1. Abstract

Aim:

The goal of this project is to implement and improve the 5-stage pipelined processor. The design is aiming to perform 15-by-15 matrix multiplication.

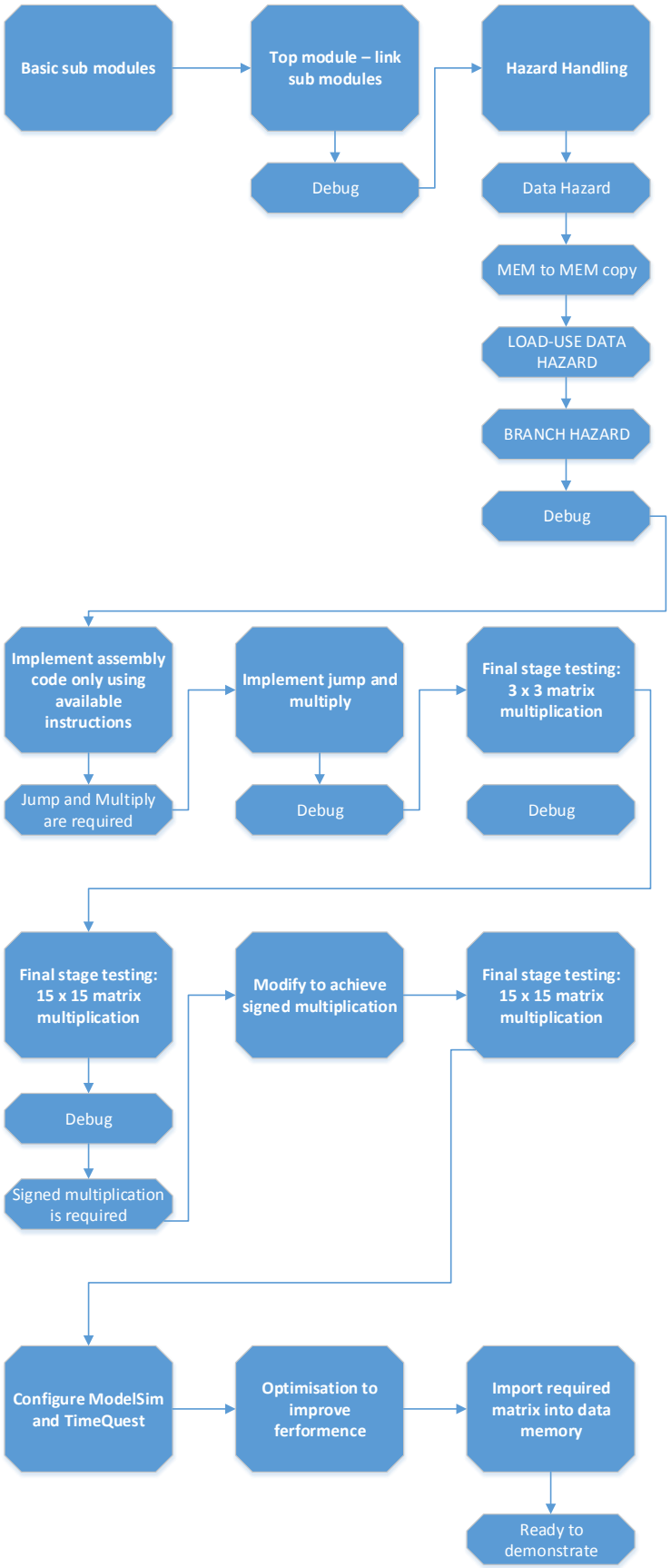
Hazard handled:

- Data hazard
  - the next Rtype need the result of current Rtype instruction.
- Mem to mem copy
  - sw after lw
- Load use hazard
  - The next Rtype need the result of lw in current instruction.
- Brach and jump
  - Stall or/and flush are required.

Assumptions:

- The matrix multiplication result fits in 32 bits.

2. Development process

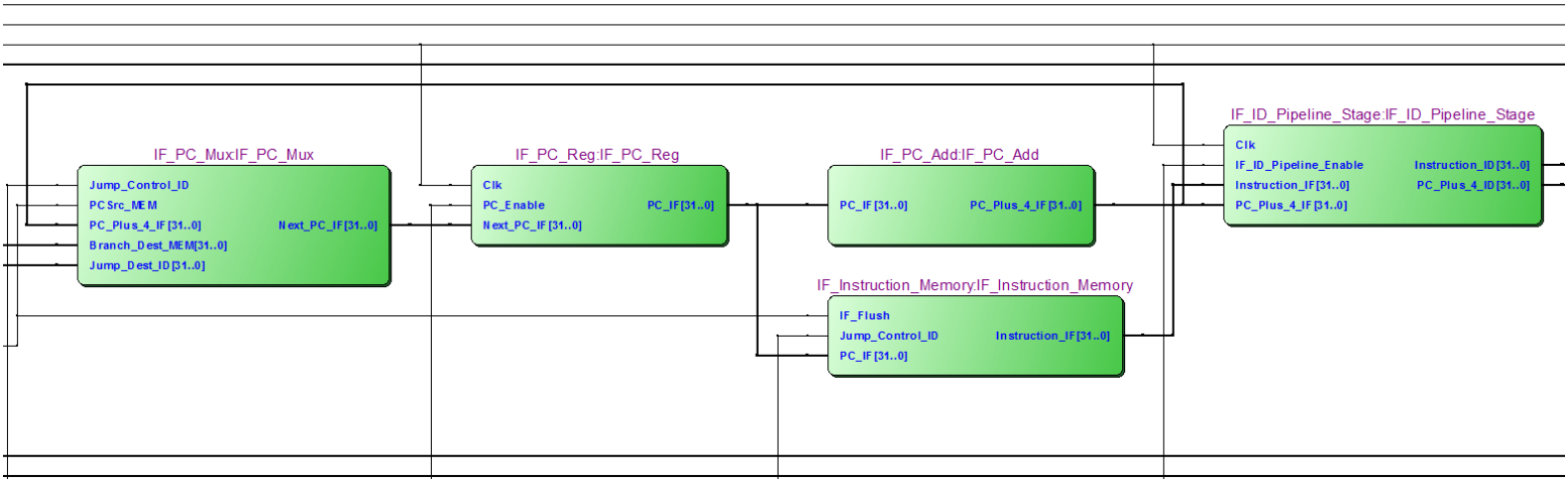


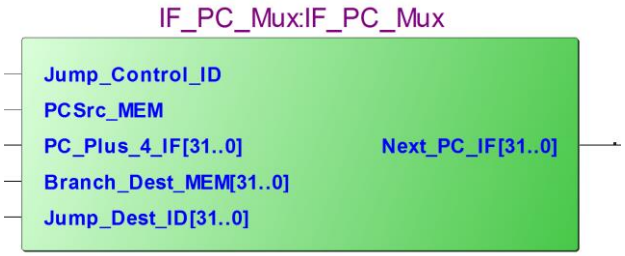
3. Datapath

In order to achieve high performance, process is divided into five pipeline stages.

IF	ID	EX	MEM	WB
Instruction Fetch	Instruction Decode	Execution	Memory	Write Back

Instruction Fetch stage:

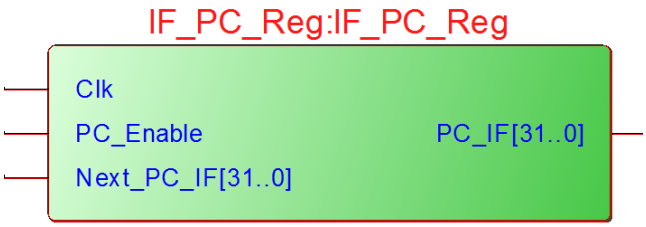




- // The IF_PC_Mux chooses what is next pc, normally pc_plus_4 is outputted to be next_pc_if, but in case of branch and jump, and corresponding next_pc_if is passed over.
- // PCSrc_MEM triggers the branch response, forwarding Branch_Dest_MEM to Next_PC_IF.
- // PCSrc_MEM is from MEM_Branch_AND, Branch_Dest_MEM is from EX_PC_Add
- // Note PCSrc_MEM, MEM_Branch_AND, Branch_Dest_MEM and EX_PC_Add is actually been
- // relocated from MEM stage to ID stage.
- // Jump_Control_ID triggers the jump response, forwarding Jump_Dest_ID to Next_PC_IF.
- // Jump_Control_ID is from ID_Control, Jump_Dest_ID is from ID_Jump.

```
module IF_PC_Mux(
        input [31:0] PC_Plus_4_IF,
        input [31:0] Branch_Dest_MEM,
        input [31:0] Jump_Dest_ID,
```

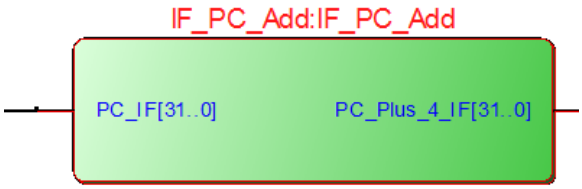
```
                input           PCSrc_MEM, // actually ID stage
                input           Jump_Control_ID,
                output [31:0]Next_PC_IF
                );

  assign Next_PC_IF = Jump_Control_ID ? Jump_Dest_ID : (PCSrc_MEM ? Branch_Dest_MEM :
PC_Plus_4_IF);

endmodule // IF_PC_Mux
```



- // IF_PC_Reg push next_PC_IF to PC_IF on positive edge of global clock.
- // A logic low in PC_Enable will stop next_PC_IF been pushed to PC_IF.
- // PC_Enable is from the Hazard Handling Unit

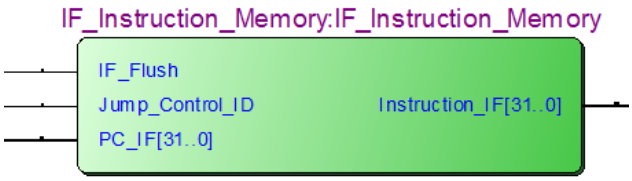
```
module IF_PC_Reg(
                input [31:0]Next_PC_IF,
                input PC_Enable,
                output reg [31:0]PC_IF,
                input Clk
                );

  always@(posedge Clk)
                begin
                        if(PC_Enable)
                                begin
                                        PC_IF <= Next_PC_IF;
                                end
                end
```

endmodule // IF_PC_Reg



// IF_PC_Add add the pc by digital four.

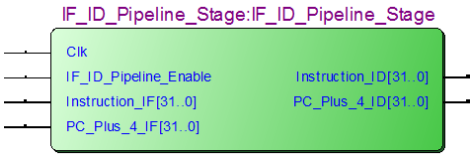
```
module IF_PC_Add(
                    input [31:0]  PC_IF,
                    output [31:0] PC_Plus_4_IF
                    );

  assign PC_Plus_4_IF=PC_IF + 4;

endmodule // IF_PC_Add
```



- // IF_Instruction_Memory is the main instruction memory, which is 32 bits wide, and have a depth of 1024.
- // Once the PC_IF eceeds the depth of instruction memory, it will output zero.
- // A logic high of IF_Flush triggers the flush response, the output will be zero.
- // IF_Flush is from Mem_Brach_AND, note Mem_Brach_AND is been relocated from MEM stage to ID stage.
- // A logic high of Jump_Control_ID triggers the flush response, the output will be zero.
- // Jump_Control_ID is from ID_Control

```
module IF_Instruction_Memory(
                    input [31:0]PC_IF,
```

```
        input           IF_Flush,
        input           Jump_Control_ID,
        output [31:0]Instruction_IF
        );

        reg [31:0]Instruction_Memory[0:1023];

        initial begin
            $readmemh("instruction_memory.list", Instruction_Memory);
            //Instruction_IF <= 32'd0;
        end

        assign Instruction_IF = ((PC_IF>32'd1023) | IF_Flush | Jump_Control_ID) ? 32'd0 :
Instruction_Memory[PC_IF];

endmodule // IF_Instruction_Memory
```

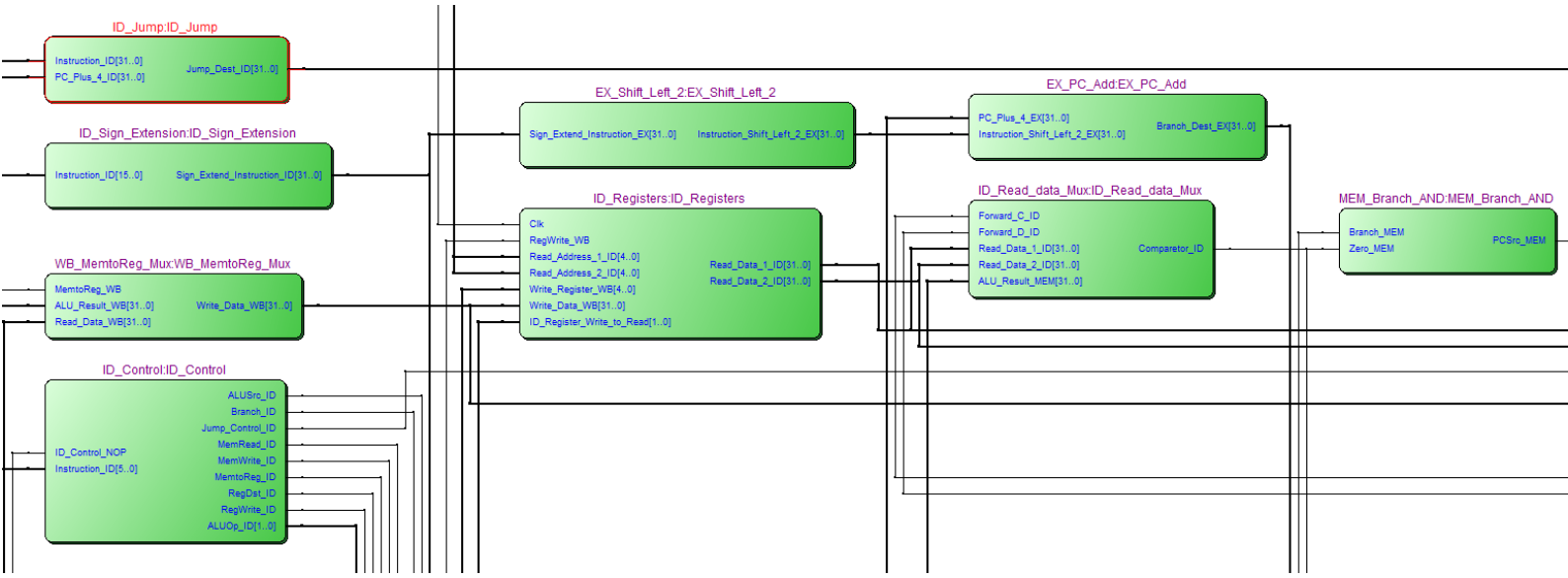


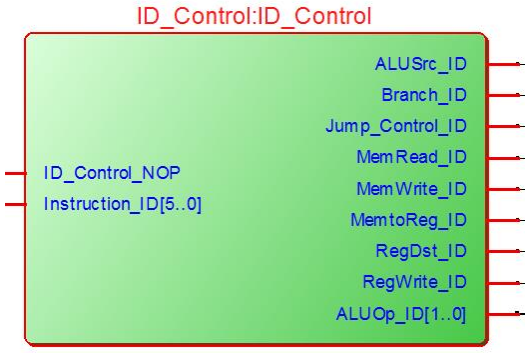
- // Instruction fetch stage to instruction decode stage pipeline, passing data on positive edge of global clock

```
module IF_ID_Pipeline_Stage(
                    input [31:0]   Instruction_IF,
                    input [31:0]   PC_Plus_4_IF,
                    input          IF_ID_Pipeline_Enable,

                    output reg [31:0] Instruction_ID,
                    output reg [31:0] PC_Plus_4_ID,
```

```
                    input           Clk
                    );

  initial begin
            Instruction_ID      <= 32'd0;
            PC_Plus_4_ID        <= 32'd0;
  end

  always@(posedge Clk) begin
            if(IF_ID_Pipeline_Enable)
                    begin
                            Instruction_ID<=Instruction_IF;
                            PC_Plus_4_ID<=PC_Plus_4_IF;
                    end
        end
endmodule // IF_ID_Pipeline_Stage
```

Instruction Decode Stage





- // ID_Control outputs important control signals, most of these signals will be feed into flowing pipeline, hence into other stages.
- // Rtype, lw, sw, beq, nop and jump are handled.
- // Control signal is based on opcode, which is [31:26] of Instruction_ID.
- // When ID_Control_NOP is logic one, nop control signals are forced to the outputs regardless opcode.
- // ID_Control_NOP is from Hazard handling unit.

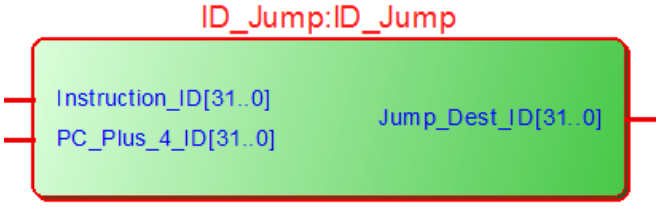
```
module ID_Control(
                input [5:0]      Instruction_ID,
                input            ID_Control_NOP,
                output reg       RegWrite_ID,
                output reg       MemtoReg_ID,
```

```
        output reg      Branch_ID,
        output reg      Jump_Control_ID,
        output reg      MemRead_ID,
        output reg      MemWrite_ID,

        output reg      RegDst_ID,
        output reg [1:0] ALUOp_ID,
        output reg       ALUSrc_ID
        );

parameter RTYPE      = 6'b000000;
parameter LW         = 6'b100011;
parameter SW         = 6'b101011;
parameter BEQ        = 6'b000100;
parameter NOP        = 6'b100000;
parameter JUMP       = 6'b000010;

wire [5:0]opcode;
assign opcode = (ID_Control_NOP & (Instruction_ID != BEQ)) ? NOP : Instruction_ID;

initial
 begin
        RegDst_ID           <= 0;
        ALUOp_ID            <= 2'd0;
        ALUSrc_ID           <= 0;

        Branch_ID           <= 0;
        Jump_Control_ID<= 0;
        MemRead_ID          <= 0;
        MemWrite_ID         <= 0;
```

```
            RegWrite_ID             <= 0;
            MemtoReg_ID             <= 0;
        end

        always@* begin
            case(opcode)
                RTYPE: begin
                    RegWrite_ID <= 1'b1;
                    MemtoReg_ID <= 1'b0;
                    Branch_ID       <= 1'b0;
                    Jump_Control_ID<= 1'b0;
                    MemRead_ID    <= 1'b0;
                    MemWrite_ID   <= 1'b0;
                    RegDst_ID       <= 1'b1;
                    ALUOp_ID        <= 2'b10;
                    ALUSrc_ID       <= 1'b0;
                end //RTYPE
                LW: begin
                    RegWrite_ID     <= 1'b1;
                    MemtoReg_ID   <= 1'b1;
                    Branch_ID       <= 1'b0;
                    Jump_Control_ID<= 1'b0;
                    MemRead_ID    <= 1'b1;
                    MemWrite_ID   <= 1'b0;
                    RegDst_ID       <= 1'b0;
                    ALUOp_ID        <= 2'b00;
                    ALUSrc_ID       <= 1'b1;
                end //LW
                SW: begin
                    RegWrite_ID      <= 1'b0;
                    MemtoReg_ID    <= 1'b0;
```

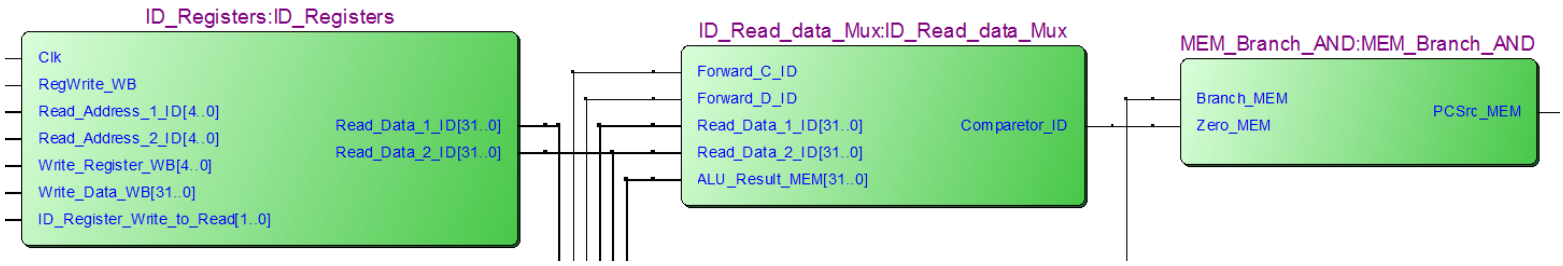
```
            Branch_ID        <= 1'b0;
            Jump_Control_ID<= 1'b0;
            MemRead_ID     <= 1'b0;
            MemWrite_ID   <= 1'b1;
            RegDst_ID        <= 1'bx;
            ALUOp_ID         <= 2'b00;
            ALUSrc_ID        <= 1'b1;
        end //SW
        BEQ: begin
            RegWrite_ID     <= 1'b0;
            MemtoReg_ID   <= 1'b0;
            Branch_ID        <= 1'b1;
            Jump_Control_ID<= 1'b0;
            MemRead_ID    <= 1'b0;
            MemWrite_ID   <= 1'b0;
            RegDst_ID        <= 1'bx;
            ALUOp_ID         <= 2'b01;
            ALUSrc_ID        <= 1'b0;
        end //BEQ
        NOP: begin
            RegWrite_ID     <= 1'b0;
            MemtoReg_ID  <= 1'b0;
            Branch_ID        <= 1'b0;
            Jump_Control_ID<= 1'b0;
            MemRead_ID    <= 1'b0;
            MemWrite_ID   <= 1'b0;
            RegDst_ID        <= 1'b0;
            ALUOp_ID         <= 2'b00;
            ALUSrc_ID        <= 1'b0;
        end //NOP
        JUMP: begin
```

```
                RegDst_ID       <= 1'b0;
                ALUOp_ID        <= 2'b00;
                ALUSrc_ID       <= 1'b0;
                Branch_ID       <= 1'b0;
                Jump_Control_ID <= 1'b1;
                MemRead_ID      <= 1'b0;
                MemWrite_ID     <= 1'b0;
                RegWrite_ID     <= 1'b0;
                MemtoReg_ID     <= 1'b0;
            end //JUMP
            default: begin
                RegWrite_ID     <= 1'b0;
                MemtoReg_ID     <= 1'b0;
                Branch_ID       <= 1'b0;
                Jump_Control_ID<= 1'b0;
                MemRead_ID      <= 1'b0;
                MemWrite_ID     <= 1'b0;
                RegDst_ID       <= 1'b0;
                ALUOp_ID        <= 2'b00;
                ALUSrc_ID       <= 1'b0;
            end
        endcase
    end //always
endmodule // ID_Control
```



- // ID_Jump calculates the jump destination based on Instruction_ID and PC_Plus_4_ID.

```
module ID_Jump(
                input [31:0]        Instruction_ID,
                input [31:0]        PC_Plus_4_ID,

                output[31:0]        Jump_Dest_ID
                );

  assign Jump_Dest_ID = {{PC_Plus_4_ID[31:28]},{Instruction_ID[27:0] << 2}};

endmodule // IF_PC_Mux
```

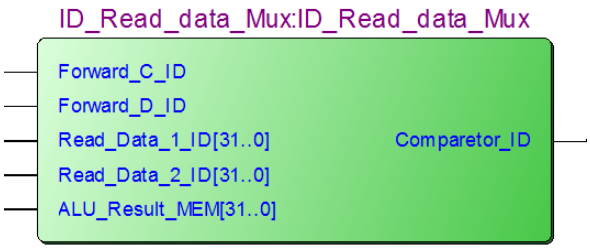


- // ID_Registers is the main register, 32 bit wide, 32 bit deep
- // A logic high of ID_Register_Write_to_Read will trigger the write_Data_WB been forwarded to Read_Data
- // ID_Register_Write_to_Read is from Hazard_Handling_Unit

```
module ID_Registers(
                input [4:0]   Read_Address_1_ID,
                input [4:0]   Read_Address_2_ID,
                input [4:0]   Write_Register_WB,
                input [31:0]  Write_Data_WB,
```

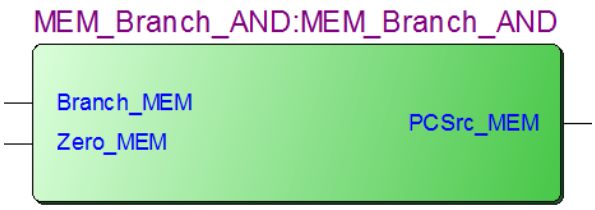
```
                output reg  [31:0] Read_Data_1_ID,
                output reg  [31:0] Read_Data_2_ID,
                input   Clk,
                input   RegWrite_WB,
                        input [1:0]       ID_Register_Write_to_Read
                );

  reg [31:0]Register_File[0:31];

        initial begin
                $readmemh("register_file.list", Register_File);
                Read_Data_1_ID <= 32'd0;
                Read_Data_2_ID <= 32'd0;
        end

        // Forwarding Write_Data_WB to Read_Data_1_ID
        always@(Read_Address_1_ID or Register_File[Read_Address_1_ID] or
ID_Register_Write_to_Read) begin
                if(Read_Address_1_ID==5'd0)
                        begin
                                Read_Data_1_ID <= 32'd0;
                        end
                else
                        begin
                                if(ID_Register_Write_to_Read == 2'b01)
                                        begin
                                                Read_Data_1_ID <= Write_Data_WB;
                                        end
                                else
                                        begin
                                                Read_Data_1_ID <=
Register_File[Read_Address_1_ID];
```

```
                                        end
                        end //else
        end

        // Forwarding Write_Data_WB to Read_Data_2_ID
        always@(Read_Address_2_ID or Register_File[Read_Address_2_ID] or
ID_Register_Write_to_Read) begin
                if(Read_Address_2_ID==5'd0)
                        begin
                                Read_Data_2_ID <= 32'd0;
                        end
                else
                        begin
                                if(ID_Register_Write_to_Read == 2'b10)
                                        begin
                                                Read_Data_2_ID <= Write_Data_WB;
                                        end
                                else
                                        begin
                                                Read_Data_2_ID <=
Register_File[Read_Address_2_ID];
                                        end
                        end //else
        end

        always@(posedge Clk) begin
                if( RegWrite_WB & (Write_Register_WB != 5'd0) ) begin
                        Register_File[Write_Register_WB] <= Write_Data_WB;
                end //if
        end //always

endmodule // ID_Registers
```



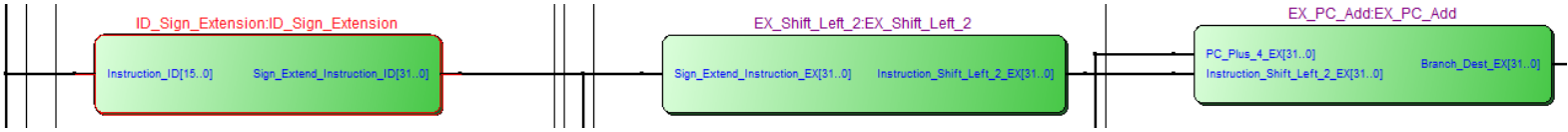
- // ID_Read_data_Mux helps handling branch hazard.
- // Since the branch unit is relocated from MEM stage to ID stage, the output Comparator_ID is the equivalent of EX_zero signal
- // Firstly, Forward_C_ID and Forward_D_ID are the select signal of two mux, selecting which of Read_Data_ID or ALU_Result_MEM is feed to comparator.
- // Forward_C_ID and Forward_D_ID are from Hazard handling unit.
- // Secondly, the comparator outputs logic high when two inputs are identical

```
module ID_Read_data_Mux(
                        input [31:0]  Read_Data_1_ID,
                        input [31:0] Read_Data_2_ID,
                        input [31:0]  ALU_Result_MEM,
                        input          Forward_C_ID,
                        input           Forward_D_ID,
                        output        Comparetor_ID
                        );

wire [31:0] Read_Data_1_MUX_ID;
wire [31:0] Read_Data_2_MUX_ID;
assign Read_Data_1_MUX_ID = Forward_C_ID ? ALU_Result_MEM : Read_Data_1_ID;
assign Read_Data_2_MUX_ID = Forward_D_ID ? ALU_Result_MEM : Read_Data_2_ID;

assign Comparetor_ID = (Read_Data_1_MUX_ID == Read_Data_2_MUX_ID);

endmodule // ID_Read_data_Mux
```



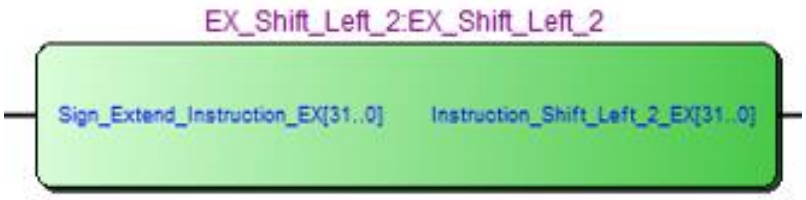
- // perform an logic and of Branch_MEM and Zero_MEM
- // note this module is relocated from MEM stage to ID stage
- // Branch_MEM is connected to Branch_ID in top module.
- // Zero_MEM is the comparator output of ID_Read_data_Mux
- // The output PCSrc_MEM indicated a branch happened. It is feed to IF_PC_Mux to handle branch hazard

```
module MEM_Branch_AND(
                    input Branch_MEM, // actually is ID stage signal
                    input  Zero_MEM,
                    output PCSrc_MEM
                    );

  assign PCSrc_MEM = Branch_MEM & Zero_MEM;

endmodule // MEM_Branch_AND
```



- // ID_Sign_Extension sign extend the immediate part of Instruction_ID

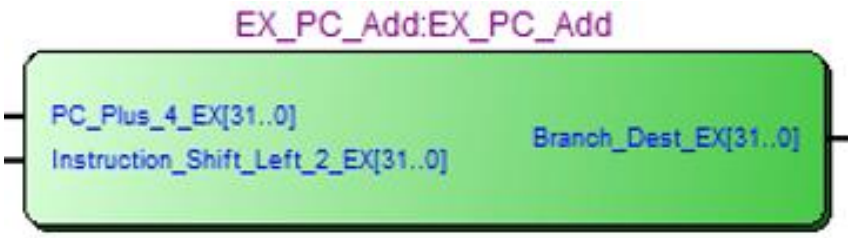
```
module ID_Sign_Extension(
                input [15:0] Instruction_ID,
                output [31:0] Sign_Extend_Instruction_ID
                );

        assign Sign_Extend_Instruction_ID = {{16{Instruction_ID[15]}},Instruction_ID[15:0]};

endmodule // ID_Sign_Extension
```



- // Note this module is relocated from EX stage to ID stage,
- // Input Sign_Extend_Instruction_EX is feed as Sign_Extend_Instruction_ID in top module.
- // output Instruction_Shift_Left_2_EX is actual offset need to be added to PC.
- // EX_Shift_Left_2 shift the input left by two.

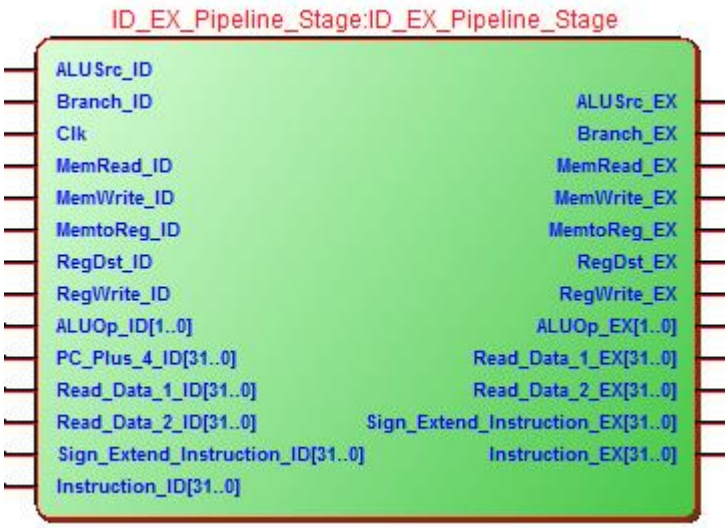
```
module EX_Shift_Left_2(
                input [31:0] Sign_Extend_Instruction_EX,
                output [31:0] Instruction_Shift_Left_2_EX
                );
        assign Instruction_Shift_Left_2_EX = Sign_Extend_Instruction_EX << 2;
```

endmodule // EX_Shift_Left_2



- // Note this module is relocated from EX stage to ID stage,
- // PC_Plus_4_EX is actually connected to PC_Plus_4_ID in top module.
- // The output Branch_Dest_EX is feed to IF_PC_Mux to help handling branch hazard.
- // EX_PC_Add add the shifted sign extended instruction immediate part and PC_Plus_4_ID.

```
module EX_PC_Add(
                    input [31:0]  PC_Plus_4_EX, // actually from ID stage
                    input [31:0]  Instruction_Shift_Left_2_EX,
                    output [31:0] Branch_Dest_EX
                    );

  assign Branch_Dest_EX = PC_Plus_4_EX + Instruction_Shift_Left_2_EX;

endmodule // EX_PC_Add
```



- // Instruction decode stage to execution stage pipeline, passing data on positive edge of global clock

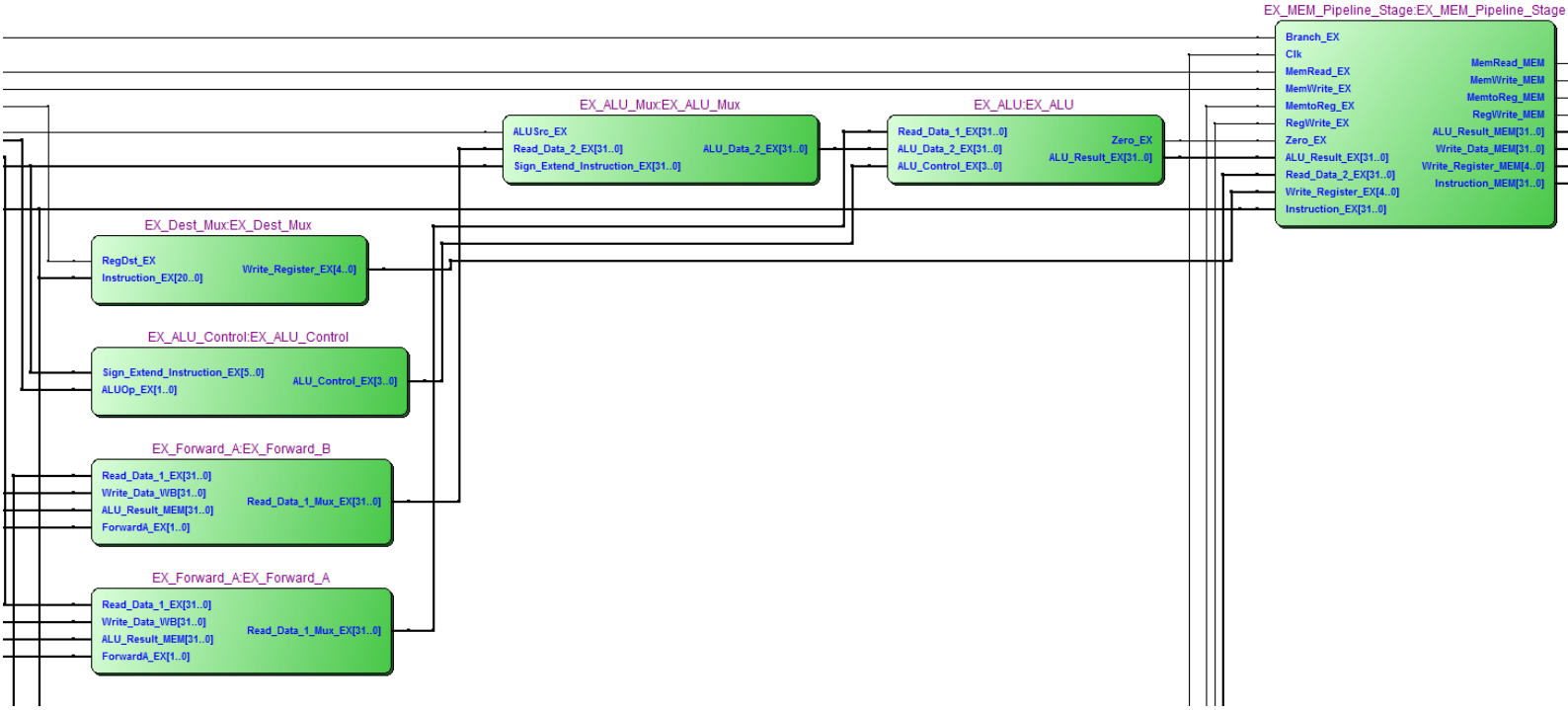
```
module ID_EX_Pipeline_Stage(

                input           RegWrite_ID,
                input           MemtoReg_ID,

                input           Branch_ID,
                input           MemRead_ID,
                input           MemWrite_ID,

                input           RegDst_ID,
                input [1:0]   ALUOp_ID,
                input           ALUSrc_ID,

                input [31:0]  PC_Plus_4_ID,

                input [31:0]  Read_Data_1_ID,
                input [31:0]  Read_Data_2_ID,

                input [31:0]  Sign_Extend_Instruction_ID,

                input [31:0]  Instruction_ID,

                output reg      RegWrite_EX,
                output reg      MemtoReg_EX,

                output reg      Branch_EX,
                output reg      MemRead_EX,
```

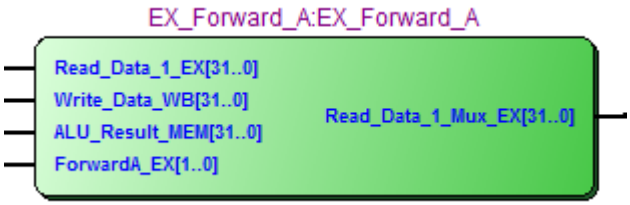
```
                output reg      MemWrite_EX,

                output reg      RegDst_EX,
                output reg [1:0]  ALUOp_EX,
                output reg      ALUSrc_EX,

                output reg [31:0] PC_Plus_4_EX,

                output reg [31:0] Read_Data_1_EX,
                output reg [31:0] Read_Data_2_EX,

                output reg [31:0] Sign_Extend_Instruction_EX,

                output reg [31:0] Instruction_EX,


                input           Clk
                );

initial
begin
        RegWrite_EX    <= 0;
        MemtoReg_EX  <= 0;

        Branch_EX               <= 0;
        MemRead_EX              <= 0;
        MemWrite_EX             <= 0;

        RegDst_EX               <= 0;
        ALUOp_EX                        <= 2'd0;
        ALUSrc_EX               <= 0;
```

```
            PC_Plus_4_EX   <= 32'd0;

            Read_Data_1_EX      <= 32'd0;
            Read_Data_2_EX      <= 32'd0;

            Sign_Extend_Instruction_EX <= 32'd0;

            Instruction_EX  <= 32'd0;
        end

always@(posedge Clk) begin
            RegWrite_EX    <= RegWrite_ID;
            MemtoReg_EX  <= MemtoReg_ID;

            Branch_EX           <= Branch_ID;
            MemRead_EX          <= MemRead_ID;
            MemWrite_EX         <= MemWrite_ID;

            RegDst_EX           <= RegDst_ID;
            ALUOp_EX                    <= ALUOp_ID;
            ALUSrc_EX           <= ALUSrc_ID;

            PC_Plus_4_EX   <= PC_Plus_4_ID;

            Read_Data_1_EX      <= Read_Data_1_ID;
            Read_Data_2_EX      <= Read_Data_2_ID;

            Sign_Extend_Instruction_EX <= Sign_Extend_Instruction_ID;

            Instruction_EX  <= Instruction_ID;
```

```
    end //always

endmodule // ID_EX_Pipeline_Stage
```

Execution Stage



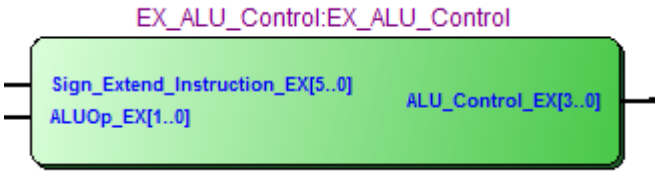


- // EX_Forward_A is a 3-to-1 mux.
- // Selection signal ForwardA_EX from Hazard handling unit.
- // ForwardA_EX of 2'b00 is the default case, it forwards Read_Data_1_EX from ID/EX pipeline to the output.
- // ForwardA_EX of 2'b01 forwards Write_Data_WB to the output.
- // ForwardA_EX of 2'b10 forwards ALU_Result_MEM to the output.
- // Selection signal ForwardA_EX is from Hazard handling unit
- // This module also been used as another instance as EX_Forward_B, which output data output to EX_ALU_Mux

```
module EX_Forward_A(
        input [31:0]        Read_Data_1_EX,
        input [31:0]        Write_Data_WB,
        input [31:0]        ALU_Result_MEM,

        input [1:0]         ForwardA_EX,

        output reg [31:0]   Read_Data_1_Mux_EX
            );

    initial
    begin
        Read_Data_1_Mux_EX <= 32'd0;
    end

    parameter   First       =   2'b00,
                Second =    2'b01,
                Third       =   2'b10;

  always@(*) begin
        case(ForwardA_EX)
            First:  Read_Data_1_Mux_EX <= Read_Data_1_EX;
            Second:     Read_Data_1_Mux_EX <= Write_Data_WB;
            Third:  Read_Data_1_Mux_EX <= ALU_Result_MEM;
            default: Read_Data_1_Mux_EX <= Read_Data_1_EX;
        endcase

    end //always

endmodule // EX_Forward_A
```

- // EX_ALU_Mux chooses what data feed into ALU_Data_2_EX
- // used when doing lw,sw, sign extended offset need to be feeded in, in order for ALU to calculate memory address.
- // when ALUSrc_EX is high, Sign_Extend_Instruction_EX is feed to the output.
- // selection signal ALUSrc_EX is originally from ID_Control

```
module EX_ALU_Mux(

                input [31:0]  Read_Data_2_EX,

                input [31:0]  Sign_Extend_Instruction_EX,

                input                   ALUSrc_EX,

                output [31:0] ALU_Data_2_EX


                );


  assign ALU_Data_2_EX = ALUSrc_EX ? Sign_Extend_Instruction_EX : Read_Data_2_EX;


endmodule // EX_ALU_Mux
```



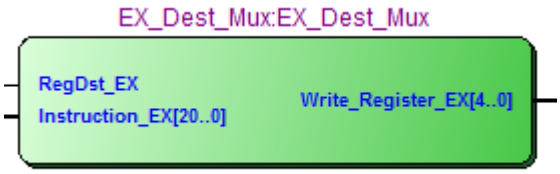
- // EX_ALU_Control decideds the operation need to be done is EX_ALU, depending on the opcode within the instruction.
- // R type: and, sub, and, or, slt, mul
- // I type: lw, sw, beq
- // J type: jump and branch are handled in other sub modules

```
module EX_ALU_Control(

                input [5:0] Sign_Extend_Instruction_EX, // Note: You only need 6 bits of this.

                input [1:0]  ALUOp_EX,
```

```
        output reg [3:0] ALU_Control_EX
        );

    parameter   Rtype   =   2'b10,//this is a 2 bit paramter,
                Radd    =   6'b100000,
                Rsub    =   6'b100010,
                Rand    =   6'b100100,
                Ror     =   6'b100101,
                Rslt    =   6'b101010,
                Rmul    =   6'b100001;  //this is a function code of addu but we treat it as mul.s

    parameter   lwsw    =   2'b00,      //since LW and SW use the same bit pattern, only way to store them as a paramter
                Itype   =   2'b01,      // beq
                xis     =   6'bXXXXXX;


    parameter   ALUadd  =   4'b0010,
                ALUsub  =   4'b0110,
                ALUand  =   4'b0000,
                ALUor   =   4'b0001,
                ALUslt  =   4'b0111,
                ALUmul  =   4'b1111;

    parameter   unknown =   2'b11,
                ALUx    =   4'b0011;

    wire [5:0]funct;
    assign funct = Sign_Extend_Instruction_EX[5:0];

    initial begin
```

```
		ALU_Control_EX <= ALUx;
	end

	always@* begin
/*
		if(ALUOp_EX == Rtype) begin
			case(funct)
				Radd:		ALU_Control_EX <= ALUadd;
				Rsub:		ALU_Control_EX <= ALUsub;
				Rand:		ALU_Control_EX <= ALUand;
				Ror:		ALU_Control_EX <= ALUor;
				Rslt:		ALU_Control_EX <= ALUslt;
				default:	ALU_Control_EX <= ALUx;
			endcase
		end //if

		else if(ALUOp_EX == lwsw) begin
			ALU_Control_EX <= ALUadd;
		end //else if

		else if(ALUOp_EX == Itype) begin
			ALU_Control_EX	<= ALUsub;
		end //else if

		else if(ALUOp_EX == unknown) begin
			ALU_Control_EX	<= ALUx;
		end //else if

		else begin ALU_Control_EX <= ALU_Control_EX; end
*/
```

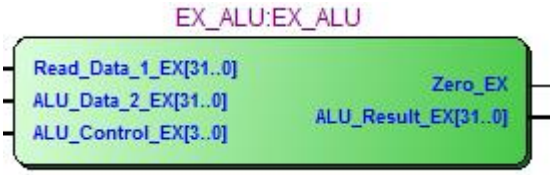
```
            case(ALUOp_EX)
                Rtype: begin
                    case(funct)
                        Radd:           ALU_Control_EX <= ALUadd;
                        Rsub:           ALU_Control_EX <= ALUsub;
                        Rand:           ALU_Control_EX <= ALUand;
                        Ror:            ALU_Control_EX <= ALUor;
                        Rslt:           ALU_Control_EX <= ALUslt;
                        Rmul:       ALU_Control_EX     <= ALUmul;
                        default:    ALU_Control_EX <= ALUx;
                    endcase
                end
                lwsw: ALU_Control_EX <= ALUadd;
                Itype: ALU_Control_EX  <= ALUsub;
                unknown: ALU_Control_EX    <= ALUx;
                default: ALU_Control_EX <= ALU_Control_EX;
            endcase


        end //always

endmodule // EX_ALU_Control
```



- // EX_Dest_Mux determines which register will be written to.

```
module EX_Dest_Mux(
            input [20:0] Instruction_EX,
```

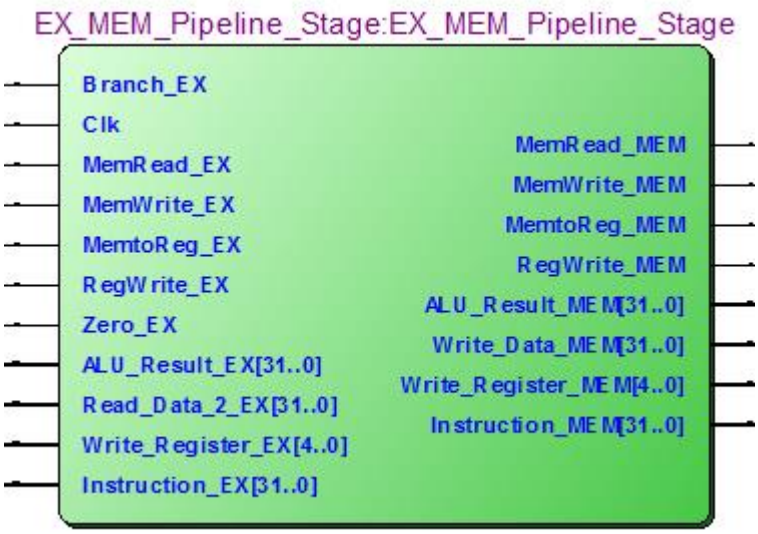
```
        input RegDst_EX,
        output [4:0] Write_Register_EX
        );

  assign Write_Register_EX = RegDst_EX ? Instruction_EX[15:11] : Instruction_EX[20:16];

endmodule // EX_Dest_Mux
```



- // ALU do the following operation
- // addition, subtraction, bitwise logic and, bitwise logic or, set less than by comparison, multiplication
- // signed calculation is considered and handled.
- // sign mismatch for set less than is considered and handled.

```
module EX_ALU(
        input signed [31:0]  Read_Data_1_EX,
        input signed [31:0]  ALU_Data_2_EX,
        input                [3:0]   ALU_Control_EX,
        output reg    [31:0]  ALU_Result_EX,
        output                        Zero_EX

        );

    parameter    ALUadd      =     4'b010,
                             ALUsub      =     4'b110,
                             ALUand      =     4'b000,
                             ALUor       =     4'b001,
                             ALUslt      =     4'b111,
                             ALUmul      =     4'b1111;
```

```
	initial
		begin
			ALU_Result_EX <= 32'd0;
		end

	// Handles negative inputs
	wire sign_mismatch;
	assign sign_mismatch = (Read_Data_1_EX[31]==ALU_Data_2_EX[31]);

	always@* begin
		case(ALU_Control_EX)
			ALUadd:		ALU_Result_EX <= Read_Data_1_EX + ALU_Data_2_EX;
			ALUsub:		ALU_Result_EX <= Read_Data_1_EX - ALU_Data_2_EX;
			ALUand:		ALU_Result_EX <= Read_Data_1_EX & ALU_Data_2_EX;
			ALUor:		ALU_Result_EX <= Read_Data_1_EX | ALU_Data_2_EX;
			ALUslt:		ALU_Result_EX <= Read_Data_1_EX < ALU_Data_2_EX ? (1 - sign_mismatch) : (0 + sign_mismatch);
			ALUmul:		ALU_Result_EX <= Read_Data_1_EX * ALU_Data_2_EX;
			default:	ALU_Result_EX <= 32'bx;		// control = ALUx | *
		endcase
	end //always

	assign Zero_EX = (ALU_Result_EX==0);

endmodule // EX_ALU
```

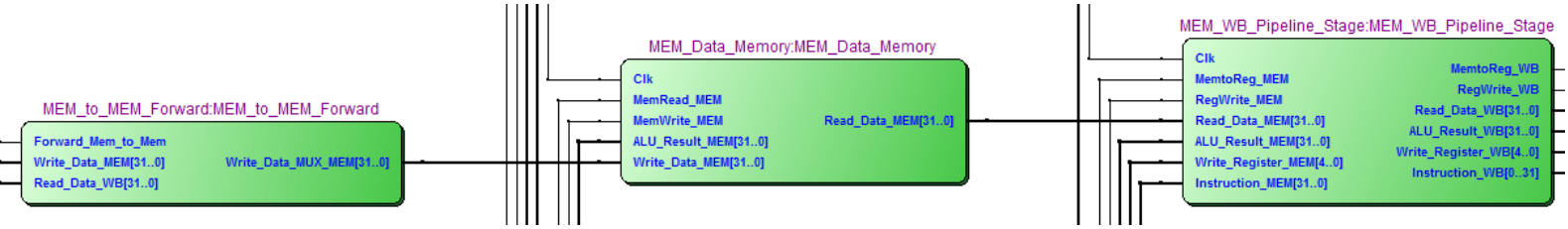


- // Execution to memory stage pipeline, passing data on positive edge of global clock

```
module EX_MEM_Pipeline_Stage(
                        input           RegWrite_EX,
                        input           MemtoReg_EX,

                        input           Branch_EX,
                        input           MemRead_EX,
                        input           MemWrite_EX,

                        input           Zero_EX,
                        input [31:0]  ALU_Result_EX,
                        input [31:0]   Read_Data_2_EX,
                        input [4:0]   Write_Register_EX,

                              input [31:0]              Instruction_EX,

                        output reg      RegWrite_MEM,
                        output reg      MemtoReg_MEM,
```

```
                output reg      Branch_MEM,
                output reg      MemRead_MEM,
                output reg      MemWrite_MEM,

                output reg      Zero_MEM,
                output reg [31:0] ALU_Result_MEM,
                output reg [31:0] Write_Data_MEM,
                output reg [4:0]  Write_Register_MEM,

                        output reg [31:0]               Instruction_MEM,

                input           Clk
                );

        always@(posedge Clk) begin
                RegWrite_MEM        <= RegWrite_EX;
                MemtoReg_MEM                <= MemtoReg_EX;

                Branch_MEM                  <= Branch_EX;
                MemRead_MEM                         <= MemRead_EX;
                MemWrite_MEM                <= MemWrite_EX;

                Zero_MEM                            <= Zero_EX;
                ALU_Result_MEM              <= ALU_Result_EX;
                Write_Data_MEM              <= Read_Data_2_EX;
                Write_Register_MEM<= Write_Register_EX;

                Instruction_MEM         <= Instruction_EX[31:0];
        end //always
endmodule // EX_MEM_Pipeline_Stage
```

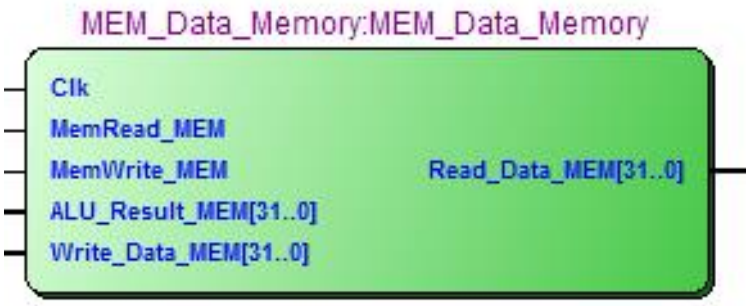
Memory Stage



- // when Forward_Mem_to_Mem is high, Write_Data_MEM is forwarded to Write_Data_MUX_MEM.
- // This deals with hazard when lw, sw to same location.
- // selection signal Forward_Mem_to_Mem is from Hazard handling unit

```
module MEM_to_MEM_Forward(
                                        input [31:0]        Write_Data_MEM,
                                        input [31:0]            Read_Data_WB,

                                        input
        Forward_Mem_to_Mem,

                                        output reg [31:0]       Write_Data_MUX_MEM
                        );

        initial
        begin
                Write_Data_MUX_MEM <= 32'd0;
        end

        parameter       First           =       0,
                                Second =        1;
```

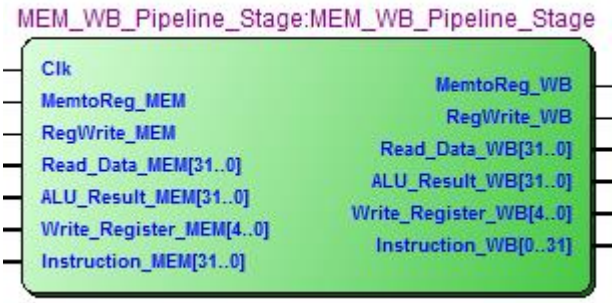
```
  always@(*) begin
                case(Forward_Mem_to_Mem)
                        First:     Write_Data_MUX_MEM <= Write_Data_MEM;
                        Second:          Write_Data_MUX_MEM <= Read_Data_WB;
                        default: Write_Data_MUX_MEM <= Write_Data_MEM;
                endcase

        end //always

endmodule // MEM_to_MEM_Forward
```



- // Main data_Memory
- // 32 bit width, 512 bit depth
- // location 0 is always 0.
- // invalid location get reading of 0.

```
module MEM_Data_Memory(
                        input [31:0]  ALU_Result_MEM,
                        input [31:0]  Write_Data_MEM,
                        output [31:0] Read_Data_MEM,
                        input         MemRead_MEM,
                        input         MemWrite_MEM,
                        input         Clk
                        );

        reg [31:0]Data_Memory[0:1023];
```

```
        initial begin
                $readmemh("data_memory.list", Data_Memory);
        end

assign Read_Data_MEM = MemRead_MEM ? ( (ALU_Result_MEM == 0 || ALU_Result_MEM > 32'd1023) ? 32'd0 : Data_Memory[ALU_Result_MEM]) : Read_Data_MEM;

        always@(posedge Clk)
                begin
                        if(MemWrite_MEM)
                                begin
                                        Data_Memory[ALU_Result_MEM] <= Write_Data_MEM;
                                end
                end

endmodule // MEM_Data_Memory
```



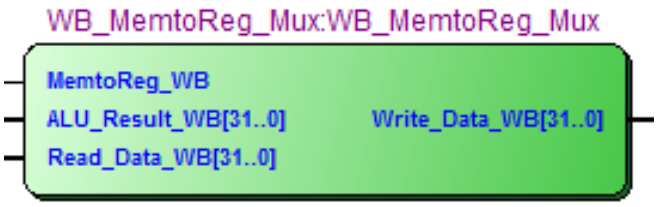
- // memory stage to write back stage pipeline, passing data on positive edge of global clock

```
module MEM_WB_Pipeline_Stage(
                        input           Clk,

                        input           RegWrite_MEM,
                        input           MemtoReg_MEM,
                        input [31:0]    Read_Data_MEM,
```

```
                input [31:0] ALU_Result_MEM,
                input [4:0]  Write_Register_MEM,

                        input [31:0]     Instruction_MEM,

                output reg      RegWrite_WB,
                output reg      MemtoReg_WB,
                output reg [31:0] Read_Data_WB,
                output reg [31:0] ALU_Result_WB,
                output reg [4:0]  Write_Register_WB,

                        output reg [31:0]        Instruction_WB
                );

initial begin
        RegWrite_WB         <= 0;
        MemtoReg_WB             <= 0;
        Read_Data_WB            <= 32'd0;
        ALU_Result_WB           <= 32'd0;
        Write_Register_WB   <= 5'd0;

        Instruction_WB      <= 32'd0;
    end

always@(posedge Clk) begin
        RegWrite_WB         <= RegWrite_MEM;
        MemtoReg_WB             <= MemtoReg_MEM;
        Read_Data_WB            <= Read_Data_MEM;
        ALU_Result_WB           <= ALU_Result_MEM;
        Write_Register_WB   <= Write_Register_MEM;
```

```
                Instruction_WB <= Instruction_MEM;
        end //always

endmodule // MEM_WB_Pipeline_Stage
```

Write back stage




```
// Choses which one of ALU_Result_WB and Read_Data_WB need to be writen back to register.
// When Write_Data_WB is high read data from memory is forwarded to the output.
// selection signe MemtoReg_WB is originally from ID_Control

module WB_MemtoReg_Mux(
                input [31:0]  ALU_Result_WB,
                input [31:0]  Read_Data_WB,
                input         MemtoReg_WB,
                output [31:0] Write_Data_WB
                );

  assign Write_Data_WB = MemtoReg_WB ? Read_Data_WB : ALU_Result_WB;

endmodule // WB_MemtoReg_Mux
```

4. Hazard Handling Unit

Though pipelining increase the overall performance of the processor, it create hazards that have to be handled.

Three types of hazard:

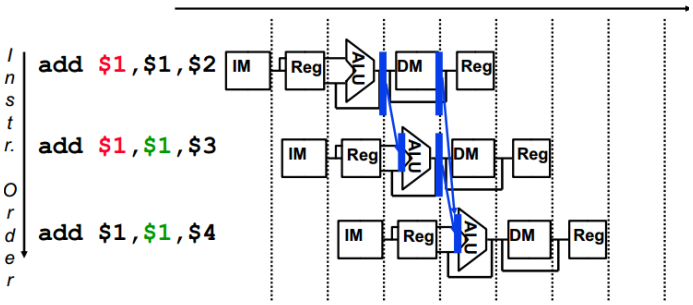
- Structure hazard: attempt to use the same resource by two different instructions at the same time. This hazard is happening is our five stage pipelined mips.
- Data hazard:
  - Read After Write (RAW)
  - Mem to Mem copy
  - Load use hazard
- Control hazard:
  - Branch
  - Jump

- Inputs and outputs:

```
module Hazard_Handling_Unit(
    input   [4:0]           IF_ID_Reg_Rs,
    input   [4:0]           IF_ID_Reg_Rt,

    input                   ID_Branch,
    input                   ID_EX_MemRead,
    input                   ID_EX_RegWrite,
    input                   ID_EX_MEMtoReg,
    input [4:0]             ID_EX_Reg_Rs,
    input [4:0]             ID_EX_Reg_Rt,
    input [4:0]             ID_EX_Reg_Rd,

    input                   EX_MEM_RegWrite,
    input                   EX_MEM_MemWrite,
    input [4:0]             EX_MEM_Reg_Rs,
    input [4:0]             EX_MEM_Reg_Rt,
    input [4:0]             EX_MEM_Reg_Rd,

    input                   MEM_WB_MemtoReg,
```

```
    input                   MEM_WB_RegWrite,
    input [4:0]             MEM_WB_Reg_Rd,
    input [4:0]             MEM_WB_Reg_Rt,

    output  [1:0]           ForwardA_EX,
    output  [1:0]           ForwardB_EX,
    output                  Forward_Mem_to_Mem,
    output                  PC_Enable,
    output                  IF_ID_Pipeline_Enable,
    output                  ID_Control_NOP,
    output  [1:0]           ID_Register_Write_to_Read,
    output                  ForwardC,
    output                  ForwardD
);
```

- DATA HAZARD - Read After Write (RAW)



1. EX Forward Unit:

```
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
      ForwardA = 10
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
```

2. MEM Forward Unit:

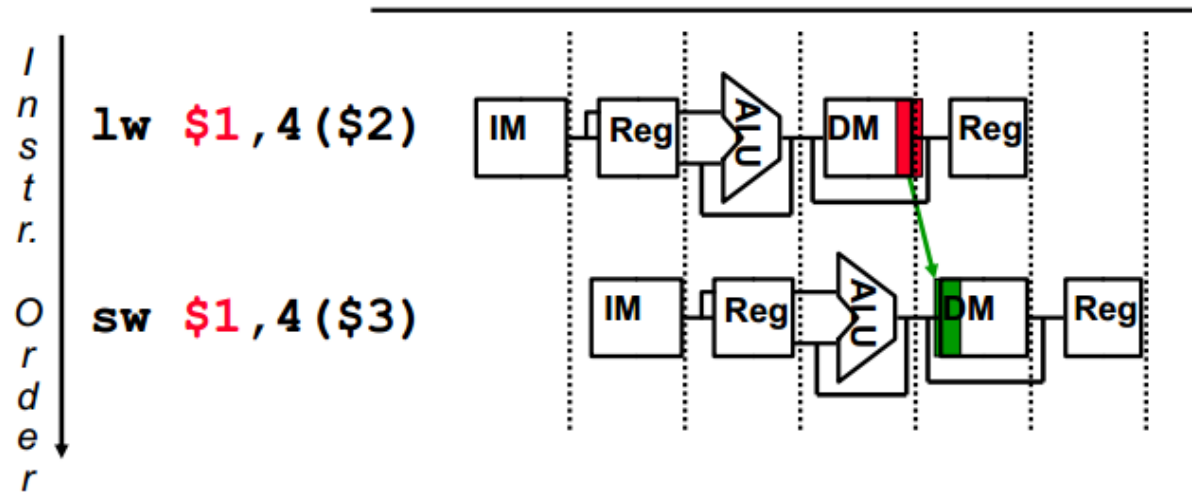
```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (EX/MEM.RegisterRd != ID/EX.RegisterRs)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
      ForwardA = 01

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (EX/MEM.RegisterRd != ID/EX.RegisterRt)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
      ForwardB = 01
```

- ForwardA_EX and ForwardB_EX: handles data hazard, they forwards ALU_Result_MEM to mux before ALU if needed, which controls the data been feed into ALU.

Verilog implement:

```
// DATA HAZARD
wire Data_Hazard_temp_1;
wire Data_Hazard_temp_2;
wire Data_Hazard_temp_3;
assign Data_Hazard_temp_1 = ( EX_MEM_RegWrite & (EX_MEM_Reg_Rd != 5'd0) );                 //common logic temp
assign Data_Hazard_temp_2 = ( MEM_WB_RegWrite & (MEM_WB_Reg_Rd != 5'd0) );                 //common logic temp
assign Data_Hazard_temp_3 = ( MEM_WB_MemtoReg & ID_EX_RegWrite & (MEM_WB_Reg_Rt != 5'd0) ); //common logic temp
assign ForwardA_EX = { ( Data_Hazard_temp_1 & (EX_MEM_Reg_Rd == ID_EX_Reg_Rs) )  //EX forward
                      ,( ( Data_Hazard_temp_2 & (EX_MEM_Reg_Rd != ID_EX_Reg_Rs) & (MEM_WB_Reg_Rd == ID_EX_Reg_Rs) ) //MEM forward
                         | ( Data_Hazard_temp_3 & (MEM_WB_Reg_Rt == ID_EX_Reg_Rs) )
                    )};
assign ForwardB_EX = { ( Data_Hazard_temp_1 & (EX_MEM_Reg_Rd == ID_EX_Reg_Rt) )  //EX forward
                      ,( ( Data_Hazard_temp_2 & (EX_MEM_Reg_Rd != ID_EX_Reg_Rt) & (MEM_WB_Reg_Rd == ID_EX_Reg_Rt) ) //MEM forward
                         | ( Data_Hazard_temp_3 & (MEM_WB_Reg_Rt == ID_EX_Reg_Rt) )
                    )};
```

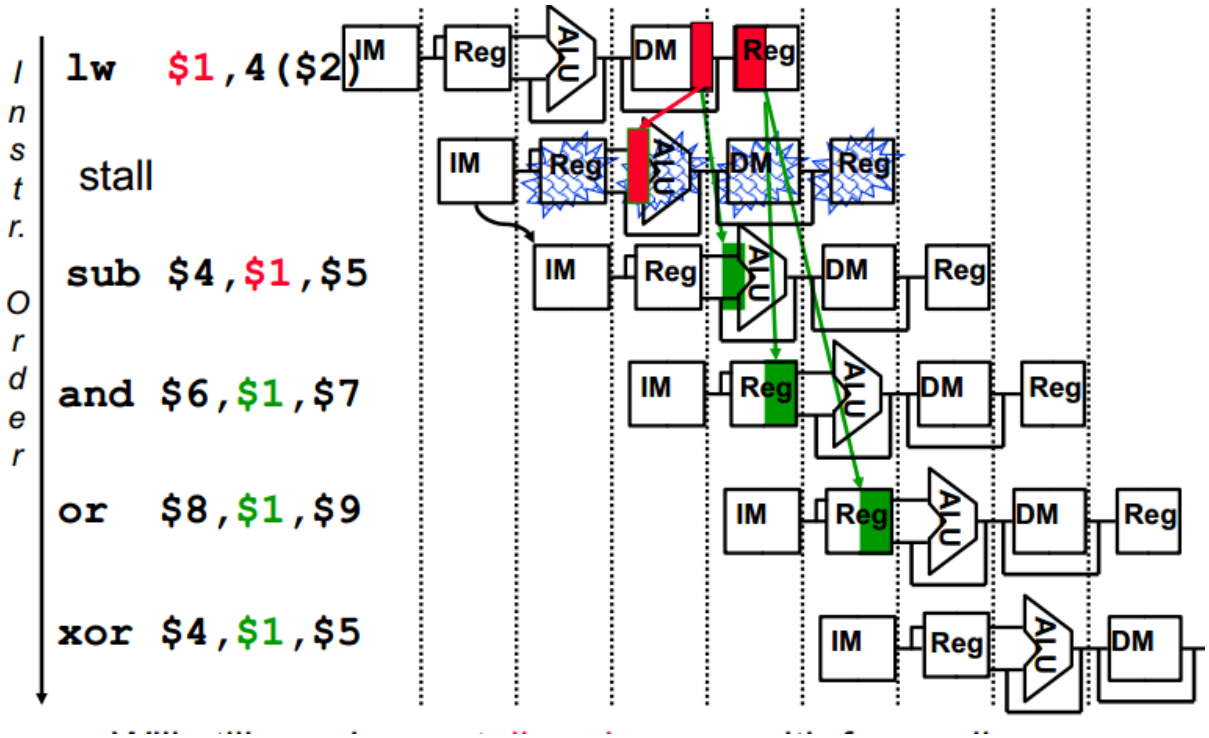
- DATA HAZARD – Mem to Mem copy
  - Forward_Mem_to_Mem: handles mem to mem copy hazard, it forwards Read_Data_WB to Write_Data_MUX_MEM if needed, which controls data been written to mem.



Verilog implement:

```
// MEM OT MEM COPY
assign Forward_Mem_to_Mem = ( (EX_MEM_Reg_Rt == MEM_WB_Reg_Rt) & MEM_WB_MemtoReg & EX_MEM_MemWrite );
```

- DATA HAZARD – Load use hazard



- PC_Enable, IF_ID_Pipeline_Enable and ID_Control_NOP:
- these signals will stall PC and IF/ID pipeline, and simultaneously force a nop is ID_Control signals

- ID_Register_Write_to_Read : handles when the clock delay due to write register is not affordable,
- it forwards the Write_Data_WB(data will be written to register) to Read_Data(read data from register) when needed.

Verilog implement:

```
// LOAD-USE DATA HAZARD
assign PC_Enable = !( (ID_EX_MemRead & ( (ID_EX_Reg_Rt == IF_ID_Reg_Rs) | (ID_EX_Reg_Rt == IF_ID_Reg_Rt) ))
                    | ( ID_Branch & ID_EX_RegWrite & ((ID_EX_Reg_Rd == IF_ID_Reg_Rs)|(ID_EX_Reg_Rd == IF_ID_Reg_Rt)) ) );
assign IF_ID_Pipeline_Enable = PC_Enable;
assign ID_Control_NOP = !PC_Enable;

wire Load_use_temp_1;
wire Load_use_temp_2;
assign Load_use_temp_1 = ( MEM_WB_MemtoReg & (MEM_WB_Reg_Rt != 5'd0) ); //common logic temp
assign Load_use_temp_2 = ( MEM_WB_RegWrite & !MEM_WB_MemtoReg );        //common logic temp

assign ID_Register_Write_to_Read = {( (Load_use_temp_1 & (MEM_WB_Reg_Rt == IF_ID_Reg_Rt)) | (Load_use_temp_2 & (MEM_WB_Reg_Rd == IF_ID_Reg_Rt)) )
                                   ,( (Load_use_temp_1 & (MEM_WB_Reg_Rt == IF_ID_Reg_Rs)) | (Load_use_temp_2 & (MEM_WB_Reg_Rd == IF_ID_Reg_Rs)) )};
```

- Control Hazard – branch

```
// BRANCH HAZARD
assign ForwardC = ( ID_Branch & EX_MEM_RegWrite & (EX_MEM_Reg_Rd != 5'd0) & (EX_MEM_Reg_Rd == IF_ID_Reg_Rs) );
assign ForwardD = ( ID_Branch & EX_MEM_RegWrite & (EX_MEM_Reg_Rd != 5'd0) & (EX_MEM_Reg_Rd == IF_ID_Reg_Rt) );

endmodule // Hazard_Handling_Unit
```

5. Assembly code

Some register value is initialised to certain value to improved performance.

R0		***0***
R1		***1***
R2		***15***
R3		***14***
R4		***224***
R5	Matrix A_pos	***100***
R6	Matrix B_pos	***400***
R7	Matrix C_pos	***700***
R8	temp	
R9		
R10	data A	
R11	data B	
R12	mul result	
R13	mul result adder	
R14	k	
R15	d	
R16	c	
R17		***400***
R18		
R19		
R20		
R21		

Both input matrix A, matrix B and result matrix C are stored in memory in column increment method.

Assembly code:

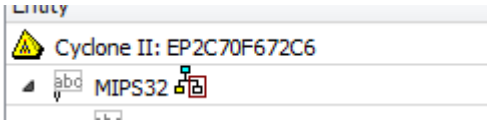
multi:	@4
lw $10,0($5)	8caa0000
lw $11,0($6)	8ccb0000
mul $12,$10,$11	014b6021
add $13,$12,$13	018d6820
add $5,$1,$5	00252820
add $6,$2,$6	00463020
add $14,$1,$14	002e7020
beq $14,$2,4	11c20001
j multi	08000001
j switch_d	08000020

switch_d:	@0x80
sw $13,0($7)	aced0000
add $7,$1,$7	00273820
sub $5,$5,$2	00a22822
sub $6,$6,$4	00c43022
and $13,$0,$0	00006824
and $14,$0,$14	000e7024
add $15,$1,$15	002f7820
beq $15,$2,4	11e20001
j multi	08000001
j switch_c	08000030

switch_c:	@0xC0
add $16,$1,$16	00308020
add $5,$2,$5	00452820
and $6,$17,$17	02313024
and $15,$0,$0	00007824
beq $16,$2,4	12020001
j multi	08000001
j exit	080000FF
exit:	@FF
beq $0,$0,-4	1000fffe

6. Result

Fmax



Slow Model Fmax Summary

	Fmax	Restricted Fmax	Clock Name	Note
1	71.56 MHz	71.56 MHz	clock1	

Total Clock Cycle

Number of clock cycle required:

= multiply + switch + jump

= 11x3375 + (15*7 + 10*225) + (15+ 225)

= 39720

Critical path

Slow Model Setup: 'clock1'

	Slack	From Node	To Node	Data Delay
1	66.025	ID_EX_Pipeline_Stage:ID_EX_Pipeline_Stage|Instruction_EX[20]	EX_MEM_Pipeline_Stage:EX_MEM_Pipeline_Stage|ALU_Result_MEM[31]	13.982
2	66.142	ID_EX_Pipeline_Stage:ID_EX_Pipeline_Stage|Instruction_EX[20]	EX_MEM_Pipeline_Stage:EX_MEM_Pipeline_Stage|ALU_Result_MEM[28]	13.897
3	66.151	ID_EX_Pipeline_Stage:ID_EX_Pipeline_Stage|Instruction_EX[20]	EX_MEM_Pipeline_Stage:EX_MEM_Pipeline_Stage|ALU_Result_MEM[27]	13.860
4	66.208	ID_EX_Pipeline_Stage:ID_EX_Pipeline_Stage|Instruction_EX[20]	EX_MEM_Pipeline_Stage:EX_MEM_Pipeline_Stage|ALU_Result_MEM[30]	13.838
5	66.319	MEM_WB_Pipeline_Stage:MEM_WB_Pipeline_Stage|Instruction_WB[14]	EX_MEM_Pipeline_Stage:EX_MEM_Pipeline_Stage|ALU_Result_MEM[31]	13.688
6	66.370	MEM_WB_Pipeline_Stage:MEM_WB_Pipeline_Stage|Instruction_WB[16]	EX_MEM_Pipeline_Stage:EX_MEM_Pipeline_Stage|ALU_Result_MEM[31]	13.639
7	66.436	MEM_WB_Pipeline_Stage:MEM_WB_Pipeline_Stage|Instruction_WB[14]	EX_MEM_Pipeline_Stage:EX_MEM_Pipeline_Stage|ALU_Result_MEM[28]	13.603
8	66.416	ID_EX_Pipeline_Stage:ID_EX_Pipeline_Stage|Instruction_EX[18]	EX_MEM_Pipeline_Stage:EX_MEM_Pipeline_Stage|ALU_Result_MEM[31]	13.592
9	66.445	MEM_WB_Pipeline_Stage:MEM_WB_Pipeline_Stage|Instruction_WB[14]	EX_MEM_Pipeline_Stage:EX_MEM_Pipeline_Stage|ALU_Result_MEM[27]	13.566
10	66.487	MEM_WB_Pipeline_Stage:MEM_WB_Pipeline_Stage|Instruction_WB[16]	EX_MEM_Pipeline_Stage:EX_MEM_Pipeline_Stage|ALU_Result_MEM[28]	13.554

7. Discussion

Optimisation done:

- By using the DE-2 FPGA embedded multiplier, the multiplication process speed in improved comparing to implement by logics.
- Since we assume result matrix elements will not exceed 32 bits, there is no need to store the multiplication result in upper and lower 32 bits.
- Data forwarding is used to solve hazards, instead of using stall. It improves the overall performance.

Drawback and improvement:

- The overall clock cycles used is relatively higher than expected. Implementing the branch and jump prediction will reduce stall after branch or jump.
- Register and instruction cache can shorten the acquire time, hence improve overall performance.

Appendix – instruction memory, register, data memory initialise file

Instruction memory

```
0
0
0
0
@4      //multi:
8caa0000        //lw $10,0($5)
0
0
0
8ccb0000        //lw $11,0($6)
0
0
0
014b6021        //mul $12,$10,$11
0
0
0
018d6820        //add $13,$12,$13
0
0
0
00252820        //add $5,$1,$5
0
0
0
00463020        //add $6,$2,$6
0
0
0
```

```
002e7020        //add $14,$1,$14
0
0
0
11c20001        //beq $14,$2,4
0
0
0
08000001        //j multi
0
0
0
08000020        //j switch_d

@80     // switch_d:
aced0000        //sw $13,0($7)
0
0
0
00273820        //add $7,$1,$7
0
0
0
00a22822        //sub $5,$5,$2
0
0
0
00c43022        //sub $6,$6,$4
0
0
0
```

```
00006824        //and $13,$0,$0
0
0
0
000e7024        //and $14,$0,$14
0
0
0
002f7820        //add $15,$1,$15
0
0
0
11e20001        //beq $15,$2,4
0
0
0
08000001        //j multi
0
0
0
08000030        //j switch_c

@C0     //switch_c:
00308020        //add $16,$1,$16
0
0
0
00452820        //add $5,$2,$5
0
0
0
```

```
02313024        //and $6,$17,$17
0
0
0
00007824        //and $15,$0,$0
0
0
0
12020001        //beq $16,$2,4
0
0
0
08000001        //j multi
0
0
0
080000FF        //j exit

@FF     //exit:
1000fffe        //beq $0,$0,-4
```

Register

0

1 //r1 1

F //r2 15

E //r3 14

E0 //r4 224

64 //r5 matrixA pos

190 //r6 matrixB pos

2BC //r7 matrixC pos

0 //r8 temp

0 //r9

0 //r10dataA

0 //r11dataB

0 //r12mul result

0 //r13mul result adder

0 //r14 k

0 //r15 d

0 //r16 c

190 //r17 400

Data Memory

@64 //100

A8

19A

1F0

79

FFFFFE78

FFFFFF92

FFFFFE98

1A2

10C

62

FFFFFE44

11F

FFFFFEC8

FFFFFE21

FFFFFFAF

30

159

163

5C

112

FFFFFFD0

12E

CA

FFFFFEA9

FFFFFEDB

D2

14C

FFFFFF31

E6

FFFFFEF0

FFFFFF24

102

1E8

FFFFFF80

FFFFFFD4

FFFFFEBA

FFFFFEA9

FFFFFE1E

FFFFFE51

24

180

F1

1E7

FFFFFF69

1D3

FFFFFE63

FFFFFF09

FFFFFF61

FFFFFE17

E7

FFFFFE45

DE

FFFFFFEC

9

FFFFFE39

FFFFFF1C

12C

FFFFFEA9

90

FFFFFF7C

153

1C9

FFFFFF1C

FFFFFEA3

FFFFFE6D

123

FFFFFE50

B5

FFFFFE5F

FFFFFFB9

C5

FFFFFF9E

11C

116

FFFFFFF7

FFFFFF42

163

FFFFFF44

6

FFFFFFC1

135

FFFFFF93

FFFFFF9C

FFFFFFC0

FFFFFEC3

FFFFFE8E

FFFFFE85

FFFFFF5D

B5

FFFFFF11

65

15E

FFFFFF58

126

3A

FFFFFFDA

143

FFFFFE9D

FFFFFF3F

FFFFFF4F

FFFFFF7F

FFFFFEA3

FFFFFFE5

FFFFFFF1

4E

FFFFFFE0

11B

31

10C

16C

168

1E

3E

FFFFFF84

1C0

157

144

FFFFFE66

13F

14

A2

FFFFFFE5

FFFFFF24

1BC

1A5

80

1CA

1CC

1FB

163

C8

FFFFFE1F

FFFFFE3D

FFFFFE4D

1EA

FFFFFF0B

167

1A9

12E

FFFFFF67

FFFFFEB8

FFFFFF44

1D6

17A

FFFFFF07

C7

2D

172

FFFFFE1D

A4

FFFFFF23

1B7

AE

FFFFFFD1

FFFFFF7D

FFFFFE1E

50

1F5

1CA

FFFFFF82

169

14B

FFFFFE48

FFFFFFBB

162

C3

FFFFFE18

124

FFFFFF60

FFFFFE68

FFFFFE84

FFFFFEC2

158

FFFFFF04

EF

FFFFFF2D

1CB

187

FFFFFE92

FFFFFE59

1E4

EA

FFFFFEBB

FFFFFE88

A1

FFFFFF43

FFFFFF91

9D

FFFFFE31

120

9A

FFFFFFD7

FFFFFF93

13B

FFFFFFB1

FFFFFF21

FFFFFFDD

FFFFFE90

3E

FFFFFEE0

ED

1FE

17A

153

FFFFFE0B

FFFFFF31

FFFFFE47

FFFFFFC2

FFFFFE01

FFFFFE2A

16A

5D

FFFFFE39

FFFFFED4

177

117

FFFFFF5F

174

C1

FFFFFEE3

1C7

16

FFFFFF67

FFFFFE62

68

@190 //400

ED

A2

FFFFFE01

FFFFFEE1

FFFFFEA6

FFFFFF3C

1C1

FFFFFF92

73

FFFFFEBD

FFFFFE4D

23

FFFFFF18

18F

5

BB

FFFFFEB5

B8

FFFFFF46

FFFFFFB0

1E5

FFFFFE3B

FFFFFFD9

100

24

FFFFFEAE

12C

FFFFFF6E

FFFFFF74

42

FFFFFE3E

94

FFFFFF21

FFFFFEE0

173

173

FFFFFEFF

F3

52

4B

B0

58

11F

121

FFFFFE46

B4

FFFFFE24

149

FFFFFE1C

A0

7A

FFFFFF80

180

1FA

6B

5C

186

FFFFFEC1

FFFFFF50

FFFFFEA7

FFFFFF3A

121

FFFFFF2D

B8

1D9

184

FFFFFFA7

1E5

FFFFFEA9

FFFFFE29

AB

FFFFFFC5

FFFFFFA0

FFFFFEA0

A

FFFFFF64

FFFFFE01

FFFFFF8F

FFFFFF25

FFFFFF95

FFFFFF3A

FFFFFEA9

E5

D5

FFFFFF14

FFFFFFCD

FFFFFE5F

C6

F5

2A

E7

FFFFFF89

1CD

FFFFFE1F

FFFFFFA5

81

1C4

7D

FFFFFEAA

FFFFFEB7

12A

1B7

151

86

A

FFFFFE6D

B0

157

FFFFFF23

11D

89

FFFFFE83

15E

FFFFFE51

1C

FFFFFFD2

FFFFFFAC

2F

66

FFFFFE7C

FFFFFF02

FFFFFE28

18F

FFFFFE73

19B

FFFFFF34

105

FFFFFF83

89

F0

193

1B

1A6

FFFFFEF6

FFFFFE9F

FFFFFFC1

FFFFFFBD

1BA

FFFFFE49

AD

54

8F

198

34

1AB

FFFFFE4A

C3

1AB

FFFFFF21

FFFFFFA2

59

FFFFFEB2

FFFFFE84

52

FFFFFEAF

FFFFFE6E

FFFFFF39

127

116

FFFFFE1F

8A

FFFFFE37

79

131

C6

FFFFFF3A

55

FFFFFF64

FFFFFE0B

FFFFFF47

131

FFFFFE76

1A2

FFFFFE38

14B

13C

FFFFFF5C

1A4

59

11F

FFFFFFBD

FFFFFEC3

DA

15F

FFFFFF15

180

1D2

FFFFFFAF

FFFFFE2F

163

FFFFFF48

12B

193

FFFFFF34

8B

FFFFFF3D

FFFFFF88

80

FFFFFE43

FFFFFEEC

97

14B

FFFFFE48

54

FFFFFFE0

24

FFFFFE06

FFFFFF61

FFFFFF2F

53

1A9

187

155

FFFFFFEF

CF

FFFFFE38

FFFFFE9B

1A1

10C

FFFFFF7B

176

FFFFFE37

1FC

FFFFFEE6

1E6