_w;

        Decode
myDecode(.clk(clk),.reset(reset),.Inst(f2dIR),.w_en(w_en),.w_data(w_data),.w_addr(w_addr),.d2erd(d2e
rd), .e2mrd(e2mrd), .m2wbrd(m2wbrd),.f2depoch(f2depoch),.f2dPC(f2dPC),

.IType(d2eitype_w),.ALUfunc(Alufunc_w),.Branchfunc(Brfunc_w),.op1(d2eOp1_w),.op2(d2eOp2_w),.des
(d2erd_w),.data(d2e_data_w),.stall(stall),.d2e_epoch(d2e_epoch_w),.d2ePC(d2ePC_w));
```

```
	alu
myAlu(.d2e_itype(d2eitype),.Alufunc(Alufunc),.Brfunc(Brfunc),.Op1(d2eOp1),.Op2(d2eOp2),.pc(PC),.dat
a(d2e_data),.rd(d2erd),.gepoch(gepoch),.d2e_epoch(d2e_epoch),.d2ePC(d2ePC),

	.next_pc(next_pc),.res(e2mres_w),.e2m_itype(e2mitype_w),.Memaddr(Memaddr_w),.rdest(e2
mrd_w),.mispred(mispred));

	assign w_data = m2wbres;
	assign w_addr = m2wbrd;

	always@(*) begin
		if( m2wbitype==Ld || m2wbitype==Alu )
			w_en = 1'b1;
		else
			w_en =1'b0;
	end

	always@(posedge clk) begin

		if(reset)begin
			if(wI_en)begin
				Imem[wI_addr]<=wI_data;
			end

			f2dIR <= 32'h ffffffff;
			PC <= 0;
			gepoch <= 0;
			f2depoch <= 0;
			d2e_epoch <= 0;
			d2eitype <= NOP;
			Alufunc <= NotALU;
			Brfunc <= NT;
			d2eOp1 <= 0;
			d2eOp2 <= 0;
			d2erd <= 0;
			d2e_data <= 0;
			f2dPC <= 0;
			d2ePC <= 0;

			e2mres <= 0;
			e2mitype <= NOP;
			Memaddr<= 0;
			e2mrd <= 0;
			m2wbres <= 0;
			m2wbitype <= NOP;
			m2wbrd <= 0;
```

```
        res_out <= 0;
end


else begin

        if(~stall & ~mispred) begin
                PC <= next_pc;
                f2dIR <= Imem[PC];
                f2depoch <= gepoch;
                f2dPC <= PC;
        end

        if (mispred) begin
                gepoch <= ~gepoch;
                PC <= next_pc;
                f2dIR <= Imem[PC];
                f2depoch <= gepoch;
                f2dPC <= PC;
        end
        d2eitype <= d2eitype_w;
        Alufunc <= Alufunc_w;
        Brfunc <= Brfunc_w;
        d2eOp1 <= d2eOp1_w;
        d2eOp2 <= d2eOp2_w;
        d2erd <= d2erd_w;
        d2e_data <= d2e_data_w;
        d2e_epoch <= d2e_epoch_w;
        d2ePC <= d2ePC_w;

        e2mres <= e2mres_w;
        e2mitype <= e2mitype_w;
        Memaddr<= Memaddr_w;
        e2mrd <= e2mrd_w;

        if (e2mitype==Ld)
                m2wbres <= Dmem[Memaddr];
        else if (e2mitype==St)
                Dmem[Memaddr] <= e2mres;
        else
                m2wbres <= e2mres;

        m2wbitype <= e2mitype;
        m2wbrd <= e2mrd;
end

res_out <= m2wbres;
```

```
        end

endmodule
```

MODULE : DECODE

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
module Decode(input clk, input reset, input f2depoch,input[9:0] f2dPC,
                                input [31:0]Inst,input[4:0] d2erd,input[4:0] e2mrd,input[4:0] m2wbrd,
                                input w_en, input[31:0]  w_data, input[4:0] w_addr,
                                output reg[2:0]IType, output reg[3:0] ALUfunc,output reg[2:0]
Branchfunc,output reg stall,
                                output reg[31:0] op1, output reg[31:0]op2, output reg[4:0]des, output
reg[31:0]data, output d2e_epoch,output[9:0] d2ePC);

        ///// OPcodes ////////
        parameter opLW=             6'b100011;  // LW and St  inst ///
        parameter opSW=             6'b101011;

  parameter opADDIU= 6'b001001; /// I- type instructions //
        parameter opSLTI=           6'b001010;
        parameter opSLTIU=      6'b001011;
        parameter opANDI=           6'b001100;
        parameter opORI=            6'b001101;
        parameter opXORI=           6'b001110;
        parameter opLUI=            6'b001111;

        parameter opJ=              6'b000010; /// J- type instructions //
        parameter opJAL=            6'b000011;
        parameter opJR=             6'b000000;
        parameter opJALR=           6'b000000;
        parameter opBEQ=            6'b000100;
        parameter opBNE=            6'b000101;
        parameter opBLEZ=           6'b000110;
        parameter opBGTZ=           6'b000111;
        parameter opRT=             6'b000001;


        parameter opFUNC=           6'b000000;

        ///////// Functions For R Type ////////
        parameter funcSLL=      6'b000000;
        parameter funcSRL=      6'b000010;
        parameter funcSRA=      6'b000011;
        parameter funcSLLV=     6'b000100;
        parameter funcSRLV=     6'b000110;
```

```
	parameter funcSRAV=	6'b000111;
	parameter funcADDU=	6'b100001;
	parameter funcSUBU=	6'b100011;
	parameter funcAND=	6'b100100;
	parameter funcOR=	6'b100101;
	parameter funcXOR=	6'b100110;
	parameter funcNOR=	6'b100111;
	parameter funcSLT=	6'b101010;
	parameter funcSLTU=	6'b101011;

	parameter funcJR=	6'b001000;
	parameter funcJALR=	6'b001001;

	///// Instruction Type ////
parameter Alu=3'b000;
	parameter Ld=3'b001;
	parameter St=3'b010;
	parameter J=3'b011;
	parameter Jr=3'b100;
	parameter Br=3'b101;
	parameter NOP=3'b111;

	//// Alufuncs ///////
	parameter Add=4'b 0000;
	parameter Sub=4'b 0001;
	parameter And=4'b 0010;
	parameter Or=4'b 0011;
	parameter Xor=4'b 0100;
	parameter Nor=4'b 0110;
	parameter Slt=4'b 0111;
	parameter Sltu=4'b 1000;
	parameter Lshift=4'b 1001;
	parameter Rshift=4'b 1010;
	parameter Sra=4'b 1011;
	parameter NotALU=4'b 1111;

	///// Branch Functions ///
	parameter Eq=3'b 000;
	parameter Neq=3'b 001;
	parameter Le=3'b 010;
	parameter Lt=3'b 011;
	parameter Ge=3'b 100;
	parameter Gt=3'b 101;
	parameter AT=3'b 110;
	parameter NT=3'b 111;

	parameter rtBLTZ=5'b00000;
	parameter rtBGTZ=5'b00100;
	reg[31:0] Rfile[0:31];
	//reg[2:0]IType,Branchfunc;
	//reg[3:0] ALUfunc;
	//reg[31:0]op1,op2,data;
	//reg[4:0]des;
	integer i;

  wire [4:0]rs,rt,rd;
	wire [15:0]imm;
	wire [5:0]funct;
	wire [5:0]shamt;
	wire [5:0]Opcode;

	assign Opcode=Inst[31:26];
  assign rs=Inst[25:21];
	assign rt=Inst[20:16];
	assign rd=Inst[15:11];
	assign funct=Inst[5:0];
	assign shamt=Inst[10:6];
	assign imm=Inst[15:0];
	assign d2e_epoch = f2depoch;
	assign d2ePC = f2dPC;

always@(Rfile[0],Rfile[1],Rfile[2],Rfile[3],Rfile[4],Rfile[5],Rfile[6],Rfile[7],Rfile[8],Rfile[9],Rfile[10],

	Rfile[11],Rfile[12],Rfile[13],Rfile[14],Rfile[15],Rfile[16],Rfile[17],Rfile[18],Rfile[19],Rfile[20],

	Rfile[21],Rfile[22],Rfile[23],Rfile[24],Rfile[25],Rfile[26],Rfile[27],Rfile[28],Rfile[29],Rfile[30],Rfile[
31],
		Opcode,rs,rt,rd,funct,shamt,imm,Inst,d2erd,e2mrd,m2wbrd) begin
	case(Opcode)
		opADDIU, opSLTI, opSLTIU, opANDI, opORI, opXORI, opLUI:  // ALU instructions
				begin
					case(Opcode)
							opADDIU, opLUI: ALUfunc=  Add;
							opSLTI: ALUfunc= Slt;
							opSLTIU: ALUfunc= Sltu;
							opANDI: ALUfunc= And;
							opORI: ALUfunc= Or;
							opXORI: ALUfunc= Xor;
							default: ALUfunc= NotALU ;
					endcase

					Branchfunc= NT;
					op1=Rfile[rs];
					op2=imm;
```

```
                                        //des=rt;
                                        data=0;
                                        if(rs==d2erd || rs==e2mrd || rs==m2wbrd) begin
                                                stall = 1;
                                                IType=NOP;
                                                des=0;
                                        end
                                        else begin
                                                stall = 0;
                                                IType=Alu;
                                                des=rt;
                                        end
                                end

                opLW:begin              // DMem instructions
                        ALUfunc=NotALU;
                        Branchfunc=NT;
                        op1=Rfile[rs];
                        op2=imm;
                        data=0;
                        if(rs==d2erd || rs==e2mrd || rs==m2wbrd) begin
                                stall = 1;
                                IType=NOP;
                                des=0;
                        end
                        else begin
                                stall = 0;
                                IType=Ld;
                                des=rt;
                        end
                        end

                opSW:begin
                        ALUfunc=NotALU;
                        Branchfunc=NT;
                        op1=Rfile[rs];
                        op2=imm;
                        des=0;
                        data=Rfile[rt];
                        if(rs==d2erd || rs==e2mrd || rs==m2wbrd || rt==d2erd || rt==e2mrd ||
rt==m2wbrd) begin
                                stall = 1;
                                IType=NOP;
                        end
                        else begin
                                stall = 0;
                                IType=St;
                        end
                        end

                opJ,opJAL:begin         // Jump istructions
                        IType=J;
                        ALUfunc=NotALU;
                        Branchfunc=NT;
                        op1=Inst[25:0];
                        op2=0;
                        des=0;
                        data=0;
                        stall = 0;
                        end

                opBEQ, opBNE, opBLEZ, opBGTZ, opRT:begin   // Branch Instructions
                        ALUfunc=NotALU;
                        case(Opcode)
                                opBEQ:Branchfunc=Eq;
                                opBNE:Branchfunc=Neq;
                                opBLEZ:Branchfunc=Le;
                                opBGTZ:Branchfunc=Gt;
                                opRT:Branchfunc=(rt==rtBLTZ ? Lt : Ge);
                                default:Branchfunc=NT;
                         endcase

                        op1=Rfile[rs];
                        if (Opcode==opBEQ || Opcode==opBNE)          begin
                                op2=Rfile[rt];
                                if(rs==d2erd || rs==e2mrd || rs==m2wbrd || rt==d2erd || rt==e2mrd
            || rt==m2wbrd) begin
                                        stall = 1;
                                        IType=NOP;
                                end
                                else begin
                                        stall = 0;
                                        IType=Br;
                                end
                        end
                        else begin
                                op2 = 0;
                                if(rs==d2erd || rs==e2mrd || rs==m2wbrd) begin
                                        stall = 1;
                                        IType=NOP;
                                end
                                else begin
                                        stall = 0;
                                        IType=Br;
                                end
                        end
            des=0;
            data={{16{imm[15]}},imm};
            end

opFUNC:
            case(funct)
                        funcJR:begin // funcJALR not used
                                    ALUfunc=NotALU;
                                    Branchfunc=AT;
                                    op1=Rfile[rs];
                                    op2=0;
                                    des=0;
                                    data=0;
                                    if(rs==d2erd || rs==e2mrd || rs==m2wbrd) begin
                                                stall = 1;
                                                IType=NOP;
                                    end
                                    else begin
                                                stall = 0;
                                                IType=Jr;
                                    end
                                    end

                        funcSLL, funcSRL, funcSRA:begin
                                    case(funct)
                                                funcSLL:ALUfunc=Lshift;
                                                funcSRL:ALUfunc=Rshift;
                                                funcSRA:ALUfunc=Rshift;
                                                default:ALUfunc=NotALU;
                                    endcase
                                    Branchfunc=NT;
                                    op1=Rfile[rt];
                                    op2={{26{shamt[5]}},shamt};
                                    data=0;
                                    if(rt==d2erd || rt==e2mrd || rt==m2wbrd) begin
                                                stall = 1;
                                                IType=NOP;
                                                des=0;
                                    end
                                    else begin
                                                stall = 0;
                                                IType = Alu;
                                                des=rd;
                                    end
                          end
```

```
                    funcADDU, funcSUBU, funcAND, funcOR, funcXOR, funcNOR, funcSLT,
funcSLTU:begin
                        case(funct)
                            funcADDU: ALUfunc= Add;
                            funcSUBU: ALUfunc= Sub;
                            funcAND : ALUfunc=And;
                            funcOR  : ALUfunc=Or;
                            funcXOR : ALUfunc=Xor;
                            funcNOR : ALUfunc=Nor;
                            funcSLT : ALUfunc=Slt;
                            funcSLTU: ALUfunc=Sltu;
                            default:ALUfunc=NotALU;
                        endcase
                        Branchfunc=NT;
                        op1=Rfile[rs];
                        op2=Rfile[rt];
                        data=0;
                        if(rs==d2erd || rs==e2mrd || rs==m2wbrd || rt==d2erd ||
rt==e2mrd || rt==m2wbrd) begin
                            stall = 1;
                            IType = NOP;
                            des=0;
                        end
                        else begin
                            stall = 0;
                            IType = Alu;
                            des=rd;
                        end
                      end
                    default:begin
                        IType= NOP;
                        ALUfunc=NotALU;
                        Branchfunc=NT;
                        op1=0;
                        op2=0;
                        des=0;
                        data=0;
                        stall=0;
                        end
                endcase

                default:  begin
                    IType= NOP;
                    ALUfunc=NotALU;
                    Branchfunc=NT;
                    op1=0;
                    op2=0;
                    des=0;
```

```
                                        data=0;
                                        stall=0;
                              end
                    endcase
end

always@(posedge clk)begin
                              if (reset)
                                        for(i=0;i<=31;i=i+1)
                                                  Rfile[i]<=0;
                              else if(w_en)
                                        Rfile[w_addr]<=w_data;
                              else
                                        ;
end


endmodule
```

MODULE : ALU

```
`timescale 1ns / 1ps
module alu(input[2:0] d2e_itype,input d2e_epoch,input gepoch, input[3:0] Alufunc,input[2:0]
Brfunc,input[31:0] Op1,Op2,data,input[9:0] pc, input[4:0] rd,input [9:0] d2ePC,
   output reg[9:0] next_pc, output reg mispred, output reg[31:0] res, output reg[2:0] e2m_itype,
output[9:0] Memaddr, output reg[4:0] rdest);

          ///// Instruction Type ////
          parameter Alu=3'b 000;
          parameter Ld=3'b 001;
          parameter St=3'b 010;
          parameter J=3'b 011;
          parameter Jr=3'b 100;
          parameter Br=3'b 101;
          parameter NOP=3'b111;
          //// Alufuncs ///////
          parameter Add=4'b 0000;
          parameter Sub=4'b 0001;
          parameter And=4'b 0010;
          parameter Or=4'b 0011;
          parameter Xor=4'b 0100;
          parameter Nor=4'b 0110;
          parameter Slt=4'b 0111;
          parameter Sltu=4'b 1000;
          parameter Lshift=4'b 1001;
          parameter Rshift=4'b 1010;
          ///// Branch Functions ///
          parameter Eq=3'b 000;
          parameter Neq=3'b 001;
          parameter Le=3'b 010;
          parameter Lt=3'b 011;
          parameter Ge=3'b 100;
          parameter Gt=3'b 101;
          //parameter AT=4'b 110;
          //parameter NT=4'b 111;

          //output reg[31:0] res,next_pc;
          wire [31:0] sum;
          wire [9:0] br_calc,pc_calc;//,pc_pred;

          //assign rdest = rd;
          //assign e2m_itype = d2e_itype;
          assign Memaddr = sum[9:0];
          assign sum = Op1+Op2;
          assign pc_calc = pc+1;
          //assign pc_pred = d2ePC+1;
          assign br_calc = data[9:0]+d2ePC+1;
          //assign store_value = data;

          always@(*) begin
                  if (gepoch == d2e_epoch) begin
                          e2m_itype = d2e_itype;
                          rdest = rd;
                          case(d2e_itype)
                                  Alu: begin
                                          next_pc = pc_calc;
                                          mispred = 0;
                                          case(Alufunc)
                                                  Add: res = sum;
                                                  Sub: res = Op1-Op2;
                                                  And: res = Op1&Op2;
                                                  Or:  res = Op1|Op2;
                                                  Xor: res = Op1^Op2;
                                                  Nor: res = ~(Op1|Op2);
                                                  Slt: res = (Op1<Op2) ? 1 : 0 ;
                                                  Sltu: res = (Op1<Op2) ? 1 : 0 ;
                                                  Lshift: res = Op1<<Op2;
                                                  Rshift: res = Op1>>Op2;
                                                  default: res = 0;
                                          endcase
                                  end

                                  J: begin
                                          next_pc= Op1[9:0];
                                          mispred=1;
                                          res=0;
```

```
end

Jr: begin
        next_pc = Op1[9:0];
        mispred=1;
        res=0;
end

Br: begin
        res=0;
        case(Brfunc)
                Eq: next_pc = (Op1==Op2) ? br_calc : pc_calc;
                Neq:next_pc = (Op1!=Op2) ? br_calc : pc_calc;
                Lt: next_pc = (Op1<0) ? br_calc : pc_calc;
                Le: next_pc = (Op1<=0) ? br_calc : pc_calc;
                Gt: next_pc = (Op1>0) ? br_calc : pc_calc;
                Ge: next_pc = (Op1>=0) ? br_calc : pc_calc;
                default: next_pc= pc_calc;

        endcase
        case(Brfunc)
                Eq: mispred = (Op1==Op2) ? 1 : 0;
                Neq:mispred = (Op1!=Op2) ? 1 : 0;
                Lt: mispred = (Op1<0) ? 1 : 0;
                Le: mispred = (Op1<=0) ? 1 : 0;
                Gt: mispred = (Op1>0) ? 1 : 0;
                Ge: mispred = (Op1>=0) ? 1 : 0;
                default: mispred = 0;

        endcase

end

Ld: begin
        next_pc = pc_calc;
        res=0;//res <= data;
        mispred = 0;
end
St: begin
        next_pc = pc_calc;
        res = data;
        mispred = 0;
end
default: begin
        res = 0;
        next_pc = pc_calc;
        mispred = 0;
end
```

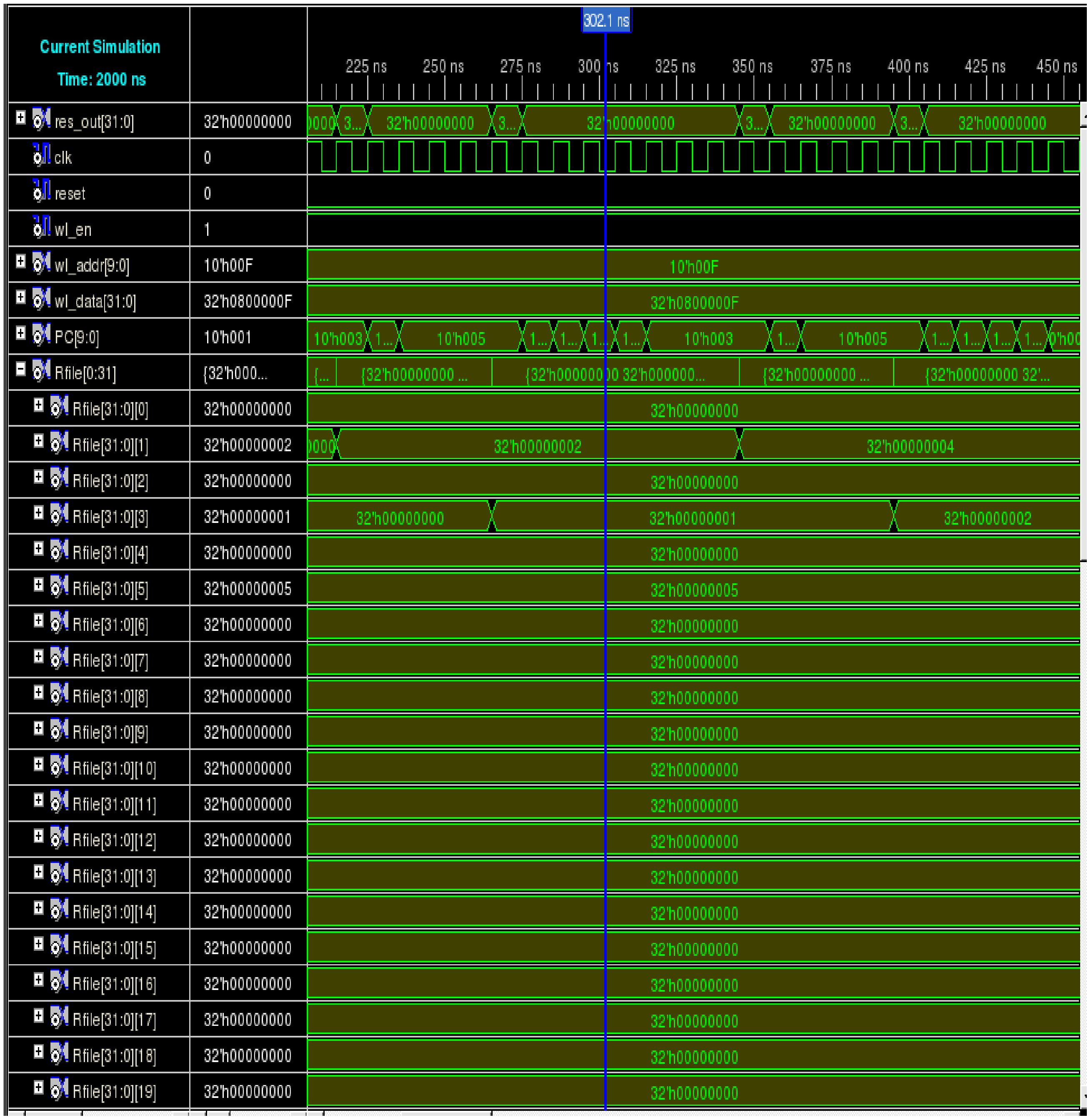
```
                endcase

            end
            else begin
                res = 0;
                next_pc = pc_calc;
                e2m_itype = NOP;
                mispred = 0;
                rdest = 0;
             end

        end


endmodule
```

MODULE : Test Bench: Finding an element in an array.

```
`timescale 1ns / 1ps

module test;

    // Inputs
    reg clk;
    reg reset;
    reg wl_en;
    reg [9:0] wl_addr;
    reg [31:0] wl_data;

    // Outputs
    wire [31:0] res_out;

    // Instantiate the Unit Under Test (UUT)
    cpu uut (
        .clk(clk),
        .reset(reset),
        .wl_en(wl_en),
        .wl_addr(wl_addr),
        .wl_data(wl_data),
        .res_out(res_out)
    );

    initial begin
        // Initialize Inputs
        clk = 0;
        reset = 1;
        wl_en = 1;
        #1
        wI_addr = 0;
        wI_data=32'b001001_00101_00101_0000_0000_0000_0101;
        #10
        wI_addr = 1;
        wI_data=32'b001001_00001_00001_0000_0000_0000_0010;
        #10
        wI_addr = 2;
        wI_data=32'b101011_00011_00001_0000_0000_0000_0000;
        #10
        wI_addr = 3;
        wI_data=32'b001001_00011_00011_0000_0000_0000_0001;
        #10
        wI_addr = 4;
        wI_data=32'b000100_00011_00101_0000_0000_0000_0001;
        #10
        wI_addr = 5;
        wI_data=32'b000010_00000_00000_0000_0000_0000_0001;
        #10
        wI_addr = 6;
        wI_data=32'b000000_00011_00011_00011_00000_100011;
        #10
        wI_addr = 7;
        wI_data=32'b001001_00010_00010_0000_0000_0000_0110;
        #10
        wI_addr = 8;
        wI_data=32'b100011_00011_00001_0000_0000_0000_0000;
        #10
        wI_addr = 9;
        wI_data=32'b001001_00011_00011_0000_0000_0000_0001;
        #10
        wI_addr = 10;
        wI_data=32'b000100_00001_00010_0000_0000_0000_0010;
        #10
        wI_addr = 11;
        wI_data=32'b000100_00011_00101_0000_0000_0000_0010;
        #10
        wI_addr = 12;
        wI_data=32'b000010_00000_00000_0000_0000_0000_1000;
        #10
        wI_addr = 13;
        wI_data=32'b101011_00100_00100_0000_0000_0000_0001;
        #10
        wI_addr = 14;
        wI_data=32'b101011_00100_00100_0000_0000_0000_0001;
        #10
        reset=0;

end
```

```
        always begin
         #5 clk=~clk;
        end

endmodule
```

Simulation Diagram for the Test Bench



res_out[31:0]	32'h00000000
clk	0
reset	0
wl_en	1
wl_addr[9:0]	10'h00F
wl_data[31:0]	32'h0800000F
PC[9:0]	10'h001
Rfile[0:31]	{32'h000...
Rfile[31:0][0]	32'h00000000
Rfile[31:0][1]	32'h00000002
Rfile[31:0][2]	32'h00000000
Rfile[31:0][3]	32'h00000001
Rfile[31:0][4]	32'h00000000
Rfile[31:0][5]	32'h00000005
Rfile[31:0][6]	32'h00000000
Rfile[31:0][7]	32'h00000000
Rfile[31:0][8]	32'h00000000
Rfile[31:0][9]	32'h00000000
Rfile[31:0][10]	32'h00000000
Rfile[31:0][11]	32'h00000000
Rfile[31:0][12]	32'h00000000
Rfile[31:0][13]	32'h00000000
Rfile[31:0][14]	32'h00000000
Rfile[31:0][15]	32'h00000000
Rfile[31:0][16]	32'h00000000
Rfile[31:0][17]	32'h00000000
Rfile[31:0][18]	32'h00000000
Rfile[31:0][19]	32'h00000000