# 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.

# Development process

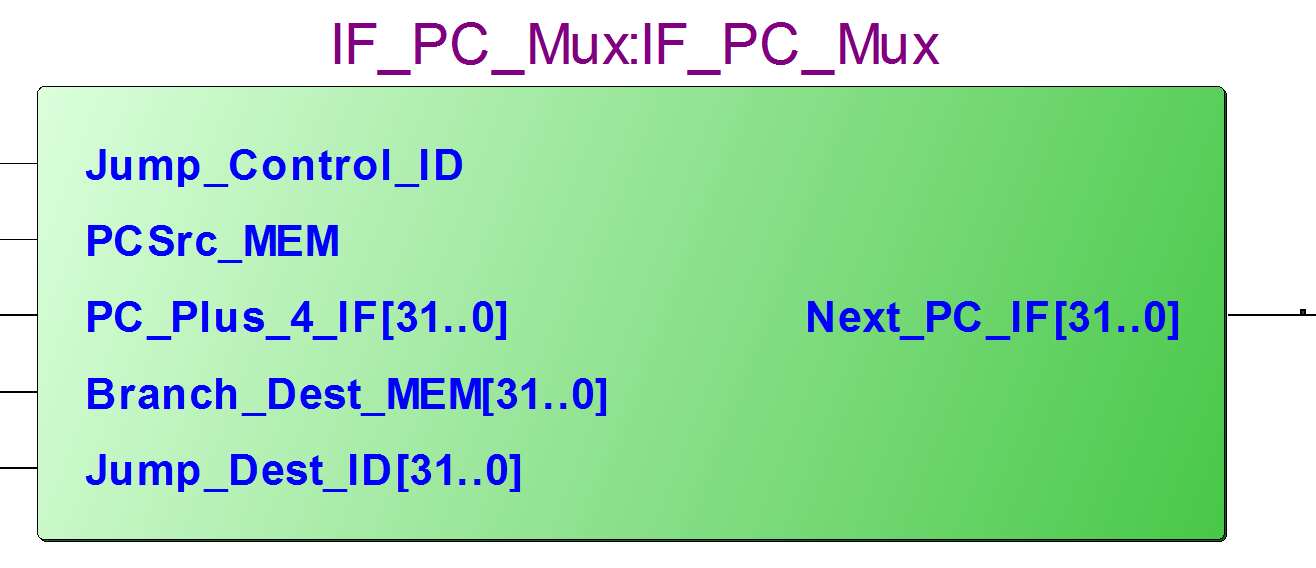


# Datapath

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

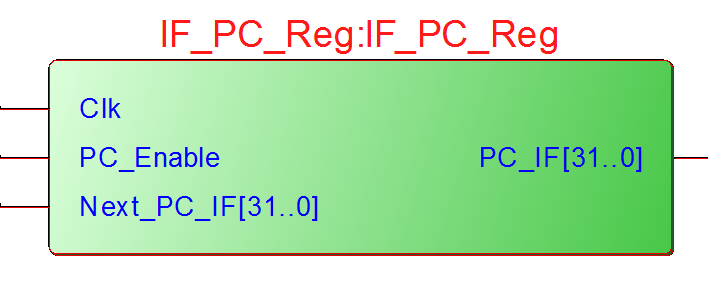
input Jump\_Control\_ID,

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)

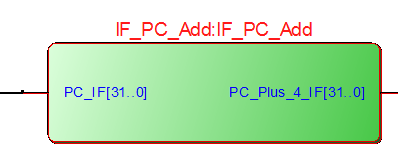
begin

PC\_IF <= Next\_PC\_IF;

end

end

endmodule // IF\_PC\_Reg



// IF\_PC\_Add add the pc by digital four.

module IF\_PC\_Add(

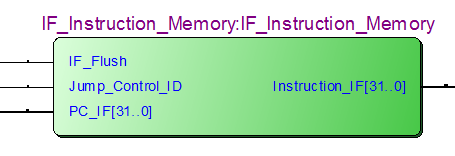
input [31:0] PC\_IF,

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

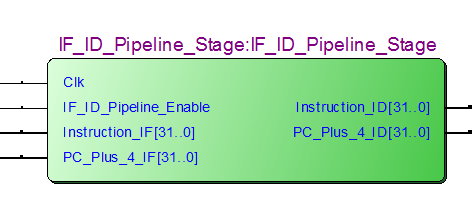
$readmemh("instruction\_memory.list", Instruction\_Memory);

//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;

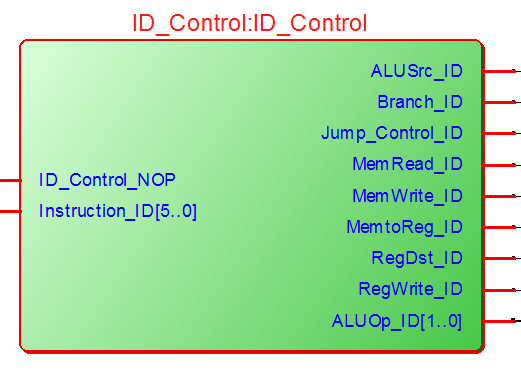
PC\_Plus\_4\_ID<=PC\_Plus\_4\_IF;

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;

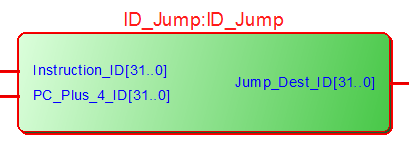
ALUSrc\_ID <= 1'b0;

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(

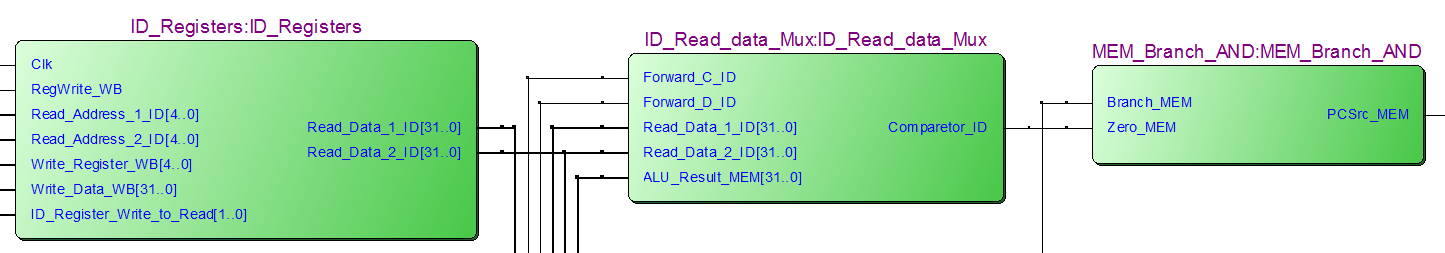
input [31:0] Instruction\_ID,

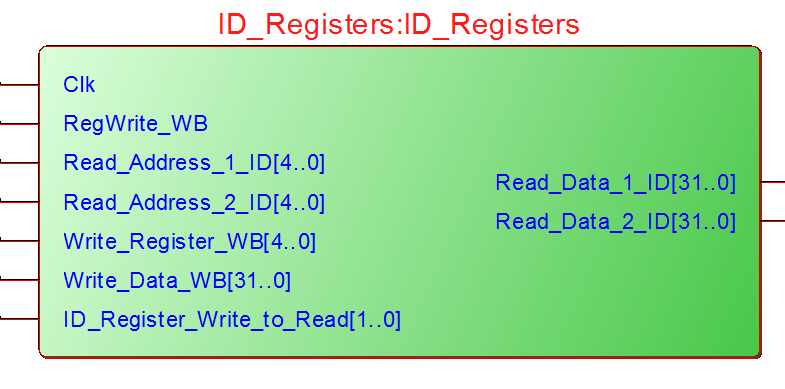
input [31:0] PC\_Plus\_4\_ID,

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

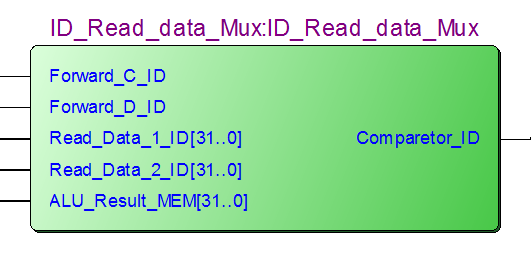
if( RegWrite\_WB & (Write\_Register\_WB != 5'd0) ) begin

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;

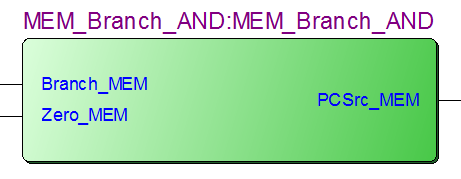
wire [31:0] Read\_Data\_2\_MUX\_ID;

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

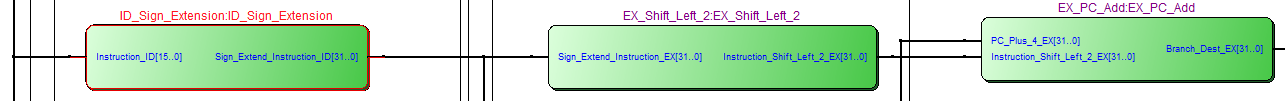
input Zero\_MEM,

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(

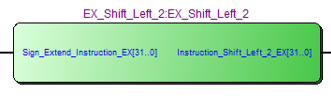
input [15:0] Instruction\_ID,

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(

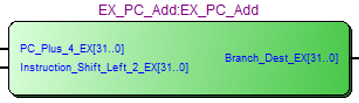
input [31:0] Sign\_Extend\_Instruction\_EX,

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

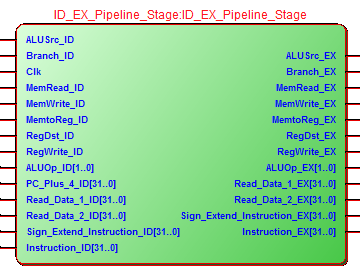
input [31:0] Instruction\_Shift\_Left\_2\_EX,

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;

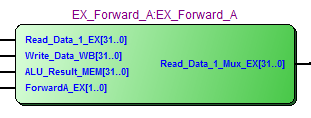
Sign\_Extend\_Instruction\_EX <= Sign\_Extend\_Instruction\_ID;

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;

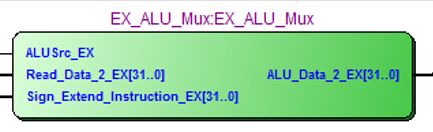
Third: Read\_Data\_1\_Mux\_EX <= ALU\_Result\_MEM;

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,

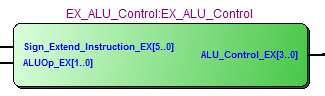
input ALUSrc\_EX,

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;

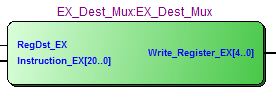
unknown: ALU\_Control\_EX <= ALUx;

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,

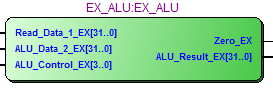
input RegDst\_EX,

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;

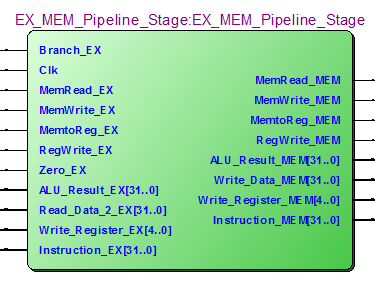
default: ALU\_Result\_EX <= 32'bx; // control = ALUx | \*

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;

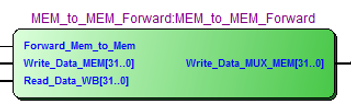
Write\_Register\_MEM<= Write\_Register\_EX;

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;

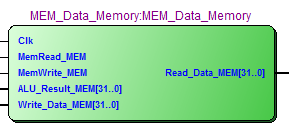
Second: Write\_Data\_MUX\_MEM <= Read\_Data\_WB;

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)

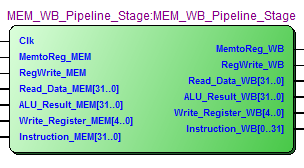
begin

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;

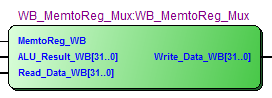
Write\_Register\_WB <= Write\_Register\_MEM;

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

# 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,

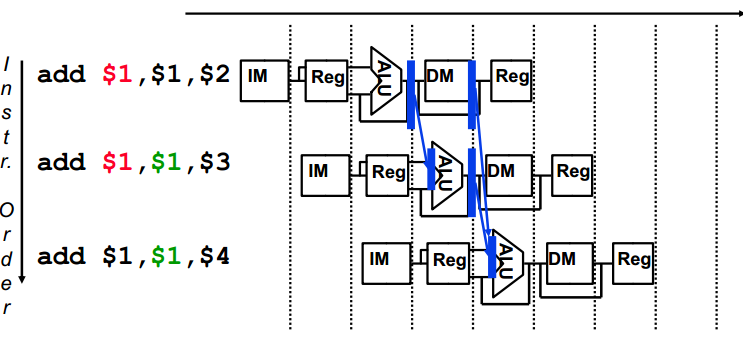
output [1:0] ID\_Register\_Write\_to\_Read,

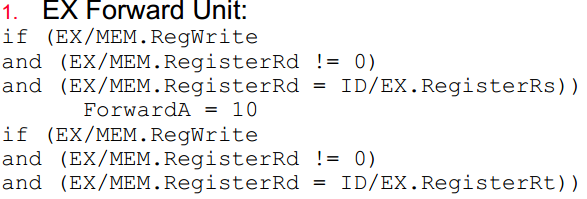
output ForwardC,

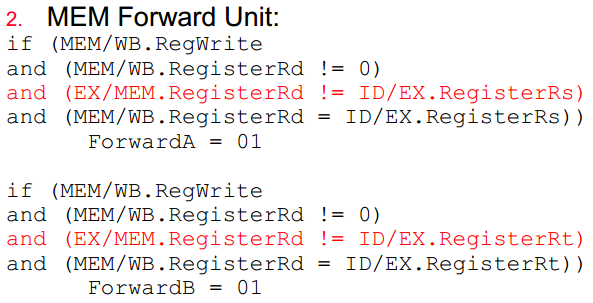
output ForwardD

);

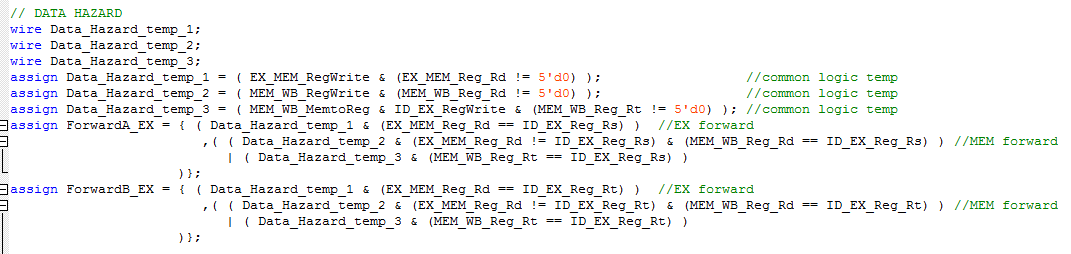
* DATA HAZARD - Read After Write (RAW)



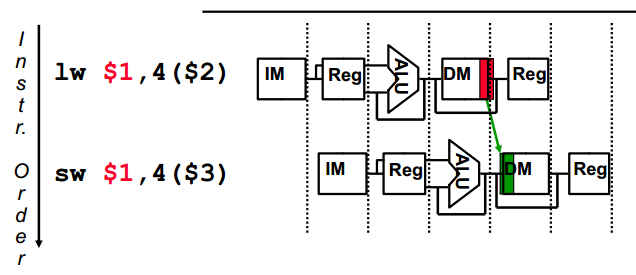




* ForwardA\_EX and ForwardB\_EX: handles data hazard, they forwards ALU\_Result\_MEM to mux before ALU if needed, which controls the data been feed into ALU.

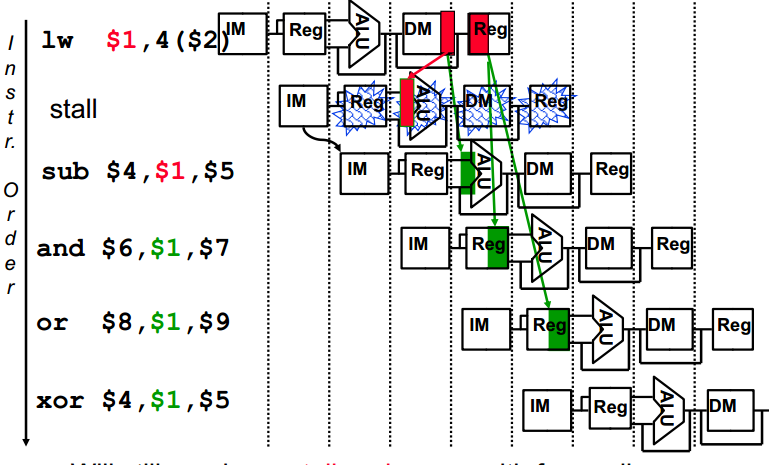
Verilog implement:

* DATA HAZARD – 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.

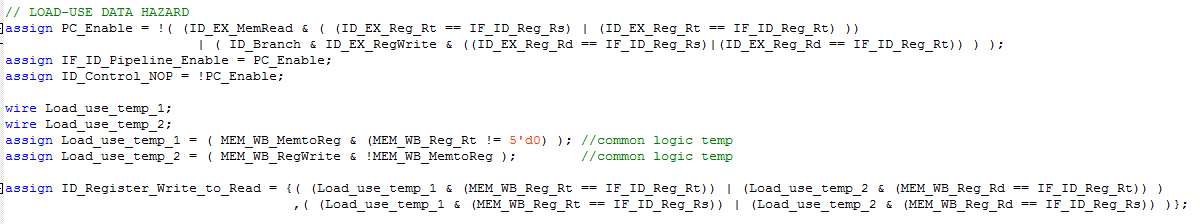


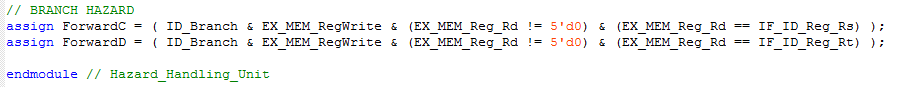
Verilog implement:

* 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:

* Control Hazard – branch 

# Assembly code

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



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

Assembly code:



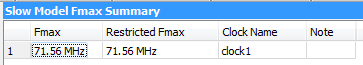




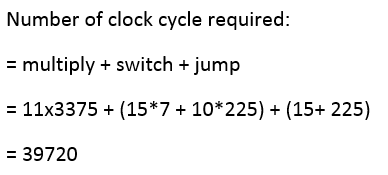
# Result

Fmax

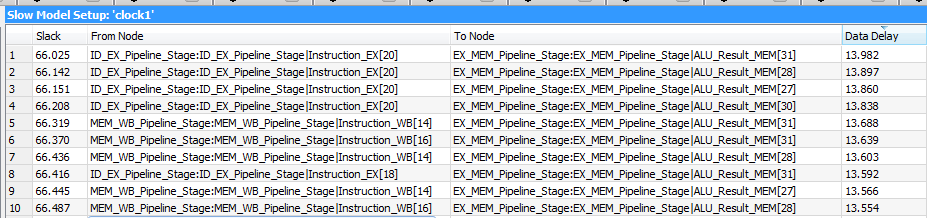




Total Clock Cycle



Critical path



# 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