Information Technology Institute

Digital Design and Computer Architecture

RISC-V: From Single-Cycle to Pipelined Implementation

Project implemented and documented by Ali Yehia

Under supervision of Prof. Hesham Omran

Contents

1. RISC-V Single Cycle Architecture

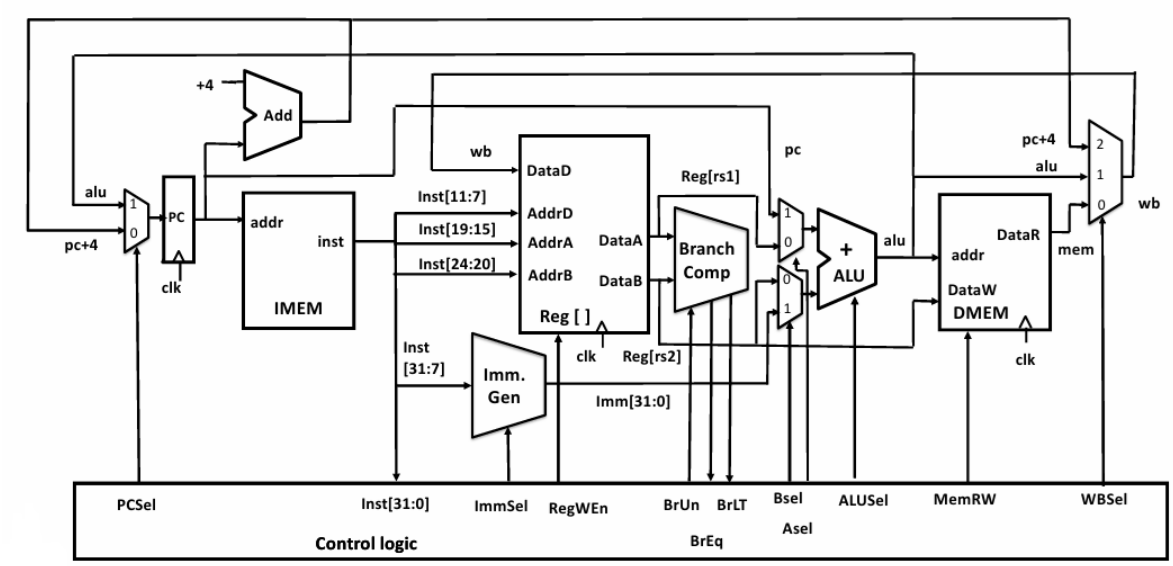


Figure 1-1: RISC-V single cycle. [1]

1.1 Data Path

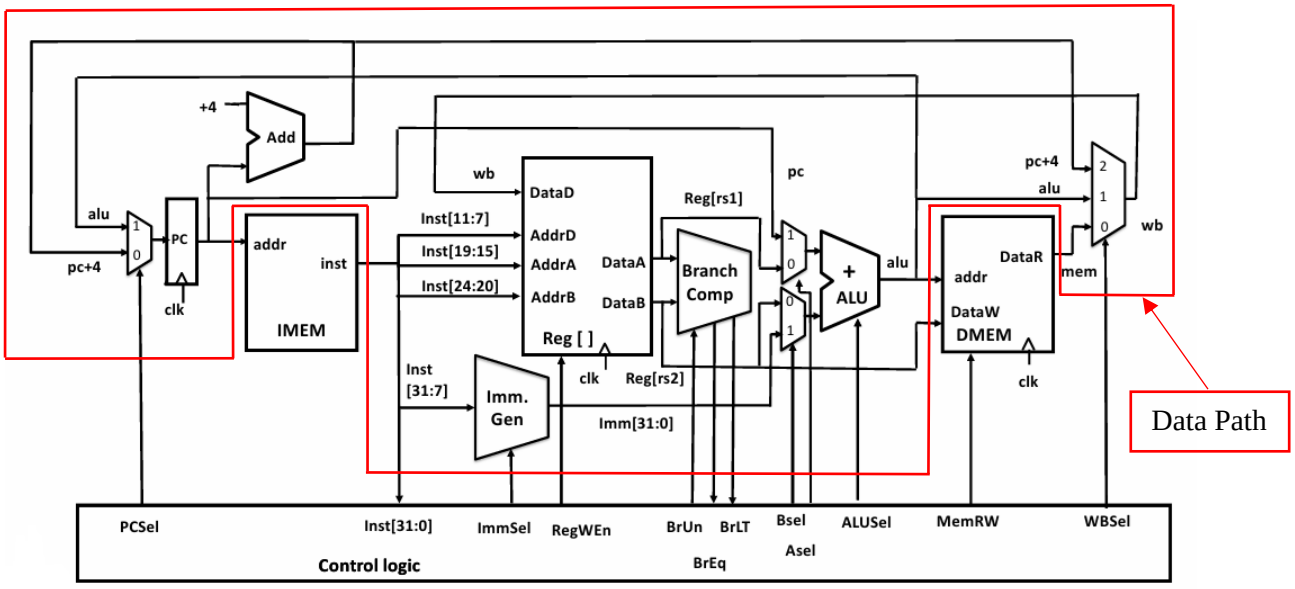


Figure 1-2: RISC-V datapath. Adapted from [1]

1.1.1 Mux2x1

```
`timescale 1ps/1fs

module Mux2x1 #(parameter DATA_WIDTH = 32)(
    input   wire                        Sel,
    input   wire    [DATA_WIDTH-1:0]    In0,
    input   wire    [DATA_WIDTH-1:0]    In1,
    output  reg     [DATA_WIDTH-1:0]    Out
    );

always@(*)
 begin
  #10; // Simple Logic Delay
  if(Sel)
   Out = In1;
  else
   Out = In0;
 end


endmodule
```

1.1.2 PC

```
`timescale 1ps/1fs

module PC #(parameter DATA_WIDTH = 32)(
    input   wire                        clk,
    input   wire                        rst_n,
    input   wire    [DATA_WIDTH-1:0]    PC_In,
    output  reg     [DATA_WIDTH-1:0]    PC_Out
    );

always@(posedge clk or negedge rst_n)
 begin
  //#10; // Simple Logic Delay
  if(!rst_n)
   PC_Out <= 'b0;
  else
   PC_Out <= PC_In;
 end


endmodule
```

1.1.3 Add

```
`timescale 1ps/1fs

module Add #(parameter DATA_WIDTH = 32)(
    input   wire    [DATA_WIDTH-1:0]    PC,
    output  reg     [DATA_WIDTH-1:0]    PC_Plus_4
    );

always@(*)
 begin
    #10 PC_Plus_4 = PC + 3'd4; // Simple Logic Delay
 end

endmodule
```

1.1.4 Register File

```
`timescale 1ps/1fs

module RegisterFile #(parameter DATA_WIDTH = 32) (
    input   wire                    clk,
    input   wire [DATA_WIDTH-1:0]   DataD,          // Data to be written to the destination register
    input   wire [4:0]              AddrD,          // Destination register address
    input   wire [4:0]              AddrA,          // Source register 1 address
    input   wire [4:0]              AddrB,          // Source register 2 address
    input   wire                    RegWEn,
    output  wire [DATA_WIDTH-1:0]   DataA,          // Data from source register 1
    output  wire [DATA_WIDTH-1:0]   DataB           // Data from source register 2
    );


reg [DATA_WIDTH-1:0] RegFile [0:31];


// Read ports and hard wire x0 to '0'
assign DataA = (AddrA == 0) ? {DATA_WIDTH{1'b0}} : RegFile[AddrA];
assign DataB = (AddrB == 0) ? {DATA_WIDTH{1'b0}} : RegFile[AddrB];

// Write port (write on the negedge clock "later needed in pipeline") and prevent writing to x0
always @(negedge clk)
begin
 #50; // Major Stage Delay
 if (RegWEn && AddrD != 5'b0)
  RegFile[AddrD] <= DataD;
end

endmodule
```

1.1.5 Immediate Generator

```
`timescale 1ps/1fs

/*
  Truth Table for ImmSel:
  +-------------+---------+----------------------+
  | ImmSel Value| Type    | Description          |
  +-------------+---------+----------------------+
  |    3'b001   | I-Format| Immediate            |
  |    3'b010   | S-Format| Store Instructions   |
  |    3'b011   | B-Format| Branch Instructions  |
  |    3'b100   | J-Format| Jump                 |
  |    3'b101   | U-Format| Upper Immediate      |
  +-------------+---------+----------------------+
*/

module ImmGen (
    input  wire [31:7] Instr,
    input  wire [2:0]  ImmSel,
    output reg  [31:0] Imm
);

always @(*)
 begin
  #50; // Major Stage Delay
  casex (ImmSel)
   3'b001 : Imm = {{21{Instr[31]}}, Instr[30:20]};                          // I-type (includes JALR)
   3'b010 : Imm = {{21{Instr[31]}}, Instr[30:25], Instr[11:7]};             // S-type
   3'b011 : Imm = {{20{Instr[31]}}, Instr[7], Instr[30:25], Instr[11:8], 1'b0};   // B-type
   3'b100 : Imm = {{12{Instr[31]}}, Instr[19:12], Instr[20], Instr[30:21], 1'b0}; // J-type
   3'b101 : Imm = {{13{Instr[31]}}, Instr[30:12]};                          // U-type
   default: Imm = 32'bx;                                                    // to check in testbench for illegal cases
  endcase
 end

endmodule
```

1.1.6 Branch Comparator

```
`timescale 1ps/1fs

module BranchComp #(parameter DATA_WIDTH = 32) (
    input   wire    [DATA_WIDTH-1:0]    DataA,
    input   wire    [DATA_WIDTH-1:0]    DataB,
    input   wire                        BrUn,
    output  reg                         BrEq,
    output  reg                         BrLT
    );

always@(*)
 begin
  #50; // Major Stage Delay
  if (BrUn == 1'b1)
   begin
    if ($signed(DataA) < $signed(DataB))
     BrLT = 1'b1;
    else
     BrLT = 1'b0;
    if ($signed(DataA) == $signed(DataB))
     BrEq = 1'b1;
    else
     BrEq = 1'b0;
   end
  else
   begin
    if ((DataA) < (DataB))
     BrLT = 1'b1;
    else
     BrLT = 1'b0;
    if ((DataA) == (DataB))
     BrEq = 1'b1;
    else
     BrEq = 1'b0;
   end
 end

// assign #10 BrEq = ( ( BrUn == 1'b1 && ($signed(DataA) == $signed(DataB)) ) || ( BrUn == 1'b0 && ((DataA) == $(DataB)) ) ) ? 1'b1 : 1'b0;
// assign #10 BrLT = ( ( BrUn == 1'b1 && ($signed(DataA) < $signed(DataB)) ) || ( BrUn == 1'b0 && ((DataA) < $(DataB)) ) ) ? 1'b1 : 1'b0;

endmodule
```

1.1.7 ALU

```
`timescale 1ps/1fs

/*
  Truth Table for ALUSel:
  +-------------+-------------------------------+
  | ALUSel      | Operation                     |
  +-------------+-------------------------------+
  | 4'b0000     | Add                           |
  | 4'b1000     | Subtract                      |
  | 4'b0001     | Shift Left Logical (SLL)      |
  | 4'b0010     | Set Less Than signed   (SLT)  |
  | 4'b0011     | Set Less Than unsigned (SLTU) |
  | 4'b0100     | XOR                           |
  | 4'b0101     | Shift Right Logical (SRL)     |
  | 4'b1101     | Shift Right Arithmetic (SRA)  |
  | 4'b0110     | OR                            |
  | 4'b0111     | AND                           |
  | default     | Illegal (Till now)            |
  +-------------+-------------------------------+
*/

module ALU #(parameter DATA_WIDTH = 32) (
    input  wire [DATA_WIDTH-1:0] Src_A,
    input  wire [DATA_WIDTH-1:0] Src_B,
    input  wire [3:0]            ALUSel,
    output reg  [DATA_WIDTH-1:0] ALU_Result
    );

wire [DATA_WIDTH-1:0] Src_B_inv_or_not;
wire [DATA_WIDTH-1:0] result;

assign Src_B_inv_or_not = ALUSel[3] ? ~Src_B : Src_B;
assign result = Src_A + Src_B_inv_or_not + ALUSel[3];


always @(*)
 begin
    #50; // Major Stage Delay
    casex (ALUSel[2:0])
        3'b000  : ALU_Result = result;  // add & sub
/*
        modification is needed for ALUSel

        4'b0001: ALU_Result = Src_A << Src_B;                                       // sll
        4'b0010: ALU_Result = ($signed(Src_A) < $signed(Src_B))? 32'b1 : 32'b0;  // slt
        4'b0011: ALU_Result = (Src_A < Src_B)? 32'b1 : 32'b0;                    // sltu
        4'b0100: ALU_Result = Src_A ^ Src_B;                                        // xor
        4'b0101: ALU_Result = Src_A >> Src_B;                                       // srl
        4'b1101: ALU_Result = Src_A >>> Src_B;                                      // sra
*/
        3'b110  : ALU_Result = Src_A | Src_B;
        3'b111  : ALU_Result = Src_A & Src_B;
        default : ALU_Result = 'bx;                                                 // to check in testbench for illegal cases if happened
    endcase
 end

endmodule
```

1.1.8 Mux3x1

```
`timescale 1ps/1fs

/*
  Truth Table for WBSel:
  +-------------+---------+------------------+
  | WBSel Value | Source  | Description      |
  +-------------+---------+------------------+
  |    2'b00    | Mem     | Memory Read Data |
  |    2'b01    | ALU     | ALU Result       |
  |    2'b10    | PC+4    | Return Address   |
  +-------------+---------+------------------+
*/

module Mux3x1 #(parameter DATA_WIDTH = 32)(
    input  wire [1:0]            Sel,
    input  wire [DATA_WIDTH-1:0] In0,
    input  wire [DATA_WIDTH-1:0] In1,
    input  wire [DATA_WIDTH-1:0] In2,
    output reg  [DATA_WIDTH-1:0] Out
);

always @(*)
 begin
  #10; // Simple Logic Delay
  case (Sel)
   2'b00  : Out = In0;
   2'b01  : Out = In1;
   2'b10  : Out = In2;
   default: Out = {DATA_WIDTH{1'b0}};
  endcase
 end

endmodule
```

1.2 RISC-V Datahpath

```
`include "Mux2x1.v"
`include "PC.v"
`include "Add.v"
`include "ImmGen.v"
`include "RegisterFile.v"
`include "BranchComp.v"
`include "ALU.v"
`include "Mux3x1.v"

module Data_Path #(parameter DATA_WIDTH = 32) (
    input   wire                        clk,
    input   wire                        rst_n,
    input   wire                        PCSel,
    input   wire    [2:0]               ImmSel,
    input   wire                        RegWEn,
    input   wire                        BrUn,
    input   wire                        BSel,
    input   wire                        ASel,
    input   wire    [3:0]               ALUSel,
    input   wire                        MemRW,
    input   wire    [1:0]               WBSel,
    input   wire    [31:0]              Instr,
    input   wire    [DATA_WIDTH-1:0]    Mem,
    output  wire    [DATA_WIDTH-1:0]    PC_Next,
    output  wire    [DATA_WIDTH-1:0]    ALU_Result,
    output  wire    [DATA_WIDTH-1:0]    Rs2,
    output  wire                        BrEq,
    output  wire                        BrLT
    );


// Internal Wires

wire [DATA_WIDTH-1:0] PC_Plus_4;

wire [DATA_WIDTH-1:0] PC_Mux2x1_Out;

wire [DATA_WIDTH-1:0] Imm;

wire [DATA_WIDTH-1:0] Write_Back_Mux3x1;

wire [DATA_WIDTH-1:0] Rs1;

wire [DATA_WIDTH-1:0] Rs1_ALU;
wire [DATA_WIDTH-1:0] Rs2_ALU;


```

```
49
50      ///*********************************************************///
51      ////////////////////////// PC Mux ////////////////////////////
52      ///*********************************************************///
53
54      Mux2x1 #(.DATA_WIDTH(DATA_WIDTH)) U0_PC_Mux (
55          .Sel(PCSel),
56          .In0(PC_Plus_4),
57          .In1(ALU_Result),
58          .Out(PC_Mux2x1_Out)
59      );
60
61      ///*********************************************************///
62      ////////////////////// PC Register ///////////////////////////
63      ///*********************************************************///
64
65      PC #(.DATA_WIDTH(DATA_WIDTH)) U0_PC (
66          .clk(clk),
67          .rst_n(rst_n),
68          .PC_In(PC_Mux2x1_Out),
69          .PC_Out(PC_Next)
70      );
71
72      ///*********************************************************///
73      ////////////////////////// Adder for PC //////////////////////
74      ///*********************************************************///
75
76      Add #(.DATA_WIDTH(DATA_WIDTH)) U0_Add (
77          .PC(PC_Next),
78          .PC_Plus_4(PC_Plus_4)
79      );
80
81      ///*********************************************************///
82      ////////////////////// Immediate Generator ///////////////////
83      ///*********************************************************///
84
85      ImmGen U0_ImmGen (
86          .Instr(Instr[31:7]),
87          .ImmSel(ImmSel),
88          .Imm(Imm)
89      );
90
91      ///*********************************************************///
92      ////////////////////// Register File /////////////////////////
93      ///*********************************************************///
94
95      RegisterFile #(.DATA_WIDTH(DATA_WIDTH)) U0_Register_File (
96          .clk(clk),
97          .DataD(Write_Back_Mux3x1),
98          .AddrD(Instr[11:7]),
99          .AddrA(Instr[19:15]),
100         .AddrB(Instr[24:20]),
101         .RegWEn(RegWEn),
102         .DataA(Rs1),
103         .DataB(Rs2)
104     );
105
```

```
106     ///*********************************************************///
107     ////////////////////// Branch Comparator //////////////////////
108     ///*********************************************************///
109
110     BranchComp #(.DATA_WIDTH(DATA_WIDTH)) U0_BranchComp (
111         .DataA(Rs1),
112         .DataB(Rs2),
113         .BrUn(BrUn),
114         .BrEq(BrEq),
115         .BrLT(BrLT)
116     );
117
118     ///*********************************************************///
119     ////////////////////////// Rs1 Mux ////////////////////////////
120     ///*********************************************************///
121
122     Mux2x1 #(.DATA_WIDTH(DATA_WIDTH)) U0_Rs1_Mux (
123         .Sel(ASel),
124         .In0(Rs1),
125         .In1(PC_Next),
126         .Out(Rs1_ALU)
127     );
128
129     ///*********************************************************///
130     ////////////////////////// Rs2 Mux ////////////////////////////
131     ///*********************************************************///
132
133     Mux2x1 #(.DATA_WIDTH(DATA_WIDTH)) U0_Rs2_Mux (
134         .Sel(BSel),
135         .In0(Rs2),
136         .In1(Imm),
137         .Out(Rs2_ALU)
138     );
139
140     ///*********************************************************///
141     ////////////////////////////// ALU ////////////////////////////
142     ///*********************************************************///
143
144     ALU #(.DATA_WIDTH(DATA_WIDTH)) U0_ALU (
145         .Src_A(Rs1_ALU),
146         .Src_B(Rs2_ALU),
147         .ALUSel(ALUSel),
148         .ALU_Result(ALU_Result)
149     );
150
151     ///*********************************************************///
152     ////////////////////// Write Back Mux //////////////////////
153     ///*********************************************************///
154
155     Mux3x1 #(.DATA_WIDTH(DATA_WIDTH)) U0_WriteBack_Mux (
156         .Sel(WBSel),
157         .In0(Mem),
158         .In1(ALU_Result),
159         .In2(PC_Plus_4),
160         .Out(Write_Back_Mux3x1)
161     );
162
163
164     endmodule
```

1.3 RISC-V Control Logic

Control Logic Implementation

Only 9 bits are needed to decode the 6 RISC-V instruction formats and their corresponding functions as shown in the figure below.

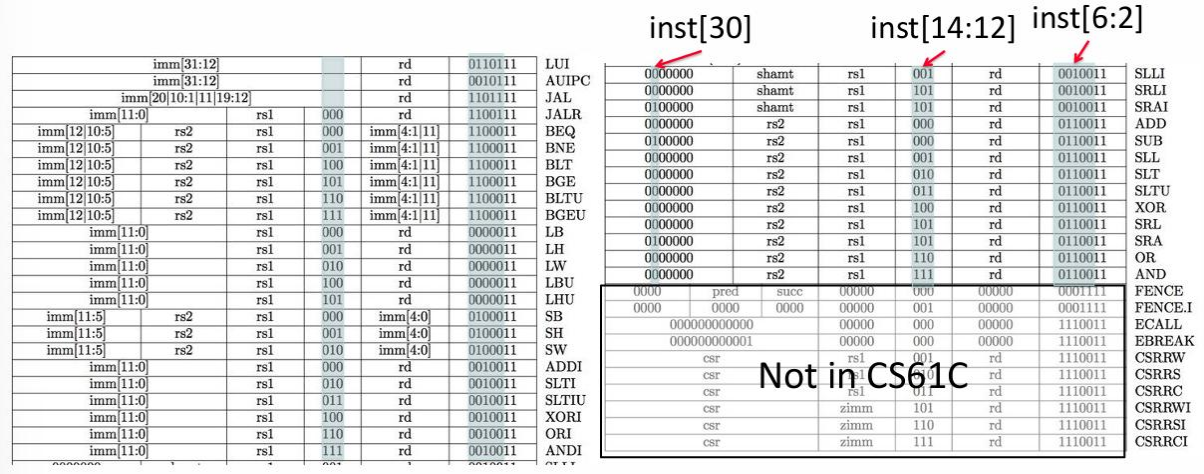


Figure 1-3: RV321 is a 9-bit ISA. [1]

Control Logic Truth Table

Inst[31:0]	BrEq	BrLT	PCSel	ImmSel	BrUn	ASel	BSel	ALUSel	MemRW	RegWEn	WBSel
add	*	*	+4 (0)	*	*	Reg (0)	Reg (0)	Add (0)	Read (0)	1	ALU (1)
sub	*	*	+4 (0)	*	*	Reg (0)	Reg (0)	Sub (1)	Read (0)	1	ALU (1)
(R-R Op)	*	*	+4 (0)	*	*	Reg (0)	Reg (0)	*(Op)* (x)	Read (0)	1	ALU (1)
addi	*	*	+4 (0)	I (1)	*	Reg (0)	Imm (1)	Add (0)	Read (0)	1	ALU (1)
lw	*	*	+4 (0)	I (1)	*	Reg (0)	Imm (1)	Add (0)	Read (0)	1	Mem (0)
sw	*	*	+4 (0)	S (2)	*	Reg (0)	Imm (1)	Add (0)	Write (1)	0	*
beq	0	*	+4 (0)	B (3)	*	PC (1)	Imm (1)	Add (0)	Read (0)	0	*
beq	1	*	ALU (1)	B (3)	*	PC (1)	Imm (1)	Add (0)	Read (0)	0	*
bne	0	*	ALU (1)	B (3)	*	PC (1)	Imm (1)	Add (0)	Read (0)	0	*
bne	1	*	+4 (0)	B (3)	*	PC (1)	Imm (1)	Add (0)	Read (0)	0	*
blt	*	1	ALU (1)	B (3)	0	PC (1)	Imm (1)	Add (0)	Read (0)	0	*
bltu	*	1	ALU (1)	B (3)	1	PC (1)	Imm (1)	Add (0)	Read (0)	0	*
jalr	*	*	ALU (1)	I (1)	*	Reg (0)	Imm (1)	Add (0)	Read (0)	1	PC+4 (2)
jal	*	*	ALU (1)	J (4)	*	PC (1)	Imm (1)	Add (0)	Read (0)	1	PC+4 (2)
auipc	*	*	+4 (0)	U (5)	*	PC (1)	Imm (1)	Add (0)	Read (0)	1	ALU (1)

Figure 1-4: Control logic truth table. Adapted from [1]

For PCSel: 0 = PC+4, 1 = ALU

For ASel: 0 = Reg, 1 = PC

For BSel: 0 = Reg, 1 = Imm

For WBSel: 0 = Mem, 1 = ALU, 10 (Binary) = PC+4

Control Logic Module

```
45    // The commented parts need revision and correction.
46
47    module Control_Logic (
48      input   wire  [31:0]  Instr,
49      input   wire          BrEq,
50      input   wire          BrLT,
51      output  reg           PCSel,
52      output  reg   [2:0]   ImmSel,
53      output  reg           RegWEn,
54      output  reg           BrUn,
55      output  reg           BSel,
56      output  reg           ASel,
57      output  reg   [3:0]   ALUSel,
58      output  reg           MemRW,
59      output  reg   [1:0]   WBSel
60      );
61
62
63    wire [8:0] Instruction_Type;
64
65
66    // Extracting bits for all instructions types
67    assign Instruction_Type = {Instr[30], Instr[14:12], Instr[6:2]};
68
```

1.3.1 R-Format

```
70  always @(*)
71   begin
72    casex (Instruction_Type)
73
74     /*************************************************************** R_Format ***************************************************************************/
75
76     9'b0_000_01100 : begin  // add
77                       PCSel = 1'b0; ImmSel = 3'bxxx; BrUn = 1'bx; ASel = 1'b0; BSel = 1'b0; ALUSel = 4'b0000; MemRW = 1'b0; RegWEn = 1'b1; WBSel = 2'b01;
78                      end
79     9'b1_000_01100 : begin  // sub
80                       PCSel = 1'b0; ImmSel = 3'bxxx; BrUn = 1'bx; ASel = 1'b0; BSel = 1'b0; ALUSel = 4'b1000; MemRW = 1'b0; RegWEn = 1'b1; WBSel = 2'b01;
81                      end
82     /*
83     9'b0_001_01100 : begin  // sll
84                       PCSel = 1'b0; ImmSel = 3'bxxx; BrUn = 1'bx; ASel = 1'b0; BSel = 1'b0; ALUSel = 4'b0001; MemRW = 1'b0; RegWEn = 1'b1; WBSel = 2'b01;
85                      end
86     9'b0_010_01100 : begin  // slt
87                       PCSel = 1'b0; ImmSel = 3'bxxx; BrUn = 1'bx; ASel = 1'b0; BSel = 1'b0; ALUSel = 4'b0010; MemRW = 1'b0; RegWEn = 1'b1; WBSel = 2'b01;
88                      end
89     9'b0_011_00000 : begin  // sltu
90                       PCSel = 1'b0; ImmSel = 3'bxxx; BrUn = 1'bx; ASel = 1'b0; BSel = 1'b0; ALUSel = 4'b0011; MemRW = 1'b0; RegWEn = 1'b1; WBSel = 2'b01;
91                      end
92     9'b0_100_00000 : begin  // xor
93                       PCSel = 1'b0; ImmSel = 3'bxxx; BrUn = 1'bx; ASel = 1'b0; BSel = 1'b0; ALUSel = 4'b0100; MemRW = 1'b0; RegWEn = 1'b1; WBSel = 2'b01;
94                      end
95     9'b0_101_00000 : begin  // srl
96                       PCSel = 1'b0; ImmSel = 3'bxxx; BrUn = 1'bx; ASel = 1'b0; BSel = 1'b0; ALUSel = 4'b0101; MemRW = 1'b0; RegWEn = 1'b1; WBSel = 2'b01;
97                      end
98     9'b1_101_00000 : begin  // sra
99                       PCSel = 1'b0; ImmSel = 3'bxxx; BrUn = 1'bx; ASel = 1'b0; BSel = 1'b0; ALUSel = 4'b1101; MemRW = 1'b0; RegWEn = 1'b1; WBSel = 2'b01;
100                     end
101    */
102     9'b0_110_00000 : begin  // or
103                       PCSel = 1'b0; ImmSel = 3'bxxx; BrUn = 1'bx; ASel = 1'b0; BSel = 1'b0; ALUSel = 4'b0110; MemRW = 1'b0; RegWEn = 1'b1; WBSel = 2'b01;
104                     end
105     9'b0_111_00000 : begin  // and
106                       PCSel = 1'b0; ImmSel = 3'bxxx; BrUn = 1'bx; ASel = 1'b0; BSel = 1'b0; ALUSel = 4'b0111; MemRW = 1'b0; RegWEn = 1'b1; WBSel = 2'b01;
107                     end
```

1.3.2 I-Format

```
/************************************************************** I_Format ***************************************************************/

 9'bx_000_00100 : begin  // addi
                   PCSel = 1'b0; ImmSel = 3'b001; BrUn = 1'bx; ASel = 1'b0; BSel = 1'b1; ALUSel = 4'b0000; MemRW = 1'b0; RegWEn = 1'b1; WBSel = 2'b01;
                  end
/*
 9'bx_010_00100 : begin  // slti
                   PCSel = 1'b0; ImmSel = 3'b001; BrUn = 1'bx; ASel = 1'b0; BSel = 1'b0; ALUSel = 4'b0010; MemRW = 1'b0; RegWEn = 1'b1; WBSel = 2'b00;
                  end
 9'bx_011_00100 : begin  // sltiu
                   PCSel = 1'b0; ImmSel = 3'b001; BrUn = 1'bx; ASel = 1'b0; BSel = 1'b0; ALUSel = 4'b0001; MemRW = 1'b0; RegWEn = 1'b1; WBSel = 2'b00;
                  end
 9'bx_100_00100 : begin  // xori
                   PCSel = 1'b0; ImmSel = 3'b001; BrUn = 1'bx; ASel = 1'b0; BSel = 1'b0; ALUSel = 4'b0010; MemRW = 1'b0; RegWEn = 1'b1; WBSel = 2'b00;
                  end
*/
 9'bx_110_00100 : begin  // ori
                   PCSel = 1'b0; ImmSel = 3'b001; BrUn = 1'bx; ASel = 1'b0; BSel = 1'b1; ALUSel = 4'b0110; MemRW = 1'b0; RegWEn = 1'b1; WBSel = 2'b01;
                  end
 9'bx_111_00100 : begin  // andi
                   PCSel = 1'b0; ImmSel = 3'b001; BrUn = 1'bx; ASel = 1'b0; BSel = 1'b1; ALUSel = 4'b0111; MemRW = 1'b0; RegWEn = 1'b1; WBSel = 2'b01;
                  end
/*
 9'b0_001_00100 : begin  // slli
                   PCSel = 1'b0; ImmSel = 3'b001; BrUn = 1'bx; ASel = 1'b0; BSel = 1'b0; ALUSel = 4'b0101; MemRW = 1'b0; RegWEn = 1'b1; WBSel = 2'b00;
                  end
 9'b0_101_00100 : begin  // srli
                   PCSel = 1'b0; ImmSel = 3'b001; BrUn = 1'bx; ASel = 1'b0; BSel = 1'b0; ALUSel = 4'b1101; MemRW = 1'b0; RegWEn = 1'b1; WBSel = 2'b00;
                  end
 9'b0_101_00100 : begin  // srai
                   PCSel = 1'b0; ImmSel = 3'b001; BrUn = 1'bx; ASel = 1'b0; BSel = 1'b0; ALUSel = 4'b0110; MemRW = 1'b0; RegWEn = 1'b1; WBSel = 2'b00;
                  end
*/
 /********************************************************** I_Format Contd. Load ******************************************************/
  /*
 9'bx_000_00000 : begin  // lb
                   PCSel = 1'b0; ImmSel = 3'b001; BrUn = 1'bx; ASel = 1'b0; BSel = 1'b0; ALUSel = 4'b0000; MemRW = 1'b0; RegWEn = 1'b1; WBSel = 2'b00;
                  end
 9'bx_001_00000 : begin  // lh
                   PCSel = 1'b0; ImmSel = 3'b001; BrUn = 1'bx; ASel = 1'b0; BSel = 1'b0; ALUSel = 4'b1000; MemRW = 1'b0; RegWEn = 1'b1; WBSel = 2'b00;
                  end
  */
 9'bx_010_00000 : begin  // lw
                   PCSel = 1'b0; ImmSel = 3'b001; BrUn = 1'bx; ASel = 1'b0; BSel = 1'b1; ALUSel = 4'b0000; MemRW = 1'b0; RegWEn = 1'b1; WBSel = 2'b00;
                  end
/*
 9'bx_100_00000 : begin  // lbu
                   PCSel = 1'b0; ImmSel = 3'b001; BrUn = 1'bx; ASel = 1'b0; BSel = 1'b0; ALUSel = 4'b0010; MemRW = 1'b0; RegWEn = 1'b1; WBSel = 2'b00;
                  end
 9'bx_101_00000 : begin  // lhu
                   PCSel = 1'b0; ImmSel = 3'b001; BrUn = 1'bx; ASel = 1'b0; BSel = 1'b0; ALUSel = 4'b0011; MemRW = 1'b0; RegWEn = 1'b1; WBSel = 2'b00;
                  end
*/
 /********************************************************** I_Format Contd. JLAR ******************************************************/

 9'bx_xxx_11001 : begin  // jalr
                   PCSel = 1'b1; ImmSel = 3'b001; BrUn = 1'bx; ASel = 1'b0; BSel = 1'b1; ALUSel = 4'b0000; MemRW = 1'b0; RegWEn = 1'b1; WBSel = 2'b10;
                  end
```

1.3.3 S-Format

```

/************************************************************** S_Format ***************************************************************/
  /*
 9'bx_000_01000 : begin  // sb
                   PCSel = 1'b0; ImmSel = 3'b010; BrUn = 1'bx; ASel = 1'b0; BSel = 1'b0; ALUSel = 4'b0000; MemRW = 1'b0; RegWEn = 1'b1; WBSel = 2'b00;
                  end
 9'bx_001_01000 : begin  // sh
                   PCSel = 1'b0; ImmSel = 3'b010; BrUn = 1'bx; ASel = 1'b0; BSel = 1'b0; ALUSel = 4'b1000; MemRW = 1'b0; RegWEn = 1'b1; WBSel = 2'b00;
                  end
  */
 9'bx_010_01000 : begin  // sw
                   PCSel = 1'b0; ImmSel = 3'b010; BrUn = 1'bx; ASel = 1'b0; BSel = 1'b1; ALUSel = 4'b0000; MemRW = 1'b1; RegWEn = 1'b0; WBSel = 2'bxx;
                  end
```

1.3.4 B-Format

```
183
184     /************************************************************** B_Format ***************************************************************/
185
186     9'bx_000_11000 : begin  // beq
187                      PCSel = (BrEq == 0) ? 1'b0 : 1'b1; ImmSel = 3'b011; BrUn = 1'bx; ASel = 1'b1; BSel = 1'b1; ALUSel = 4'b0000; MemRW = 1'b0; RegWEn = 1'b0; WBSel = 2'bxx;
188                      end
189     9'bx_001_11000 : begin  // bne
190                      PCSel = (BrEq == 0) ? 1'b1 : 1'b0; ImmSel = 3'b011; BrUn = 1'bx; ASel = 1'b1; BSel = 1'b1; ALUSel = 4'b0000; MemRW = 1'b0; RegWEn = 1'b0; WBSel = 2'bxx;
191                      end
192    /*
193     9'bx_100_11000 : begin  // blt
194                      if (BrLT)
195                       begin
196                        PCSel = 1'b1; ImmSel = 3'b011; BrUn = 1'bx; ASel = 1'b0; BSel = 1'b0; ALUSel = 4'b0001; MemRW = 1'b0; RegWEn = 1'b1; WBSel = 2'b00;
197                       end
198                      else
199                       begin
200                        PCSel = 1'bx; ImmSel = 3'bxxx; BrUn = 1'bx; ASel = 1'bx; BSel = 1'bx; ALUSel = 4'bxxxx; MemRW = 1'bx; RegWEn = 1'bx; WBSel = 2'bxx;
201                       end
202                      end
203     9'bx_101_11000 : begin  // bge
204                      PCSel = 1'b0; ImmSel = 3'b011; BrUn = 1'bx; ASel = 1'b0; BSel = 1'b0; ALUSel = 4'b0000; MemRW = 1'b0; RegWEn = 1'b1; WBSel = 2'b00;
205                      end
206     9'bx_110_11000 : begin  // bltu
207                      PCSel = 1'b0; ImmSel = 3'b011; BrUn = 1'bx; ASel = 1'b0; BSel = 1'b0; ALUSel = 4'b1000; MemRW = 1'b0; RegWEn = 1'b1; WBSel = 2'b00;
208                      end
209     9'bx_111_11000 : begin  // bgeu
210                      PCSel = 1'b0; ImmSel = 3'b011; BrUn = 1'bx; ASel = 1'b0; BSel = 1'b0; ALUSel = 4'b0001; MemRW = 1'b0; RegWEn = 1'b1; WBSel = 2'b00;
211                      end
```

1.3.5 U-Format

```
224     /************************************************************** U_Format ***************************************************************/
225     /*
226      9'bx_xxx_00101 : begin  // lui
227                       PCSel = 1'b0; ImmSel = 3'b101; BrUn = 1'bx; ASel = 1'b0; BSel = 1'b0; ALUSel = 4'b0000; MemRW = 1'b0; RegWEn = 1'b1; WBSel = 2'b00;
228                       end
229      */
230      9'bx_xxx_00101 : begin  // auipc
231                       PCSel = 1'b0; ImmSel = 3'b101; BrUn = 1'bx; ASel = 1'b1; BSel = 1'b1; ALUSel = 4'b0000; MemRW = 1'b0; RegWEn = 1'b1; WBSel = 2'b01;
232                       end
233
```

1.3.6 J-Format

```
234     /************************************************************** J_Format ***************************************************************/
235
236     9'bx_xxx_11011 : begin  // jal
237                      PCSel = 1'b1; ImmSel = 3'b100; BrUn = 1'bx; ASel = 1'b1; BSel = 1'b1; ALUSel = 4'b0000; MemRW = 1'b0; RegWEn = 1'b1; WBSel = 2'b10;
238                      end
239
240     /********************************************************** Env_Calls Not Implemented Yet **********************************************/
241
242
243     default        : begin  // illegal
244                      PCSel = 1'bx; ImmSel = 3'bxxx; BrUn = 1'bx; ASel = 1'bx; BSel = 1'bx; ALUSel = 4'bxxxx; MemRW = 1'bx; RegWEn = 1'bx; WBSel = 2'bxx;
245                      end
246    endcase
247   end
248
249  endmodule
```

1.4 RISC-V Single Cycle Top Module

```
1   `include "Data_Path/Data_Path.v"
2   `include "Control_Logic/Control_Logic.v"
3
4   module RISC_V #(parameter DATA_WIDTH = 32) (
5       input  wire                         clk,
6       input  wire                         rst_n,
7       input  wire   [31:0]                Instr,
8       input  wire   [DATA_WIDTH-1:0]      Mem,
9       output wire   [DATA_WIDTH-1:0]      Addr_IMEM,
10      output wire   [DATA_WIDTH-1:0]      Addr_DMEM,
11      output wire   [DATA_WIDTH-1:0]      Rs2,
12      output wire                         MemRW
13      );
14
15
16  // Internal Wires
17
18  wire                    PCSel;
19
20  wire [2:0]              ImmSel;
21
22  wire                    RegWEn;
23
24  wire                    BrUn;
25
26  wire                    BrEq;
27
28  wire                    BrLT;
29
30  wire                    BSel;
31
32  wire                    ASel;
33
34  wire [3:0]              ALUSel;
35
36  wire [1:0]              WBSel;
37
38
39
40
```

```
40
41      ///*************************************************************///
42      ////////////////////////// Data Path //////////////////////////////
43      ///*************************************************************///
44
45      Data_Path #(.DATA_WIDTH(DATA_WIDTH)) U_Data_Path(
46          .clk(clk),
47          .rst_n(rst_n),
48          .PCSel(PCSel),
49          .ImmSel(ImmSel),
50          .RegWEn(RegWEn),
51          .BrUn(BrUn),
52          .BSel(BSel),
53          .ASel(ASel),
54          .ALUSel(ALUSel),
55          .MemRW(MemRW),
56          .WBSel(WBSel),
57          .Instr(Instr),
58          .Mem(Mem),
59          .PC_Next(Addr_IMEM),
60          .ALU_Result(Addr_DMEM),
61          .Rs2(Rs2),
62          .BrEq(BrEq),
63          .BrLT(BrLT)
64      );
65
66      ///*************************************************************///
67      ////////////////////////// Control Logic //////////////////////////
68      ///*************************************************************///
69
70      Control_Logic U_Control_Logic(
71          .Instr(Instr),
72          .BrEq(BrEq),
73          .BrLT(BrLT),
74          .PCSel(PCSel),
75          .ImmSel(ImmSel),
76          .RegWEn(RegWEn),
77          .BrUn(BrUn),
78          .BSel(BSel),
79          .ASel(ASel),
80          .ALUSel(ALUSel),
81          .MemRW(MemRW),
82          .WBSel(WBSel)
83      );
84
85
86      endmodule
```

1.5 Memories

DMEM and IMEM were implemented for behavioral modeling purposes since they come as hard macros. These implementations interact directly with the processor without incorporating any cache-specific operations, such as hit or miss behavior, which involves managing tags, priorities, and exception handling.

1.5.1 Instruction Memory

```
`timescale 1ps / 1fs

module IMEM #(parameter DATA_WIDTH = 32, MEM_SIZE = 256) (
    input   wire     [31:0] Addr,
    output  reg      [31:0] Instr
);

reg [DATA_WIDTH-1:0] Mem [0:MEM_SIZE-1];

initial
begin
    $readmemh("Memory/instructions.txt", Mem);  // Load instructions from a file
end

// Instruction Memory Delay
always @(*) begin
    #100 Instr = Mem[Addr];
end

endmodule
```

1.5.2 Data Memory

```
`timescale 1ps/1fs

module DMEM #(parameter DATA_WIDTH = 32, MEM_SIZE = 256) (
    input  wire                  clk,
    input  wire                  MemRW,
    input  wire [7:0] Addr,
    input  wire [DATA_WIDTH-1:0] DataW,          // Data to be written
    output reg  [DATA_WIDTH-1:0] DataR           // Data read from memory
    );


reg [DATA_WIDTH-1:0] memory [0:MEM_SIZE-1];

// Memory Read Operation (Combinational)
always @(*)
 begin
  #100;
  if (!MemRW)
   DataR = memory[Addr[7:0]];
  else
   DataR = 32'bx;
 end

// Memory Write Operation
always @(posedge clk)
 begin
    #100
    if (MemRW)
       memory[Addr[7:0]] <= DataW;
 end


endmodule
```

NOTE:

For simplicity, assume that the instruction memory is a 1kB word-addressable ROM and the data memory is a 1kB word-addressable RAM, i.e., you will only connect bits 9 to 2 in the address bus.

You have a **32-bit address**, but your memory only has **256-word locations (1kB)**. So, the question is: how can a memory with only 256 locations handle addresses that span a much larger range (like 0x10000000) , as seen in the Venus simulator?

The Solution

The system **ignores most of the address bits** and only uses the bits that fit within the memory size. Specifically:

- Only **bits 9 to 2** of the address are used to index the memory.
- The remaining bits (31 to 10 and 1 to 0) are **ignored**.

How It Works

1. **Word Addressing**:
   Since the memory is **word-addressable**, each location holds 4 bytes. The **lower 2 bits** (1 and 0) of the address are not needed because they represent the byte offset within a word. For example:
   - Address `0x00000000` → Word 0, Byte 0.
   - Address `0x00000004` → Word 1, Byte 0.
   - Address `0x00000008` → Word 2, Byte 0

Example with 0x10000000

If you have an address like 0x10000000:

- In binary: 00010000000000000000000000000000
- Take bits 9 to 2: 00000000
- Result: This address maps to **word 0 in memory**.

If you instead have 0x10000004:

- In binary: 00010000000000000000000000000100
- Take bits 9 to 2: 00000001
- Result: This address maps to **word 1 in memory**.

2. Generating Fibonacci Machine Code for RISC-V using Venus

2.1 Fibonacci Sequence Assembly Code

```
.data
fib_result: .word 0      # Reserve space for one word to store the result

.text
main:
    add t0, x0, x0       # t0 = 0 (1st Fibonacci number)
    addi t1, x0, 1       # t1 = 1 (2nd Fibonacci number)
    li t4, 6             # Load n = 6 into t4

fib:
    beq t4, x0, finish   # If n == 0, exit loop
    add t2, t1, t0       # t2 = t1 + t0 (next Fibonacci number)
    mv t0, t1            # t0 = t1 (shift Fibonacci sequence)
    mv t1, t2            # t1 = t2 (shift Fibonacci sequence)
    addi t4, t4, -1      # Decrement n
    j fib                # Repeat the loop
finish:
    la t3, fib_result    # Load the address of fib_result into t3
    sw t0, 0(t3)         # Store the final Fibonacci number in fib_result
```

Figure 2-1: Fibonacci Assembly Code for the 6th Fibonacci Number.

Figure 2-2: 6th Fibonacci number is stored at register x5.

The address of fib_result is calculated during the la t3, fib_result instruction. If any instruction is inserted before this as no operation (nop) as shown in Figure 2-3, the address of fib_result will shift, causing the calculated address to increase. For example, if the address was initially expected to be 0x00010000, adding instructions before the la may result in an address like 0x00010004 instead as in Figure 2-4. This discrepancy would cause the result to no longer align with the expected DMEM address, potentially leading to incorrect behavior.

```
9
10 fib:
11     beq t4, x0, finish    # If n == 0, exit loop
12     add t2, t1, t0        # t2 = t1 + t0 (next Fibonacci number)
13     mv t0, t1             # t0 = t1 (shift Fibonacci sequence)
14     mv t1, t2             # t1 = t2 (shift Fibonacci sequence)
15     addi t4, t4, -1       # Decrement n
16     j fib                 # Repeat the loop
17     nop
18
19 finish:
20     la t3, fib_result     # Load the address of fib_result into t3
```

Figure 2-3: Insertion of instruction before loading address of the result.

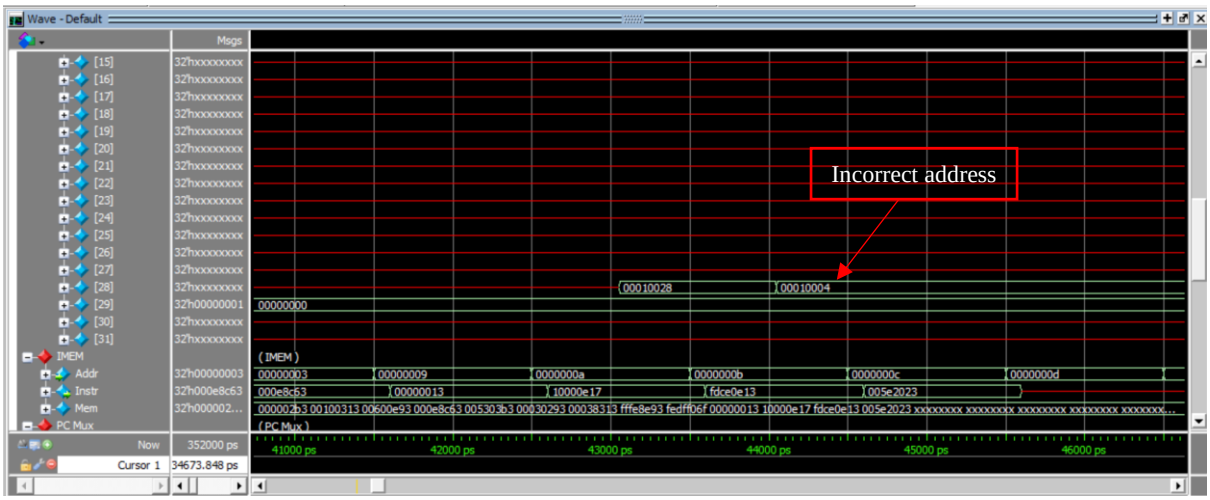


Figure 2-4: Incorrect mapped addressed.

Unlike the Venus simulator, which directly calculates the address of .word 0 based on the declared memory layout and does not rely on subsequent instruction positioning, in this case, manual adjustments are necessary to ensure consistency with the expected address mapping.

Therefore, adjustments were made to the code by moving the la t3, fib_result instruction to the top. This ensures that its calculated address remains consistent, even if additional instructions are inserted before it later in the code as shown in Figure 2-5.

```
.data
fib_result: .word 0      # Reserve space for one word to store the result

.text
main:
    la t3, fib_result    # Load the address of fib_result into t3
    add t0, x0, x0       # t0 = 0 (1st Fibonacci number)
    addi t1, x0, 1       # t1 = 1 (2nd Fibonacci number)
    li t4, 6             # Load n = 9 into t4

fib:
    beq t4, x0, finish   # If n == 0, exit loop
    add t2, t1, t0       # t2 = t1 + t0 (next Fibonacci number)
    mv t0, t1            # t0 = t1 (shift Fibonacci sequence)
    mv t1, t2            # t1 = t2 (shift Fibonacci sequence)
    addi t4, t4, -1      # Decrement n
    j fib                # Repeat the loop

finish:
    sw t0, 0(t3)         # Store the final Fibonacci number in fib_result
```

Figure 2-5: Modified assembly code.

3. Verification of RISC-V RTL

3.1 RISC-V Simple Testbench

```
`include "Memory/IMEM.v"
`include "Memory/DMEM.v"

`timescale 1ps/1fs

module RISC_V_tb ();

//////////////////////////////////////////////////////////////////
/////////////////////// Parameters ///////////////////////////////
//////////////////////////////////////////////////////////////////

parameter CLK_PERIOD = 1000;  // 1ns
parameter DATA_WIDTH = 32;
parameter MEM_SIZE   = 256;

//////////////////////////////////////////////////////////////////
/////////////////////// DUT Signals //////////////////////////////
//////////////////////////////////////////////////////////////////

reg                 clk_tb;
reg                 rst_n_tb;

wire [31:0]         Mem;       // used to connect risc_v input of wb_mux to output of DMEM DataR

wire [31:0]         Addr_IMEM;
wire [31:0]         Addr_DMEM;
wire [31:0]         Rs2;
wire [31:0]         Instr;
wire                MemRW;

//////////////////////////////////////////////////////////////////
/////////////////////// Clock Generation /////////////////////////
//////////////////////////////////////////////////////////////////

initial
 begin
    clk_tb = 0;
    forever #(CLK_PERIOD / 2) clk_tb = ~clk_tb;
 end
```

```
//////////////////////////////////////////////////////////////////
/////////////////////// DUT Instantiations ///////////////////////
//////////////////////////////////////////////////////////////////

RISC_V #(.DATA_WIDTH(DATA_WIDTH)) DUT (
    .clk(clk_tb),
    .rst_n(rst_n_tb),
    .Instr(Instr),
    .Mem(Mem),
    .Addr_IMEM(Addr_IMEM),
    .Addr_DMEM(Addr_DMEM),
    .Rs2(Rs2),
    .MemRW(MemRW)
);

IMEM #(.DATA_WIDTH(DATA_WIDTH), .MEM_SIZE(MEM_SIZE)) U_IMEM (
    .Addr(Addr_IMEM/4),     // divide by 4 because PC increments by 4 each time so we access one whole word at a time so to overcome this we divide the PC by 4 to make it move by 1 at a time
    .Instr(Instr)
);

DMEM #(.DATA_WIDTH(DATA_WIDTH), .MEM_SIZE(MEM_SIZE)) U_DMEM (
    .clk(clk_tb),
    .MemRW(MemRW),
    .Addr(Addr_DMEM[9:2]),  // Since the memory is word-addressable, each location holds 4 bytes. The lower 2 bits (1 and 0) of the address are not needed because they represent the byte offset within a word.
    .DataW(Rs2),
    .DataR(Mem)
);

//////////////////////////////////////////////////////////////////
/////////////////////// Initial Block ////////////////////////////
//////////////////////////////////////////////////////////////////

initial
 begin
    // System functions
    $dumpfile("RISC_V_tb.vcd");
    $dumpvars;

    reset();


    #(150*CLK_PERIOD)

    $display("Value stored in the register (t0): %0d", DUT.U_Data_Path.U0_Register_File.RegFile[5]);


    $display("Value stored in the Data Memory (Address 0): %0d", U_DMEM.memory[0]);



    #(200*CLK_PERIOD)
    $stop;
 end
```

3.2 Simulation Results

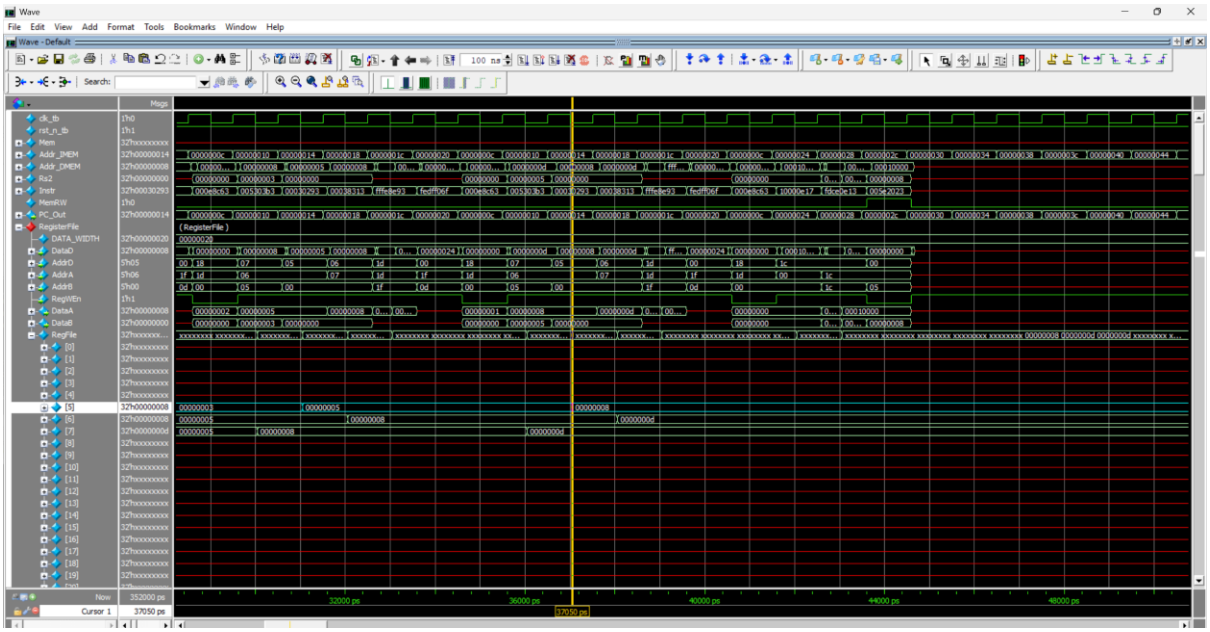


Figure 3-1: 6th Fibonacci number successfully written at register x5.

3.2.1 Testing Rest of the Instructions (store word)

Run | Step | Prev | Reset | Dump | Trace | Re-assemble from Editor

			number)
0x10	0x00600E93	addi x29 x0 6	li t4, 6 # Load n = 9 into t4
0x14	0x000E8C63	beq x29 x0 24	beq t4, x0, finish # If n == 0, exit loop
0x18	0x005303B3	add x7 x6 x5	add t2, t1, t0 # t2 = t1 + t0 (next Fibonacci number)
0x1c	0x00030293	addi x5 x6 0	mv t0, t1 # t0 = t1 (shift Fibonacci sequence)
0x20	0x00038313	addi x6 x7 0	mv t1, t2 # t1 = t2 (shift Fibonacci sequence)
0x24	0xFFFE8E93	addi x29 x29 -1	addi t4, t4, -1 # Decrement n
0x28	0xFEDFF06F	jal x0 -20	j fib # Repeat the loop
0x2c	0x005E2023	sw x5 0(x28)	sw t0, 0(t3) # Store the final Fibonacci number in fib_result

Figure 3-2: 6th Fibonacci number should be stored at PC = 0x2c.

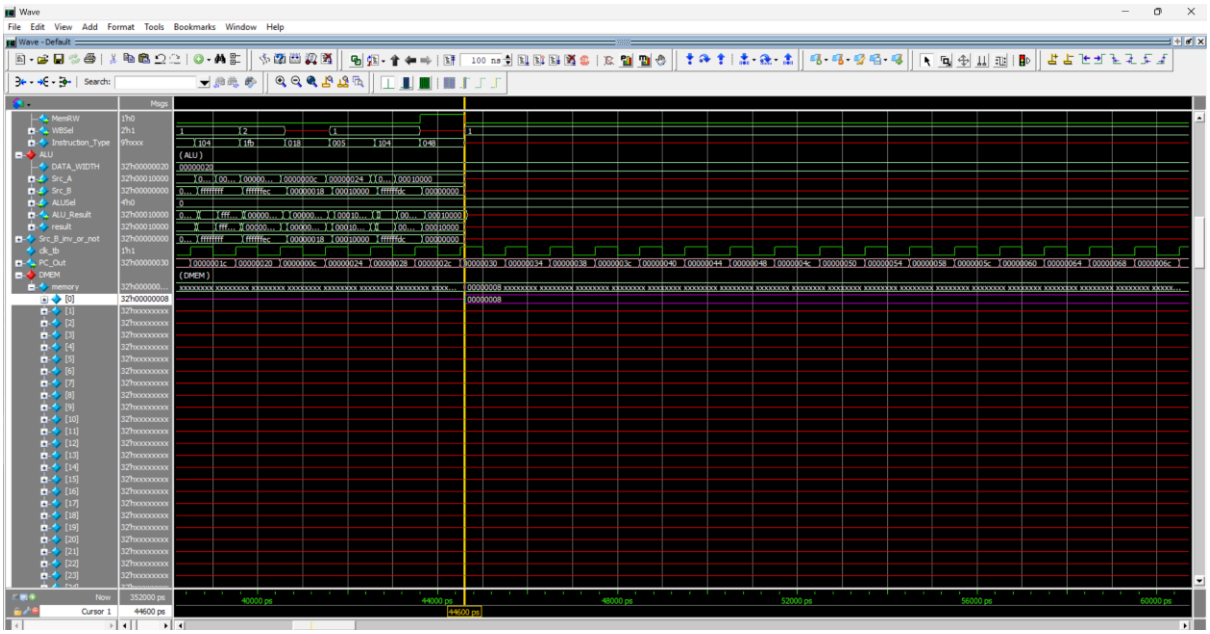


Figure 3-3: 6th Fibonacci number successfully written at memory address 0.

4. Pipelined RISC-V Archtiecture

To apply pipelining to RISC-V, we divide the datapath into five stages: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (M), and Write Back (WB), as shown in Figure 4-1.

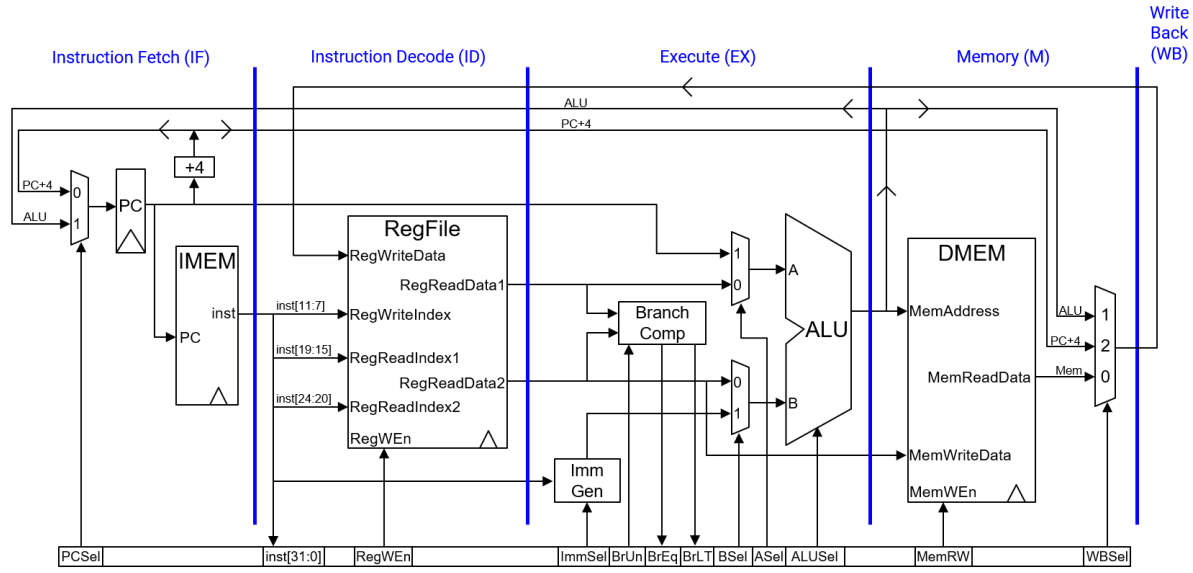


Figure 4-1: Datapath divided into 5 stages. [2]

Add registers between each stage to hold signals until the next clock cycle, shown in Figure 4-2.

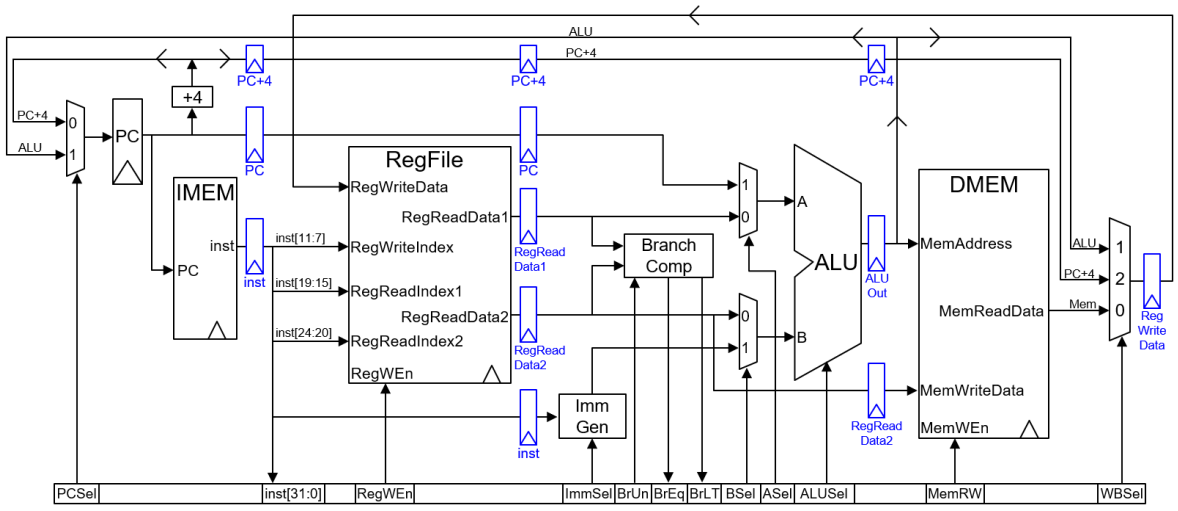


Figure 4-2: Registers added between each stage. [2]

Recompute PC+4 from PC in the memory stage to avoid storing or passing both PC and PC+4 down the pipeline, shown in Figure 4-3.

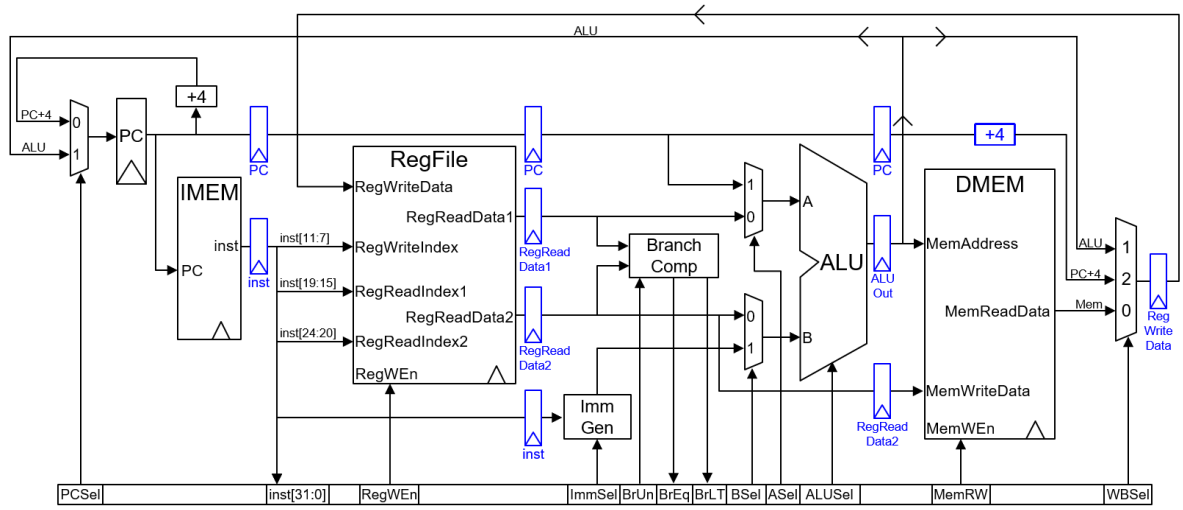


Figure 4-3: Recomputing PC+4 in the memory stage. [2]

Additionally, update RegWriteIndex to use the rd field from the WB stage instruction instead of the ID stage instruction, shown in Figure 4-4.

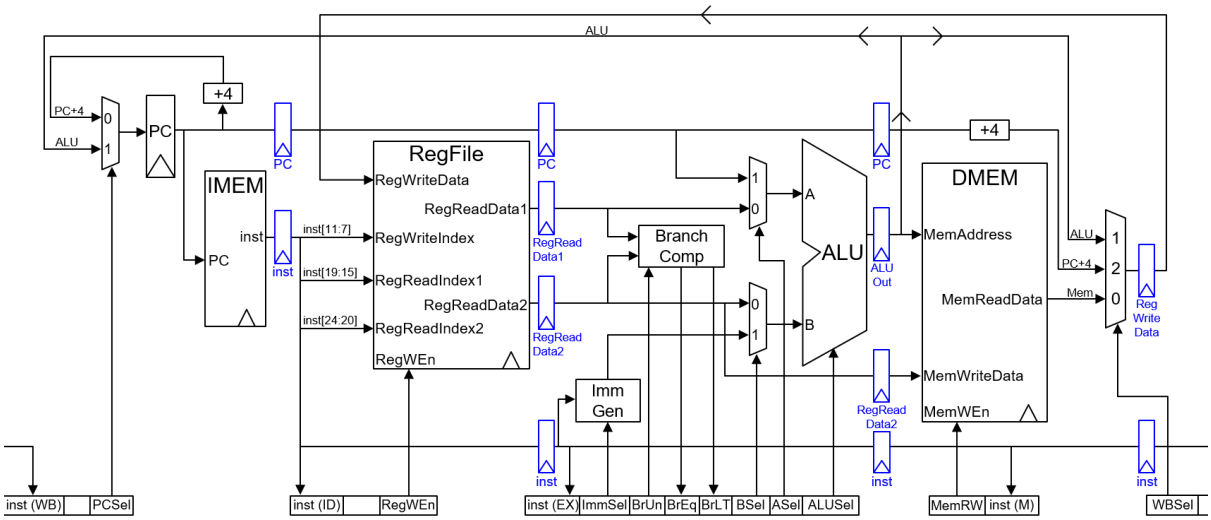


Figure 4-4: Update RegWriteIndex to use the rd field from the WB stage. [2]

Now that the concept is clear, I implemented pipelined RISC-V based on the following final diagram with some modifications. I moved BrEq and BrLT from the execution stage to the memory stage, as they are responsible only for generating the PCSel signal, shown in Figure 4-5. As a result, they needed to be registered to ensure they arrive at the right time for correct pipeline operation, as shown in the final version in Figure 4-6.

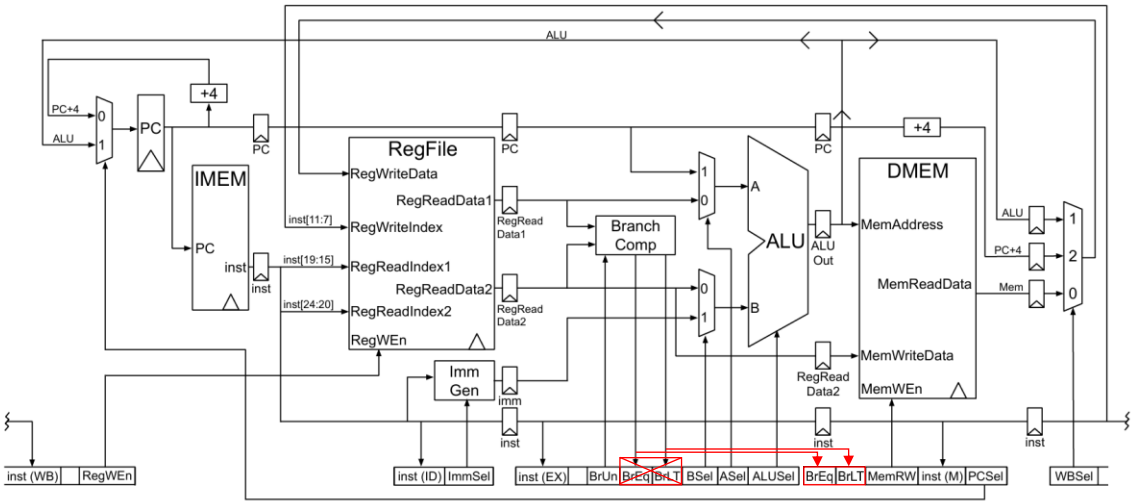


Figure 4-5: Modified pipelined RISC-V architecture. Adapted from [3]

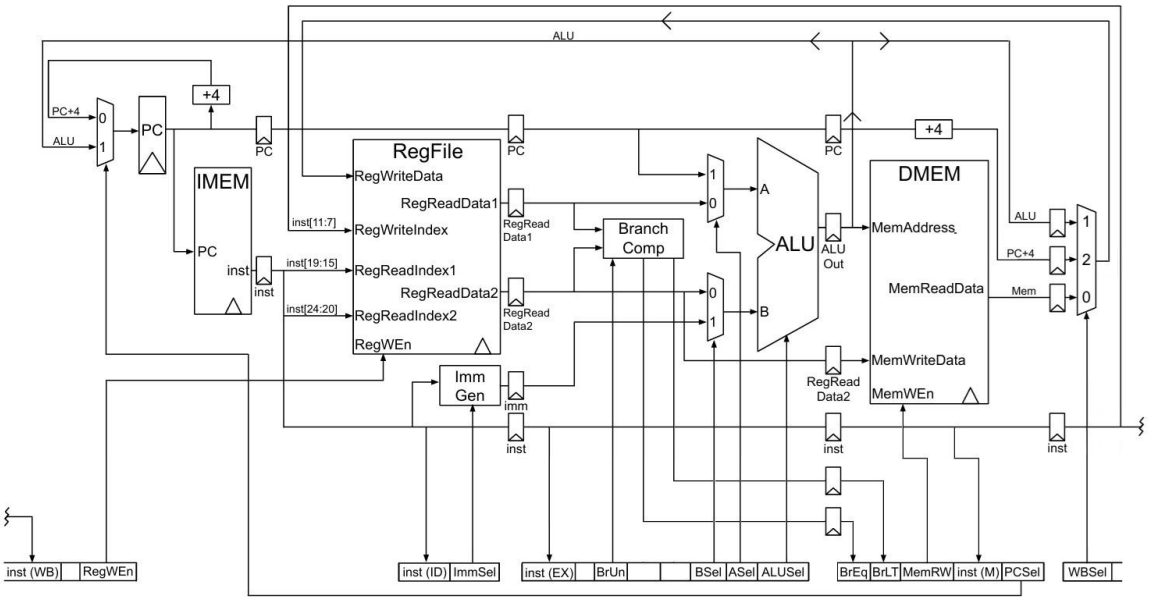


Figure 4-6: Final pipelined RISC-V architecture. Adapted from [3]

The following Figure 4-7 presents a summary of RISC-V instruction formats, categorizing them into R-type, I-type, S-type, B-type, U-type, and J-type, each with its specific bit-field structure.

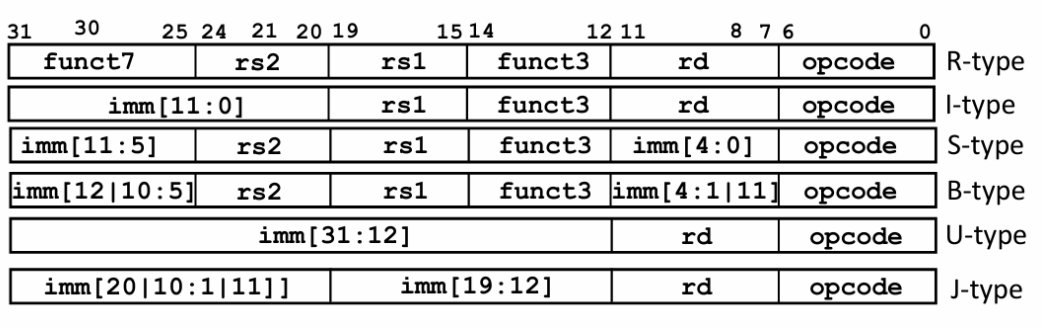


Figure 4-7: Summary of RISC-V instruction formats. [4]

The below figure illustrates the general operation of the RISC-V pipeline, showing the flow of instructions through each of the five stages until the pipeline is fully filled.

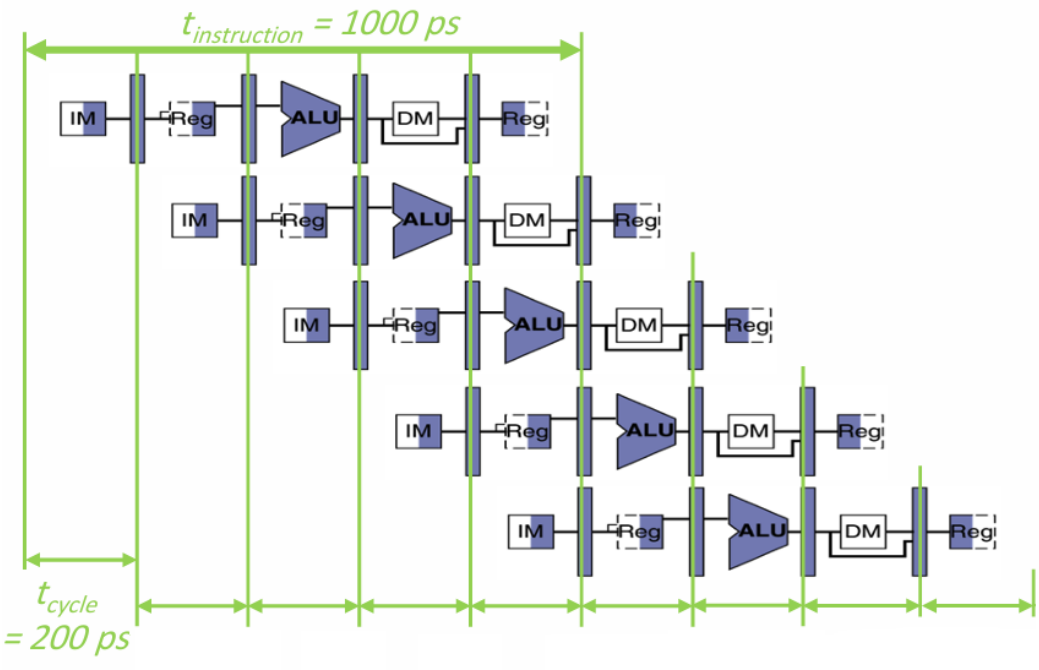


Figure 4-8: Pipeline Stages of the RISC-V. Adapted from [5]

4.1 Pipeline Hazards

4.1.1 Data Hazards

Data hazards arise when instructions that exhibit data dependencies are executed concurrently in the pipeline, causing conflicts.

Types of Data Hazards:

1. **RAW (Read After Write) R-type instructions:**
    - Occurs when an instruction depends on the result of a previous instruction that has not yet completed.
    - **Solution:** Forwarding, stalls.
2. **Load:**
    - Occurs when an instruction writes to a register that a previous instruction is still reading.
    - **Solution:** Reorder instructions, stalls.

4.1.2 Control Hazards

Control hazards occur when the pipeline cannot determine the next instruction to fetch due to a branch or jump.

Solution: Branch prediction, delay slots, stalls.

4.1.3 Structural Hazards

Structural hazards occur when two or more instructions compete for the same hardware resource.

Solution: Add resources, stalls.

NOTE: In RISC-V architectures, **structural hazards are typically avoided** due to the design of the ISAs and microarchitecture, which emphasizes simplicity and regularity.

The following Figure 4-9 illustrates a pipelined execution of RISC-V instructions, highlighting that read and write operations on the register file occur within the same clock cycle. By using the negative edge for writing and the positive edge for reading, structural hazards are avoided.

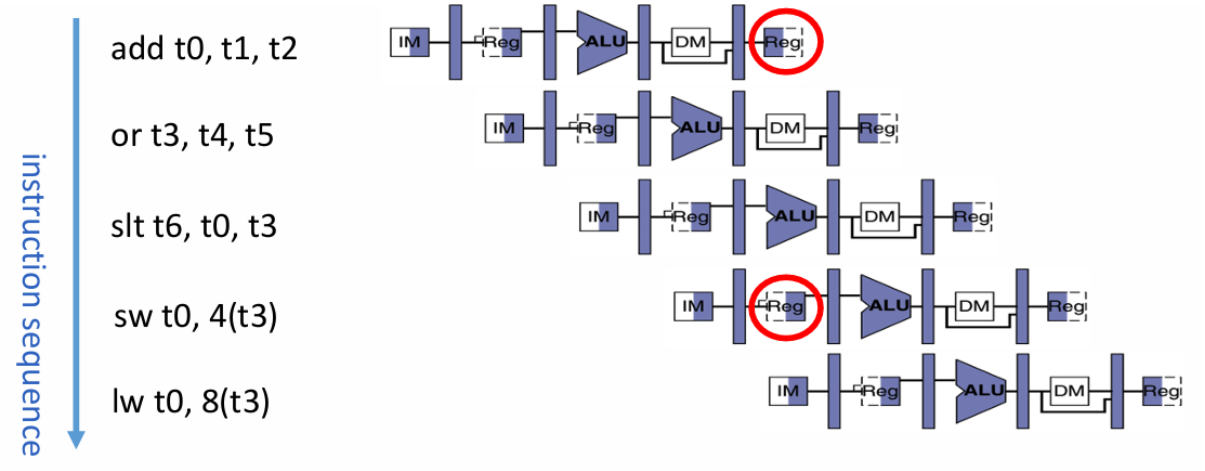


Figure 4-9: Pipelined RISC-V instructions with edge-triggered memory access: write on negative edge, read on positive edge, eliminating structural hazards. [5]

NOTE: Writing and reading in the same cycle would not always be feasible, particularly in high-frequency designs.

4.2 Implementation of Pipelined RISC-V Without Hazard Unit: Fixing Hazards by Stalling

Instruction: la t3, fib_result

- **Type of Hazard:** RAW Data Hazard

The pseudo-instruction la t3, fib_result (load address) is translated into two machine instructions in RISC-V assembly:

1. **First instruction:** auipc (add upper immediate to PC), which computes the upper part of the address relative to the program counter.
2. **Second instruction:** addi (add immediate), which computes the lower part of the address and completes the effective address computation.

After the auipc instruction, the pipeline needs two stalls (nops) to ensure the result is ready before the addi instruction uses it.

PC	Machine Code	Basic Code	Original Code
0x0	0x10000E17	auipc x28 65536	la t3, fib_result # Load the address of fib_result into t3
0x4	0x000E0E13	addi x28 x28 0	la t3, fib_result # Load the address of fib_result into t3

```
10000E17
00000013
00000013
000E0E13
000002B3
00100313
00600E93
00000013
00000013
020E8063
00000013
00000013
00000013
005303B3
FFFE8E93
00030293
00038313
FE5FF06F
005E2023
```

Instruction: mv t1, t2

- **Type of Hazard:** RAW (Read After Write) Data Hazard
    - The mv t1, t2 instruction depends on the result of the add t2, t1, t0 operation. Since t2 is written by the add instruction, the pipeline requires **1 stall** (nop) to ensure that t2 is fully updated before being copied to t1.
    - The optimal solution is to rearrange the instructions, where possible, by moving addi t4, t4, -1 after add t2, t1, t0 to minimize latency.

Instruction: beq t4, x0, finish

- **Type of Hazard:** Control Hazard (also includes RAW hazard)
    - If the branch is taken, or if there's a jump: The next three instructions in the pipeline are incorrect. This creates a control hazard! To resolve this, we need to convert the incorrect instructions in the pipeline to nops. This process is called flushing the pipeline.

Now after adding all the required stalls here and rearrangements the following is the full assembly code for pipelined RISC-V without hazard unit as shown in Figure 4-10.

```
.data
fib_result: .word 0      # Reserve space for one word to store the result

.text
main:
    la t3, fib_result    # Load the address of fib_result into t3 (requires 2 manual stalls between translated instructions)
    add t0, x0, x0       # t0 = 0 (1st Fibonacci number)
    addi t1, x0, 1       # t1 = 1 (2nd Fibonacci number)
    li t4, 6             # Load n = 6 into t4
    nop                  # Must stall
    nop                  # Must stall

fib:
    beq t4, x0, finish   # If n == 0, exit loop
    nop                  # Must stall
    nop                  # Must stall
    nop                  # Must stall
    add t2, t1, t0       # t2 = t1 + t0 (next Fibonacci number)
    addi t4, t4, -1      # Decrement n (rearranged)
    mv t0, t1            # t0 = t1 (shift Fibonacci sequence)
    mv t1, t2            # t1 = t2 (shift Fibonacci sequence)
    j fib                # Repeat the loop
finish:
    sw t0, 0(t3)         # Store the final Fibonacci number in fib_result
```

Figure 4-10: Fibonacci Sequence for Pipelined RISC-V without Hazard Unit.

NOTE: 3 nops were added after branching instruction because we did not optimistically assume that the branch would not be taken. As a result, three cycles were required to generate the PCSel signal.

4.3 Simulation Results with Stalling

```
# Value stored in the register (t0): 8
# Value stored in the Data Memory (Address 0): 8
# ** Note: $stop    : RISC_V_tb.v(90)
#    Time: 102 ns  Iteration: 0  Instance: /RISC_V_tb
```

4.4 Implementation of Pipelined RISC-V With Hazard Unit

4.4.1 Handling Data Hazards via Forwarding

The forwarding logic in the hazard unit detects data hazards and selects the correct value from later pipeline stages to resolve dependencies. I modified it to consider the instruction $inst_M$ in the third stage as shown in Figure 4-11, ensuring proper forwarding since hazards can occur across three consecutive instruction. The same applies to the rs2 multiplexer, where forwarding is required to ensure the correct value is used.

	1	2	3	4	5	6
`add t0 t1 t2`	IF	ID	EX	M	**WB**	
`slt t6 t0 t3`		IF	**ID**	EX	M	WB

	1	2	3	4	5	6
`add t0 t1 t2`	IF	ID	**EX**	M	WB	
`slt t6 t0 t3`		IF	ID	**EX**	M	WB

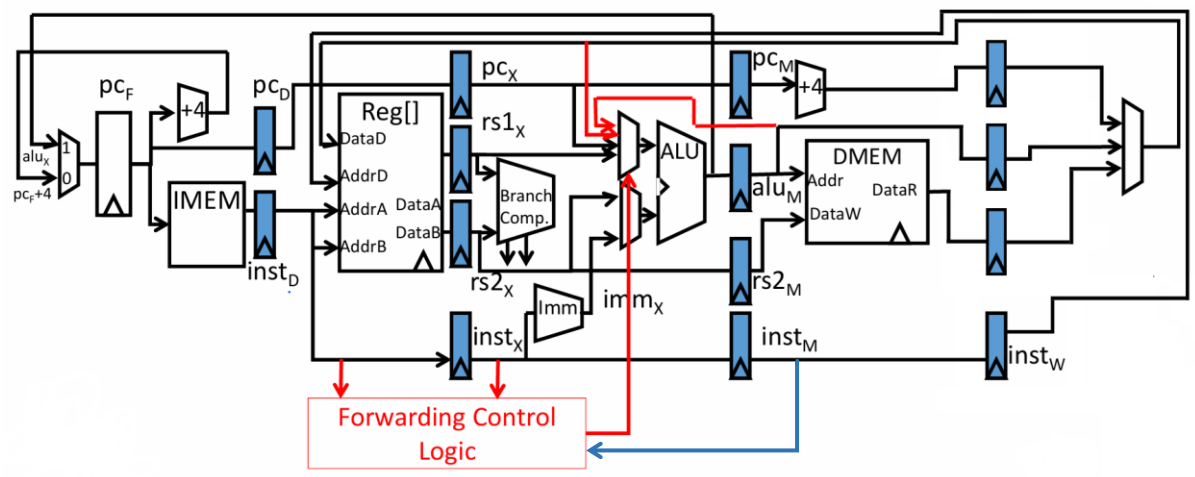


Figure 4-11: Forwarding control logic in hazard unit. Adapted from [5]

I then separated the hazard multiplexer and created an independent one from the original datapath, placing it right after the pipelined DataA and DataB. This allows the new value to be used in both the ALU and the branch comparator, as shown in Figure 4-12.

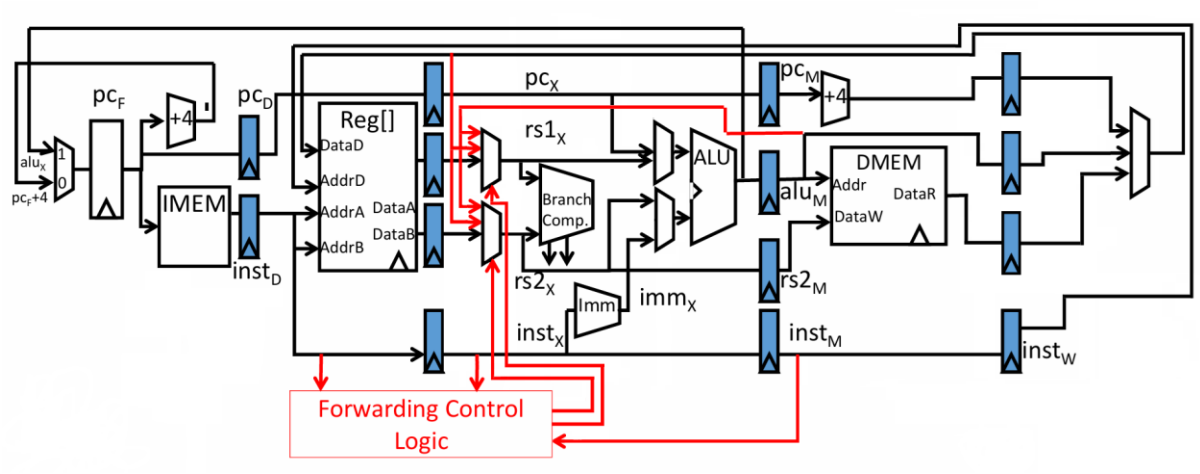


Figure 4-12: Updated forwarding logic ensuring correct data propagation across pipeline stages. Adapted from [5]

●	0x24	0x005303B3	add x7 x6 x5	add t2, t1, t0 # t2 = t1 + t0 (next Fibonacci number)
	0x28	0x00030293	addi x5 x6 0	mv t0, t1 # t0 = t1 (shift Fibonacci sequence)
●	0x2c	0x00038313	addi x6 x7 0	mv t1, t2 # t1 = t2 (shift Fibonacci sequence)

Here we can see that there is data hazard where the destination register x7 (rd) will be read in the 3rd instruction as source register (rs1) so we will need to take the value from the write back stage back to the ALU to perform the addition, as shown in Figure 4-13 and Figure 4-14. This is why $inst_M$ needed to be added to the forwarding condition, ensuring correct data dependency resolution.

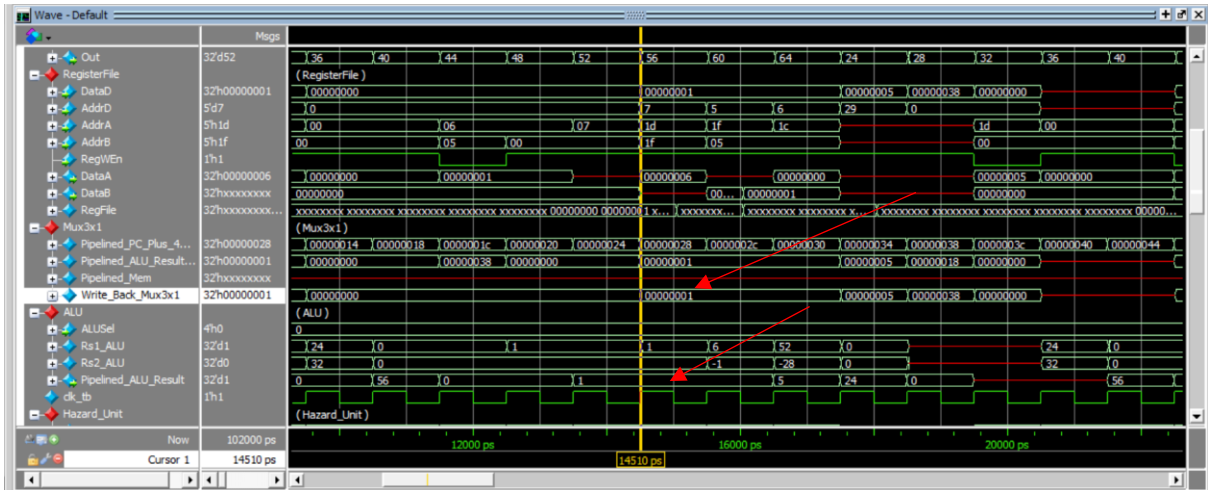


Figure 4-13: ALU result from the previous instruction is available at the write back multiplexer.

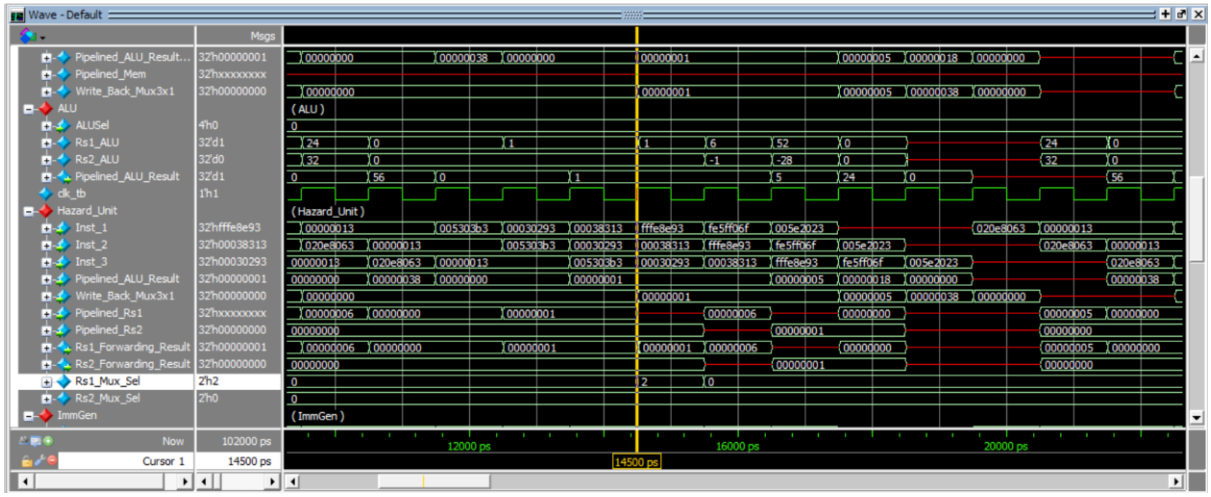


Figure 4-14: Hazard multiplexer for rs1 set to 2, selecting the value from the write back stage.

```
1  li t3, 0
2  addi t4, x0, 8
3  addi t0, x0, 4
4  addi t1, x0, 0
5  add t2, t0, t1
6  beq t2, t0, finish
7  nop
8  nop
9  nop
10 mv t2, t4
11 finish:
12 sw t2, 0(t3)
```

●	0x10	0x006283B3	add x7 x5 x6	add t2, t0, t1
●	0x14	0x00538A63	beq x7 x5 20	beq t2, t0, finish

Here, we can see why it was crucial to place the 3-to-1 multiplexer immediately after the pipeline register output of the register file. This ensures that the computed value of t2 from the addition instruction is available for the next instruction, allowing it to be correctly read by the branch comparator, thereby resolving the data hazard as shown in Figure 4-15 and Figure 4-16.

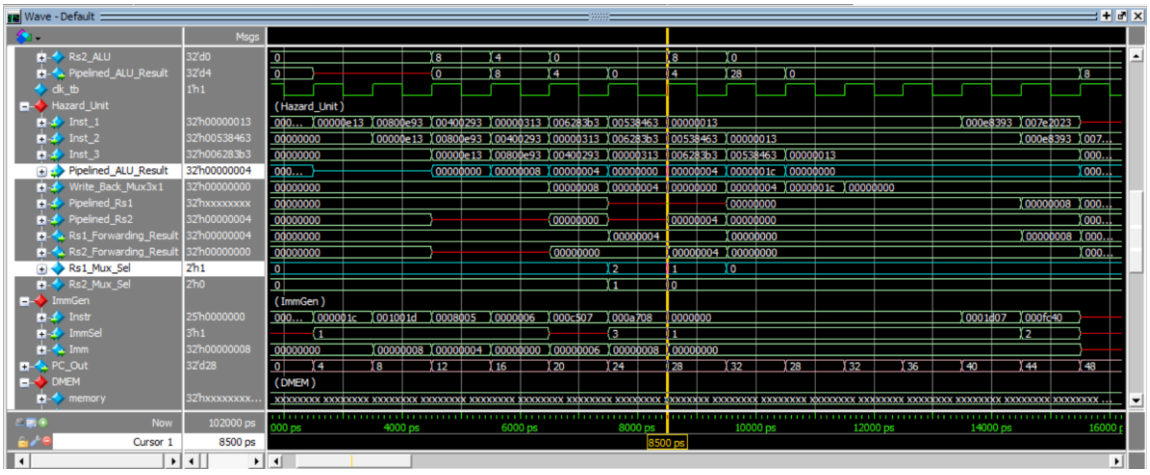


Figure 4-15: ALU result forwarded to the branch comparator for Rs1.

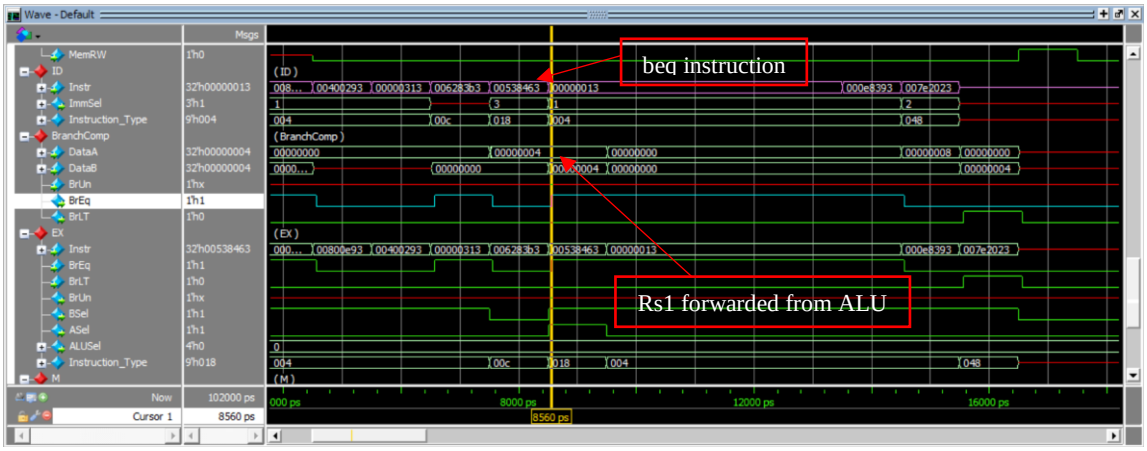
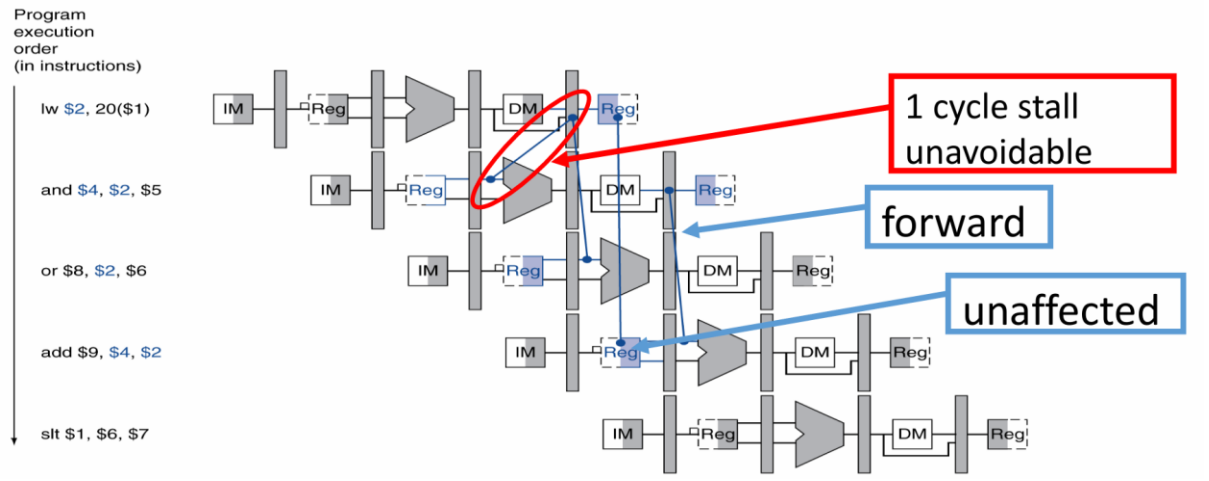
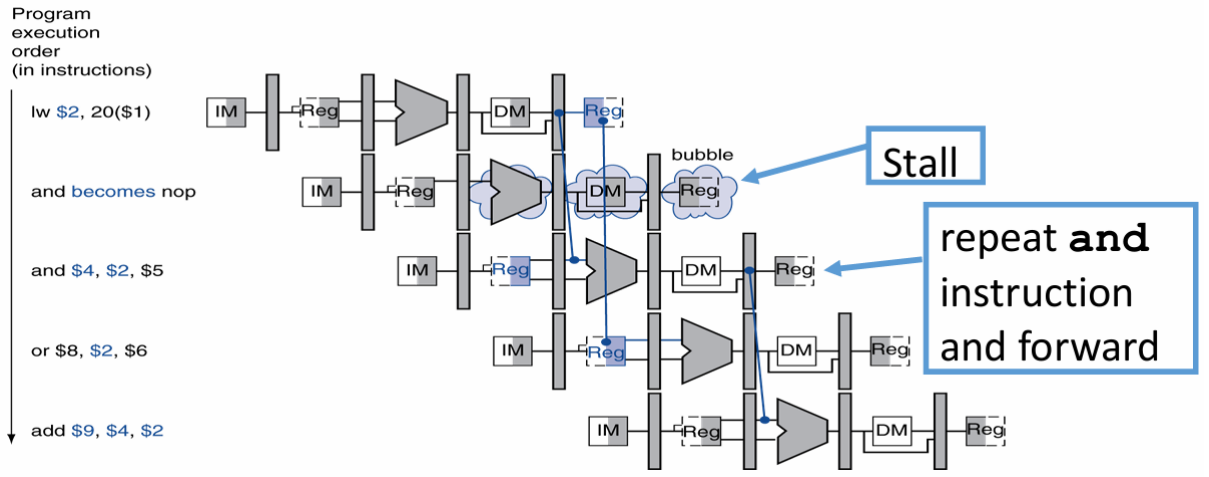


Figure 4-16: Successful forwarding of Rs1.

4.4.2 Resolving Load Hazards

In the lw (load word) instruction, there is an unavoidable stall due to the need for the data to be loaded from memory before it can be used by the subsequent instruction. This stall occurs because the data is not available immediately after the load instruction executes; it requires an additional clock cycle for the data to reach the memory stage and be forwarded to the ALU or registers, delaying the next instruction, as shown below [6].





A load hazard is resolved by detecting a load instruction (e.g., lw) followed by a dependent instruction. In load hazard control, both the PC and instruction registers are disabled to hold their previous values for one clock cycle, effectively stalling the pipeline, as shown in Figure 4-17. This allows the load instruction to complete before the dependent instruction is executed, as shown in Figure 4-18.

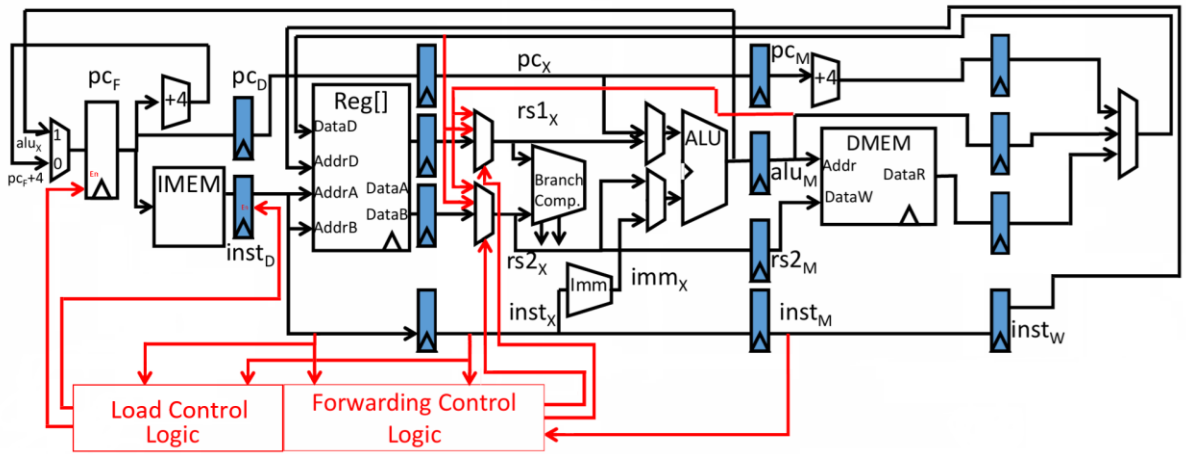


Figure 4-17: Load hazard control: stalling the PC and instruction registers to resolve data dependencies. Adapted from [5]

	0x44	0x005E2023	sw x5 0(x28)	sw t0, 0(t3) # Store the final Fibonacci number in fib_result
	0x48	0x00500F93	addi x31 x0 5	li t6, 5
●	0x4c	0x000E2F83	lw x31 0(x28)	lw t6, 0(t3) # t6 = 8
●	0x50	0x008F8F13	addi x30 x31 8	addi t5, t6, 8 # t5 = 16

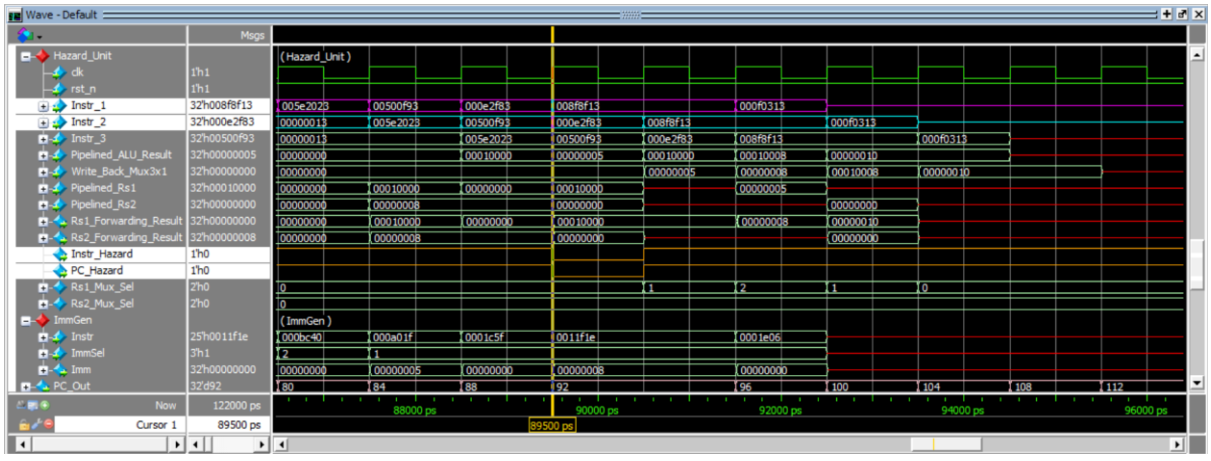


Figure 4-18: Handling load hazards.

4.5 Modified RTL for Pipelined RISC-V

4.5.1 Enabled Register

```
module En_Register #(parameter DATA_WIDTH = 32) (
    input   wire                        clk,
    input   wire                        rst_n,
    input   wire                        En,
    input   wire    [DATA_WIDTH-1:0]    In,
    output  reg     [DATA_WIDTH-1:0]    Out
    );

always@(posedge clk or negedge rst_n)
 begin
  if (!rst_n)
    Out <= 'b0;
  else if (En)
    Out <= In;
 end

endmodule
```

4.5.2 PC

```
`timescale 1ps/1fs

module PC #(parameter DATA_WIDTH = 32)(
    input   wire                        clk,
    input   wire                        rst_n,
    input   wire                        En,     // Added for Load Hazard
    input   wire    [DATA_WIDTH-1:0]    PC_In,
    output  reg     [DATA_WIDTH-1:0]    PC_Out
    );

always@(posedge clk or negedge rst_n)
 begin
  //#10; // Simple Logic Delay
  if(!rst_n)
   PC_Out <= 'b0;
  else if (En)        // Modified for Hazard
   PC_Out <= PC_In;
 end


endmodule
```

4.5.3 Datapath

```
`include "Mux2x1.v"
`include "PC.v"
`include "Add.v"
`include "ImmGen.v"
`include "RegisterFile.v"
`include "BranchComp.v"
`include "ALU.v"
`include "Mux3x1.v"
`include "Register.v"
`include "En_Register.v"
`include "../Hazard_Unit/Hazard_Unit.v"

module Data_Path #(parameter DATA_WIDTH = 32) (
    input   wire                        clk,
    input   wire                        rst_n,
    input   wire                        PCSel,
    input   wire    [2:0]               ImmSel,
    input   wire                        RegWEn,
    input   wire                        BrUn,
    input   wire                        BSel,
    input   wire                        ASel,
    input   wire    [3:0]               ALUSel,
    input   wire                        MemRW,
    input   wire    [1:0]               WBSel,
    input   wire    [31:0]              Instr,
    input   wire    [DATA_WIDTH-1:0]    Mem,
    output  wire    [DATA_WIDTH-1:0]    PC_Next,
    output  wire    [DATA_WIDTH-1:0]    Pipelined_Rs2_2,            // Modified for Pipeline
    output  wire    [DATA_WIDTH-1:0]    Pipelined_Instr_1,          // ================ Pipelined Instruction ================ //
    output  wire    [DATA_WIDTH-1:0]    Pipelined_Instr_2,          // ================ Pipelined Instruction ================ //
    output  wire    [DATA_WIDTH-1:0]    Pipelined_Instr_3,          // ================ Pipelined Instruction ================ //
    output  wire    [DATA_WIDTH-1:0]    Pipelined_Instr_4,          // ================ Pipelined Instruction ================ //
    output  wire                        Pipelined_BrEq,             // -- To Delay BrEq one clk to be used in Memory Control to generate PCSel correctly -- //
    output  wire                        Pipelined_BrLT,             // -- To Delay and BrLT one clk to be used in Memory Control to generate PCSel correctly -- //
    output  wire    [DATA_WIDTH-1:0]    Pipelined_ALU_Result,       // Modified for Piepline
    output  wire                        BrEq,
    output  wire                        BrLT
    );

```

```
    // Internal Wires

    wire [DATA_WIDTH-1:0] PC_Plus_4;

    wire [DATA_WIDTH-1:0] PC_Mux2x1_Out;

    wire [DATA_WIDTH-1:0] Imm;

    wire [DATA_WIDTH-1:0] Write_Back_Mux3x1;

    wire [DATA_WIDTH-1:0] Rs1_ALU;
    wire [DATA_WIDTH-1:0] Rs2_ALU;


    // Pipeline Wires

    wire [DATA_WIDTH-1:0] PC_Next_Reg1;
    wire [DATA_WIDTH-1:0] PC_Next_Reg2;
    wire [DATA_WIDTH-1:0] PC_Next_Reg3;
    wire [DATA_WIDTH-1:0] Pipelined_PC_Plus_4;

    wire [DATA_WIDTH-1:0] Rs1;
    wire [DATA_WIDTH-1:0] Rs2;
    wire [DATA_WIDTH-1:0] Pipelined_Rs1;
    wire [DATA_WIDTH-1:0] Pipelined_Rs2;

    wire [DATA_WIDTH-1:0] ALU_Result;

    wire [DATA_WIDTH-1:0] Pipelined_PC_Plus_4_Out_Mux;
    wire [DATA_WIDTH-1:0] Pipelined_ALU_Result_Mux;
    wire [DATA_WIDTH-1:0] Pipelined_Mem_Mux;

    wire [DATA_WIDTH-1:0] Pipelined_Imm;


    // Hazard Unit Forawrd Wires

    wire [DATA_WIDTH-1:0] Rs1_Forwarding_Result;
    wire [DATA_WIDTH-1:0] Rs2_Forwarding_Result;

    wire                  Instr_Hazard;
    wire                  PC_Hazard;



    ///*****************************************************///
    ////////////////////////// PC Mux ////////////////////////////
    ///*****************************************************///

    Mux2x1 #(.DATA_WIDTH(DATA_WIDTH)) U0_PC_Mux (
        .Sel(PCSel),
        .In0(PC_Plus_4),
        .In1(Pipelined_ALU_Result),      // Modified for Pipeline
        .Out(PC_Mux2x1_Out)
    );

```

```
    ///*******************************************************///
    ////////////////////// PC Register ////////////////////////
    ///*******************************************************///

    PC #(.DATA_WIDTH(DATA_WIDTH)) U0_PC (
        .clk(clk),
        .rst_n(rst_n),
        .En(PC_Hazard),                 // Added for Load Hazard
        .PC_In(PC_Mux2x1_Out),
        .PC_Out(PC_Next)
    );


    // ------------------------------------------------------------------ Pipelined PC ------------------------------------------------------------------ //

    //---------------------------------------------------------//
    //---------------- Pipeline_Registered_PC 1 ---------------//
    //---------------------------------------------------------//

    Register #(.DATA_WIDTH(DATA_WIDTH)) U1_Pipeline_Registered_PC (
        .clk(clk),
        .rst_n(rst_n),
        .In(PC_Next),
        .Out(PC_Next_Reg1)
    );

    //---------------------------------------------------------//
    //---------------- Pipeline_Registered_PC 2 ---------------//
    //---------------------------------------------------------//

    Register #(.DATA_WIDTH(DATA_WIDTH)) U2_Pipeline_Registered_PC (
        .clk(clk),
        .rst_n(rst_n),
        .In(PC_Next_Reg1),
        .Out(PC_Next_Reg2)
    );

    //---------------------------------------------------------//
    //---------------- Pipeline_Registered_PC 3 ---------------//
    //---------------------------------------------------------//

    Register #(.DATA_WIDTH(DATA_WIDTH)) U3_Pipeline_Registered_PC (
        .clk(clk),
        .rst_n(rst_n),
        .In(PC_Next_Reg2),
        .Out(PC_Next_Reg3)
    );

    //---------------------------------------------------------//
    //------------ Adder for Pipeline_Registered_PC -----------//
    //---------------------------------------------------------//

    Add #(.DATA_WIDTH(DATA_WIDTH)) U1_Pipeline_Add (
        .PC(PC_Next_Reg3),
        .PC_Plus_4(Pipelined_PC_Plus_4)
    );
```

```
    //---------------------------------------------------------//
    //---------------- Pipeline_Registered_PC+4 ---------------//
    //---------------------------------------------------------//

    Register #(.DATA_WIDTH(DATA_WIDTH)) U1_Pipeline_PC_Plus4 (
        .clk(clk),
        .rst_n(rst_n),
        .In(Pipelined_PC_Plus_4),
        .Out(Pipelined_PC_Plus_4_Out_Mux)
    );


    ///*******************************************************///
    ////////////////////// Adder for PC ////////////////////////
    ///*******************************************************///

    Add #(.DATA_WIDTH(DATA_WIDTH)) U0_Add (
        .PC(PC_Next),
        .PC_Plus_4(PC_Plus_4)
    );


    // ------------------------------------------------------------------ Pipelined Instruction ------------------------------------------------------------------ //

    //---------------------------------------------------------//
    //------------ Pipeline_Registered_Instruction 1-----------//
    //---------------------------------------------------------//

    En_Register #(.DATA_WIDTH(DATA_WIDTH)) U1_Pipeline_Instruction (
        .clk(clk),
        .rst_n(rst_n),
        .En(Instr_Hazard),      // Added for Load Hazard
        .In(Instr),
        .Out(Pipelined_Instr_1)
    );

    //---------------------------------------------------------//
    //------------ Pipeline_Registered_Instruction 2-----------//
    //---------------------------------------------------------//

    Register #(.DATA_WIDTH(DATA_WIDTH)) U2_Pipeline_Instruction (
        .clk(clk),
        .rst_n(rst_n),
        .In(Pipelined_Instr_1),
        .Out(Pipelined_Instr_2)
    );

    //---------------------------------------------------------//
    //------------ Pipeline_Registered_Instruction 3-----------//
    //---------------------------------------------------------//

    Register #(.DATA_WIDTH(DATA_WIDTH)) U3_Pipeline_Instruction (
        .clk(clk),
        .rst_n(rst_n),
        .In(Pipelined_Instr_2),
        .Out(Pipelined_Instr_3)
    );
```

```verilog

// -- To Delay BrEq and BrLT one clk to be used in Memory Control to generate PCSel correctly -- //

Register #(.DATA_WIDTH(1)) U1_BrEq (
    .clk(clk),
    .rst_n(rst_n),
    .In(BrEq),
    .Out(Pipelined_BrEq)
);

Register #(.DATA_WIDTH(1)) U1_BrLT (
    .clk(clk),
    .rst_n(rst_n),
    .In(BrLT),
    .Out(Pipelined_BrLT)
);


//---------------------------------------------------------//
//----------- Pipeline_Registered_Instruction 4-----------//
//---------------------------------------------------------//

Register #(.DATA_WIDTH(DATA_WIDTH)) U4_Pipeline_Instruction (
    .clk(clk),
    .rst_n(rst_n),
    .In(Pipelined_Instr_3),
    .Out(Pipelined_Instr_4)
);

// ------------------------------------------------------------------------------------------------------------------------------------------------------------ //


///*******************************************************///
/////////////////////// Register File //////////////////////
///*******************************************************///

RegisterFile #(.DATA_WIDTH(DATA_WIDTH)) U0_Register_File (
    .clk(clk),
    .DataD(Write_Back_Mux3x1),
    .AddrD(Pipelined_Instr_4[11:7]),        // Modified for Pipeline
    .AddrA(Pipelined_Instr_1[19:15]),       // Modified for Pipeline and Hazard
    .AddrB(Pipelined_Instr_1[24:20]),       // Modified for Pipeline and Hazard
    .RegWEn(RegWEn),
    .DataA(Rs1),
    .DataB(Rs2)
);


// ------------------------------------------------------------------ Pipelined Rs1 and Rs2 ------------------------------------------------------------------ //

//---------------------------------------------------------//
//---------------- Pipeline_Registered_Rs1 ---------------//
//---------------------------------------------------------//

Register #(.DATA_WIDTH(DATA_WIDTH)) U1_Pipeline_Registered_Rs1 (
    .clk(clk),
    .rst_n(rst_n),
    .In(Rs1),
    .Out(Pipelined_Rs1)
);
```

```verilog
//---------------------------------------------------------//
//---------------- Pipeline_Registered_Rs2 ---------------//
//---------------------------------------------------------//

Register #(.DATA_WIDTH(DATA_WIDTH)) U1_Pipeline_Registered_Rs2 (
    .clk(clk),
    .rst_n(rst_n),
    .In(Rs2),
    .Out(Pipelined_Rs2)
);


// ******************************************************************* Hazard Unit (Forwarding) ***************************************************************** //

///*******************************************************///
///*************** Hazard_Unit Forwarding ***************///
///*******************************************************///

Hazard_Unit #(.DATA_WIDTH(DATA_WIDTH)) U0_Hazard_Unit (
    .clk(clk),
    .rst_n(rst_n),
    .Instr_1(Pipelined_Instr_1),
    .Instr_2(Pipelined_Instr_2),
    .Instr_3(Pipelined_Instr_3),
    .Pipelined_ALU_Result(Pipelined_ALU_Result),
    .Write_Back_Mux3x1(Write_Back_Mux3x1),
    .Pipelined_Rs1(Pipelined_Rs1),
    .Pipelined_Rs2(Pipelined_Rs2),
    .Rs1_Forwarding_Result(Rs1_Forwarding_Result),
    .Rs2_Forwarding_Result(Rs2_Forwarding_Result),
    .Instr_Hazard(Instr_Hazard),
    .PC_Hazard(PC_Hazard)
);

// ************************************************************************************************************************************************************ //


//---------------------------------------------------------//
//--------------- Pipeline_Registered_Rs2_2 --------------//
//---------------------------------------------------------//

Register #(.DATA_WIDTH(DATA_WIDTH)) U1_Pipeline_Registered_Rs2_2 (
    .clk(clk),
    .rst_n(rst_n),
    .In(Rs2_Forwarding_Result),        // Modified for Hazard
    .Out(Pipelined_Rs2_2)
);

// ------------------------------------------------------------------------------------------------------------------------------------------------------------ //


///*******************************************************///
////////////////////// Immediate Generator /////////////////
///*******************************************************///

ImmGen U0_ImmGen (
    .Instr(Pipelined_Instr_1[31:7]),   // Modified for Pipeline and Hazard
    .ImmSel(ImmSel),
    .Imm(Imm)
);
```

```
335
336   // ------------------------------------------------------------------ Pipelined Imm ------------------------------------------------------------------ //
337
338   //-------------------------------------------------------//
339   //---------------- Pipeline_Registered_Imm ---------------//
340   //-------------------------------------------------------//
341
342   Register #(.DATA_WIDTH(DATA_WIDTH)) U1_Pipeline_Registered_Imm (
343       .clk(clk),
344       .rst_n(rst_n),
345       .In(Imm),
346       .Out(Pipelined_Imm)
347   );
348
349
350   ///*****************************************************///
351   ////////////////////// Branch Comparator //////////////////////
352   ///*****************************************************///
353
354   BranchComp #(.DATA_WIDTH(DATA_WIDTH)) U0_BranchComp (
355       .DataA(Rs1_Forwarding_Result),   // Modified for Pipeline and Hazard
356       .DataB(Rs2_Forwarding_Result),   // Modified for Pipeline and Hazard
357       .BrUn(BrUn),
358       .BrEq(BrEq),
359       .BrLT(BrLT)
360   );
361
362   ///*****************************************************///
363   ////////////////////////// Rs1 Mux ////////////////////////////
364   ///*****************************************************///
365
366   Mux2x1 #(.DATA_WIDTH(DATA_WIDTH)) U0_Rs1_Mux (
367       .Sel(ASel),
368       .In0(Rs1_Forwarding_Result),     // Modified for Pipeline and Hazard
369       .In1(PC_Next_Reg2),              // Modified for Pipeline
370       .Out(Rs1_ALU)
371   );
372
373   ///*****************************************************///
374   ////////////////////////// Rs2 Mux ////////////////////////////
375   ///*****************************************************///
376
377   Mux2x1 #(.DATA_WIDTH(DATA_WIDTH)) U0_Rs2_Mux (
378       .Sel(BSel),
379       .In0(Rs2_Forwarding_Result),     // Modified for Pipeline and Hazard
380       .In1(Pipelined_Imm),             // Modified for Pipeline
381       .Out(Rs2_ALU)
382   );
383
384   ///*****************************************************///
385   ////////////////////////// ALU ////////////////////////////
386   ///*****************************************************///
387
388   ALU #(.DATA_WIDTH(DATA_WIDTH)) U0_ALU (
389       .Src_A(Rs1_ALU),
390       .Src_B(Rs2_ALU),
391       .ALUSel(ALUSel),
392       .ALU_Result(ALU_Result)
393   );
394
```

```
395
396   // ------------------------------------------------------------------ Pipelined ALU ------------------------------------------------------------------ //
397
398   //-------------------------------------------------------//
399   //--------------- Pipeline_Registered_ALU 1 --------------//
400   //-------------------------------------------------------//
401
402   Register #(.DATA_WIDTH(DATA_WIDTH)) U1_Pipeline_Registered_ALU (
403       .clk(clk),
404       .rst_n(rst_n),
405       .In(ALU_Result),
406       .Out(Pipelined_ALU_Result)       // Output form Pipeline to DMEM
407   );
408
409   //-------------------------------------------------------//
410   //--------------- Pipeline_Registered_ALU 2 --------------//
411   //-------------------------------------------------------//
412
413   Register #(.DATA_WIDTH(DATA_WIDTH)) U2_Pipeline_Registered_ALU (
414       .clk(clk),
415       .rst_n(rst_n),
416       .In(Pipelined_ALU_Result),
417       .Out(Pipelined_ALU_Result_Mux)   // Output form Pipeline to Mux3X1
418   );
419
420   //-------------------------------------------------------//
421   //---------------------- Pipeline_Mem ---------------------//
422   //-------------------------------------------------------//
423
424   Register #(.DATA_WIDTH(DATA_WIDTH)) U1_Pipeline_Mem (
425       .clk(clk),
426       .rst_n(rst_n),
427       .In(Mem),
428       .Out(Pipelined_Mem_Mux)
429   );
430
431
432   ///*****************************************************///
433   ////////////////////// Write Back Mux //////////////////////
434   ///*****************************************************///
435
436   Mux3x1 #(.DATA_WIDTH(DATA_WIDTH)) U0_WriteBack_Mux (
437       .Sel(WBSel),
438       .In0(Pipelined_Mem_Mux),
439       .In1(Pipelined_ALU_Result_Mux),          // Modified for Pipeline
440       .In2(Pipelined_PC_Plus_4_Out_Mux),       // Modified for Pipeline
441       .Out(Write_Back_Mux3x1)
442   );
443
444
445   endmodule
```

4.5.4 Control Logic

The control logic in the pipelined RISC-V design is divided into five stages, with each stage mirroring the conditions of a single-cycle design. However, in the pipelined version, each stage only generates the output signals for its respective stage. These stages are:

1. Control_Logic_Decode
2. Control_Logic_Execute
3. Control_Logic_Memory
4. Control_Logic_Write.

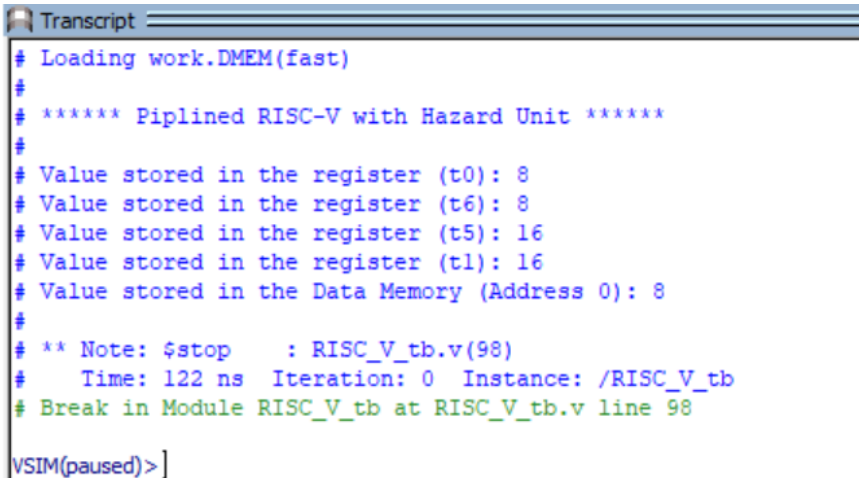
4.5.5 Hazard Unit

```
`include "../Data_Path/Mux3x1.v"
`include "../Data_Path/Mux2x1.v"

module Hazard_Unit #(parameter DATA_WIDTH = 32) (
  input   wire                    clk,
  input   wire                    rst_n,
  input   wire  [DATA_WIDTH-1:0]  Instr_1,
  input   wire  [DATA_WIDTH-1:0]  Instr_2,
  input   wire  [DATA_WIDTH-1:0]  Instr_3,
  input   wire  [DATA_WIDTH-1:0]  Pipelined_ALU_Result,
  input   wire  [DATA_WIDTH-1:0]  Write_Back_Mux3x1,
  input   wire  [DATA_WIDTH-1:0]  Pipelined_Rs1,
  input   wire  [DATA_WIDTH-1:0]  Pipelined_Rs2,
  output  wire  [DATA_WIDTH-1:0]  Rs1_Forwarding_Result,
  output  wire  [DATA_WIDTH-1:0]  Rs2_Forwarding_Result,
  output  reg                     Instr_Hazard,
  output  reg                     PC_Hazard
  );


// ******************************************************** Data Hazard ********************************************************** //

reg [1:0] Rs1_Mux_Sel;
reg [1:0] Rs2_Mux_Sel;


// Instr_1 for rs1 or rs2
// Instr_2 and Inst_3 for rd



// Note: I used a sequential always block here to delay the action to the next clock edge,
// ensuring that the multiplexer select signal is ready when the next instruction causing
// the hazard reaches the Execute stage.


// Rs1 Forwarding Selection
always@(posedge clk or negedge rst_n)
 begin
  if (!rst_n)
   begin
    Rs1_Mux_Sel <= 2'd0;
   end
  else
   begin
    if (Instr_1[19:15] == 5'b0 || Instr_2[11:7] == 5'b0 || Instr_3[11:7] == 5'b0)
     begin
      Rs1_Mux_Sel <= 2'd0;                    // No forwarding
     end
    else
     begin
      if (Instr_2[11:7] == Instr_1[19:15])
        Rs1_Mux_Sel <= 2'd1;                  // Take Result from ALU
      else if (Instr_3[11:7] == Instr_1[19:15])
        Rs1_Mux_Sel <= 2'd2;                  // Take Result from Write Back
      else
        Rs1_Mux_Sel <= 2'd0;                  // No Forwarding
     end
   end
 end
```

```
// Rs2 Forwarding Selection
always@(posedge clk or negedge rst_n)
 begin
  if (!rst_n)
   begin
    Rs2_Mux_Sel <= 2'd0;
   end
  else
   begin
    if (Instr_1[24:20] == 5'b0 || Instr_2[11:7] == 5'b0 || Instr_3[11:7] == 5'b0 || Instr_1[6:2] == 5'b00100) // In I-type instruction, rs2 bits are repurposed for the immediate value. So it will be misleading if rs2 was equal to rd
     begin                                                                                                    // Therefore, the opcode of I-type instructions is included to handle this issue (effective 5 bits )
      Rs2_Mux_Sel <= 2'd0;                    // No forwarding
     end
    else
     begin
      if (Instr_2[11:7] == Instr_1[24:20])
        Rs2_Mux_Sel <= 2'd1;                  // Take Result from ALU
      else if (Instr_3[11:7] == Instr_1[24:20])
        Rs2_Mux_Sel <= 2'd2;                  // Take Result from Write Back
      else
        Rs2_Mux_Sel <= 2'd0;                  // No Forwarding
     end
   end
 end


Mux3x1 Rs1_Mux(.In0(Pipelined_Rs1), .In1(Pipelined_ALU_Result), .In2(Write_Back_Mux3x1), .Sel(Rs1_Mux_Sel), .Out(Rs1_Forwarding_Result));
Mux3x1 Rs2_Mux(.In0(Pipelined_Rs2), .In1(Pipelined_ALU_Result), .In2(Write_Back_Mux3x1), .Sel(Rs2_Mux_Sel), .Out(Rs2_Forwarding_Result));


// ********************************************************* Load Hazard ********************************************************** //


always@(*)
 begin
  if ( Instr_2[6:0] == 7'b0000011 && (Instr_2[11:7] == Instr_1[19:15] || Instr_2[11:7] == Instr_1[24:20]) )
   begin
    Instr_Hazard <= 1'b0;   // Stall Instruction
    PC_Hazard <= 1'b0;      // Stall PC
   end
  else
   begin
    Instr_Hazard <= 1'b1;
    PC_Hazard <= 1'b1;
   end
 end


endmodule
```

4.6 Simulation Results of Pipelined RISC-V with Hazard Unit

```
.data
fib_result: .word 0      # Reserve space for one word to store the result

.text
main:
    la t3, fib_result    # Load the address of fib_result into t3 (requires 2 manual stalls between translated instructions)
    add t0, x0, x0       # t0 = 0 (1st Fibonacci number)
    addi t1, x0, 1       # t1 = 1 (2nd Fibonacci number)
    li t4, 6             # Load n = 6 into t4

fib:
    beq t4, x0, finish   # If n == 0, exit loop
    nop
    nop
    nop
    add t2, t1, t0       # t2 = t1 + t0 (next Fibonacci number)
    mv t0, t1            # t0 = t1 (shift Fibonacci sequence)
    mv t1, t2            # t1 = t2 (shift Fibonacci sequence)
    addi t4, t4, -1      # Decrement n
    j fib                # Repeat the loop
    nop
    nop
    nop
finish:
    sw t0, 0(t3)         # Store the final Fibonacci number in fib_result
    li t6, 5             # t6 = 5
    lw t6, 0(t3)         # t6 = 8
    addi t5, t6, 8       # t5 = 16
    addi t1, t5, 0       # t1 = 16
```



5. References

[1] N. Garcia and B. Nikolić, CS61C: Datapath II, Lecture 31, University of California, Berkeley, Fall 2018.

[2] Dan Garcia, Bora Nikolić, and Peyrin Kao, CS61C: Pipelining, Lectures 21–22, University of California, Berkeley, Fall 2024.

[3] "reference-card.pdf." Accessed: Jan. 29, 2025. [Online]. Available: https://cs61c.org/sp25/pdfs/resources/reference-card.pdf

[4] N. Garcia and B. Nikolić, CS61C: RISC-V Formats II, Lecture 11, University of California, Berkeley, Fall 2018.

[5] N. Garcia and B. Nikolić, CS61C: Machine Structures, Lecture 32 – Pipeline II, University of California, Berkeley, Fall 2018.

[6] N. Garcia and B. Nikolić, CS61C: Machine Structures, Lecture 32 – Pipeline III, University of California, Berkeley, Fall 2018.

[7] D. Harris and S. Harris, Digital Design and Computer Architecture: RISC-V Edition, 2nd ed., Burlington, MA: Morgan Kaufmann, 2021.