



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	3
1.1	Data Path.....	3
1.1.1	Mux2x1	4
1.1.2	PC.....	4
1.1.3	Add.....	5
1.1.4	Register File	5
1.1.5	Immediate Generator.....	6
1.1.6	Branch Comparator.....	6
1.1.7	ALU	7
1.1.8	Mux3x1	8
1.2	RISC-V Datapath	9
1.3	RISC-V Control Logic	12
1.3.1	R-Format	0
1.3.2	I-Format	1
1.3.3	S-Format	1
1.3.4	B-Format	2
1.3.5	U-Format.....	2
1.3.6	J-Format	2
1.4	RISC-V Single Cycle Top Module	3
1.5	Memories	5
1.5.1	Instruction Memory.....	5
1.5.2	Data Memory	5
2.	Generating Fibonacci Machine Code for RISC-V using Venus	7
2.1	Fibonacci Sequence Assembly Code	7
3.	Verification of RISC-V RTL	10
3.1	RISC-V Simple Testbench.....	10
3.2	Simulation Results	11
3.2.1	Testing Rest of the Instructions (store word)	11
4.	Pipelined RISC-V Archtiecture	13
4.1	Pipeline Hazards	17
4.1.1	Data Hazards	17
4.1.2	Control Hazards	17
4.1.3	Structural Hazards.....	17
4.2	Implementation of Pipelined RISC-V Without Hazard Unit: Fixing Hazards by Stalling ...	19
4.3	Simulation Results with Stalling	20
4.4	Implementation of Pipelined RISC-V With Hazard Unit	21

4.4.1	Handling Data Hazards via Forwarding.....	21
4.4.2	Resolving Load Hazards	25
4.5	Modified RTL for Pipelined RISC-V	27
4.5.1	Enabled Register	27
4.5.2	PC.....	27
4.5.3	Datapath	28
4.5.4	Control Logic.....	32
4.5.5	Hazard Unit.....	33
4.5.6	Simulation Results of Pipelined RISC-V with Hazard Unit	34
5.	References.....	35

1. RISC-V Single Cycle Architecture

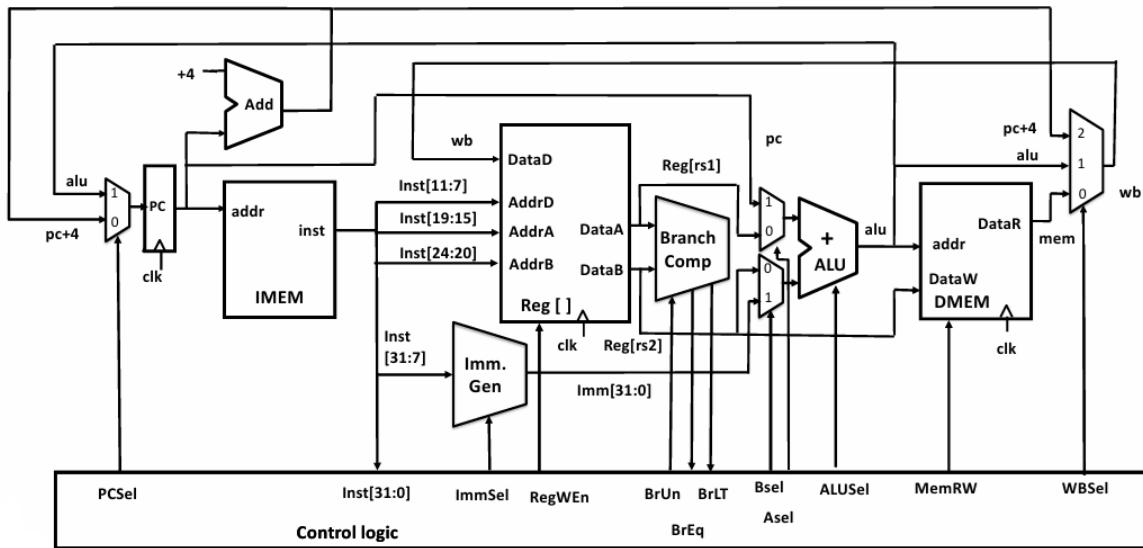


Figure 1-1: RISC-V single cycle. [1]

1.1 Data Path

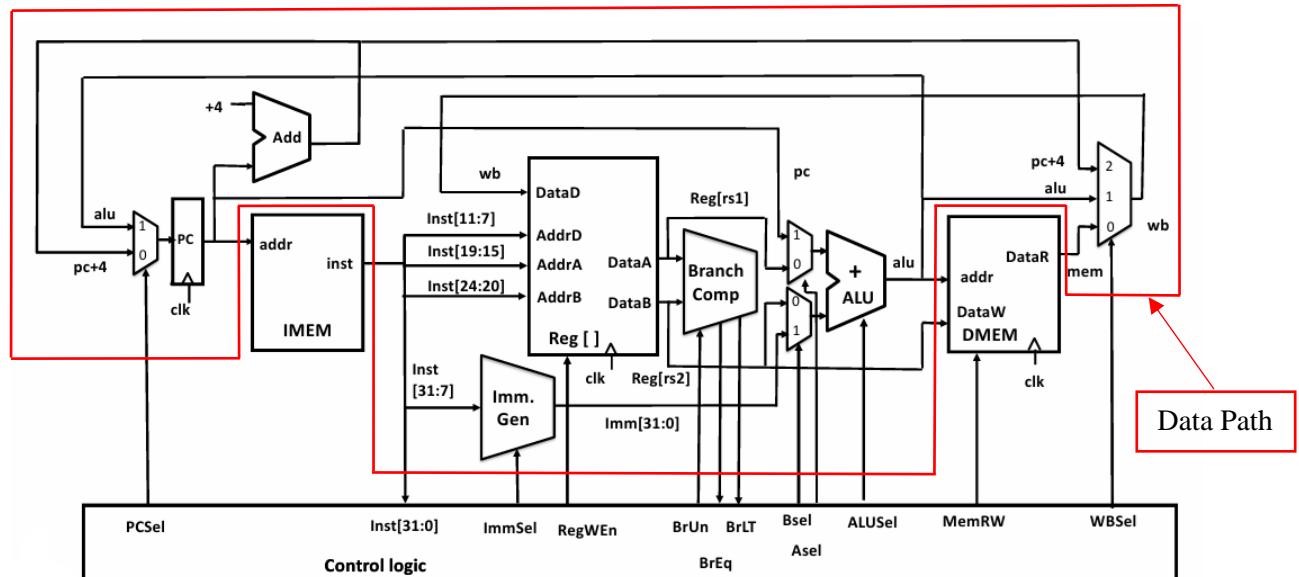


Figure 1-2: RISC-V datapath. Adapted from [1]

1.1.1 Mux2x1

```
1  `timescale 1ps/1fs
2
3  module Mux2x1 #(parameter DATA_WIDTH = 32)(
4      input  wire           Sel,
5      input  wire [DATA_WIDTH-1:0] In0,
6      input  wire [DATA_WIDTH-1:0] In1,
7      output reg  [DATA_WIDTH-1:0] Out
8  );
9
10 always@(*)
11 begin
12     #10; // Simple Logic Delay
13     if(Sel)
14         Out = In1;
15     else
16         Out = In0;
17 end
18
19
20 endmodule
```

1.1.2 PC

```
1  `timescale 1ps/1fs
2
3  module PC #(parameter DATA_WIDTH = 32)(
4      input  wire           clk,
5      input  wire           rst_n,
6      input  wire [DATA_WIDTH-1:0] PC_In,
7      output reg  [DATA_WIDTH-1:0] PC_Out
8  );
9
10 always@(posedge clk or negedge rst_n)
11 begin
12     //#10; // Simple Logic Delay
13     if(!rst_n)
14         PC_Out <= 'b0;
15     else
16         PC_Out <= PC_In;
17 end
18
19
20 endmodule
```

1.1.3 Add

```
1  `timescale 1ps/1fs
2
3  module Add #(parameter DATA_WIDTH = 32)(
4      input  wire      [DATA_WIDTH-1:0]      PC,
5      output reg       [DATA_WIDTH-1:0]      PC_Plus_4
6  );
7
8  always@(*)
9  begin
10     #10 PC_Plus_4 = PC + 3'd4; // Simple Logic Delay
11 end
12
13 endmodule
```

1.1.4 Register File

```
1  `timescale 1ps/1fs
2
3  module RegisterFile #(parameter DATA_WIDTH = 32) (
4      input  wire          clk,
5      input  wire [DATA_WIDTH-1:0] DataD,           // Data to be written to the destination register
6      input  wire [4:0]      AddrD,                // Destination register address
7      input  wire [4:0]      AddrA,                // Source register 1 address
8      input  wire [4:0]      AddrB,                // Source register 2 address
9      input  wire          RegWEn,
10     output wire [DATA_WIDTH-1:0] DataA,           // Data from source register 1
11     output wire [DATA_WIDTH-1:0] DataB           // Data from source register 2
12 );
13
14 reg [DATA_WIDTH-1:0] RegFile [0:31];
15
16
17 // Read ports and hard wire x0 to '0'
18 assign DataA = (AddrA == 0) ? {DATA_WIDTH{1'b0}} : RegFile[AddrA];
19 assign DataB = (AddrB == 0) ? {DATA_WIDTH{1'b0}} : RegFile[AddrB];
20
21 // Write port (write on the negedge clock "later needed in pipeline") and prevent writing to x0
22 always @(negedge clk)
23 begin
24     #50; // Major Stage Delay
25     if (RegWEn && AddrD != 5'b0)
26         RegFile[AddrD] <= DataD;
27 end
28
29 endmodule
```

1.1.5 Immediate Generator

```

1  `timescale 1ps/1fs
2
3  /*
4   * Truth Table for ImmSel:
5   +-----+-----+
6   | ImmSel Value| Type    | Description
7   +-----+-----+
8   | 3'b001     | I-Format| Immediate
9   | 3'b010     | S-Format| Store Instructions
10  | 3'b011     | B-Format| Branch Instructions
11  | 3'b100     | J-Format| Jump
12  | 3'b101     | U-Format| Upper Immediate
13  +-----+-----+
14 */
15
16 module ImmGen (
17   input wire [31:7] Instr,
18   input wire [2:0] ImmSel,
19   output reg [31:0] Imm
20 );
21
22 always @(*)
23 begin
24   #50; // Major Stage Delay
25   casex (ImmSel)
26   3'b001 : Imm = {{21{Instr[31]}}, Instr[30:20]]; // I-type (includes JALR)
27   3'b010 : Imm = {{21{Instr[31]}}, Instr[30:25], Instr[11:7]]; // S-type
28   3'b011 : Imm = {{20{Instr[31]}}, Instr[7], Instr[30:25], Instr[11:8], 1'b0}; // B-type
29   3'b100 : Imm = {{12{Instr[31]}}, Instr[19:12], Instr[20], Instr[30:21], 1'b0}; // J-type
30   3'b101 : Imm = {{13{Instr[31]}}, Instr[30:12]]; // U-type
31   default: Imm = 32'bx; // to check in testbench for illegal cases
32 endcase
33 end
34
35 endmodule

```

1.1.6 Branch Comparator

```

1  `timescale 1ps/1fs
2
3  module BranchComp #(parameter DATA_WIDTH = 32) (
4    input wire [DATA_WIDTH-1:0] DataA,
5    input wire [DATA_WIDTH-1:0] DataB,
6    input wire BrUn,
7    output reg BrEq,
8    output reg BrLT
9  );
10
11 always@(*)
12 begin
13   #50; // Major Stage Delay
14   if (BrUn == 1'b1)
15     begin
16       if ($signed(DataA) < $signed(DataB))
17         BrLT = 1'b1;
18       else
19         BrLT = 1'b0;
20       if ($signed(DataA) == $signed(DataB))
21         BrEq = 1'b1;
22       else
23         BrEq = 1'b0;
24     end
25   else
26     begin
27       if ((DataA) < (DataB))
28         BrLT = 1'b1;
29       else
30         BrLT = 1'b0;
31       if ((DataA) == (DataB))
32         BrEq = 1'b1;
33       else
34         BrEq = 1'b0;
35     end
36   end
37
38 // assign #10 BrEq = ( ( BrUn == 1'b1 && ($signed(DataA) == $signed(DataB)) ) || ( BrUn == 1'b0 && ((DataA) == $(DataB)) ) ) ? 1'b1 : 1'b0;
39 // assign #10 BrLT = ( ( BrUn == 1'b1 && ($signed(DataA) < $signed(DataB)) ) || ( BrUn == 1'b0 && ((DataA) < $(DataB)) ) ) ? 1'b1 : 1'b0;
40
41 endmodule

```

1.1.7 ALU

```

1 `timescale 1ps/1fs
2
3 /*
4  Truth Table for ALUSel:
5 +-----+
6 | ALUSel | Operation |
7 +-----+
8 | 4'b0000 | Add      |
9 | 4'b1000 | Subtract |
10 | 4'b0001 | Shift Left Logical (SLL) |
11 | 4'b0010 | Set Less Than signed (SLT) |
12 | 4'b0011 | Set Less Than unsigned (SLTU) |
13 | 4'b0100 | XOR      |
14 | 4'b0101 | Shift Right Logical (SRL) |
15 | 4'b1101 | Shift Right Arithmetic (SRA) |
16 | 4'b0110 | OR       |
17 | 4'b0111 | AND      |
18 | default  | Illegal (Till now) |
19 +-----+
20 */
21
22 module ALU #(parameter DATA_WIDTH = 32) (
23   input wire [DATA_WIDTH-1:0] Src_A,
24   input wire [DATA_WIDTH-1:0] Src_B,
25   input wire [3:0]          ALUSel,
26   output reg [DATA_WIDTH-1:0] ALU_Result
27 );
28
29 wire [DATA_WIDTH-1:0] Src_B_inv_or_not;
30 wire [DATA_WIDTH-1:0] result;
31
32 assign Src_B_inv_or_not = ALUSel[3] ? ~Src_B : Src_B;
33 assign result = Src_A + Src_B_inv_or_not + ALUSel[3];
34
35
36 always @(*)
37 begin
38   #50; // Major Stage Delay
39   casex (ALUSel[2:0])
40     3'b000 : ALU_Result = result; // add & sub
41   /*
42    modification is needed for ALUSel
43
44    4'b0001: ALU_Result = Src_A << Src_B;           // sll
45    4'b0010: ALU_Result = ($signed(Src_A) < $signed(Src_B))? 32'b1 : 32'b0; //slt
46    4'b0011: ALU_Result = (Src_A < Src_B)? 32'b1 : 32'b0; //sltu
47    4'b0100: ALU_Result = Src_A ^ Src_B;             // xor
48    4'b0101: ALU_Result = Src_A >> Src_B;           // srl
49    4'b1101: ALU_Result = Src_A >>> Src_B;          // sra
50  */
51   3'b110 : ALU_Result = Src_A | Src_B;
52   3'b111 : ALU_Result = Src_A & Src_B;
53   default : ALU_Result = 'bx;                      // to check in testbench for illegal cases if happened
54 endcase
55 end
56
57 endmodule

```

1.1.8 Mux3x1

```
1  `timescale 1ps/1fs
2
3  /*
4   * Truth Table for WBSel:
5   * +-----+-----+-----+
6   * | WBSel Value | Source    | Description          |
7   * +-----+-----+-----+
8   * | 2'b00      | Mem       | Memory Read Data   |
9   * | 2'b01      | ALU        | ALU Result          |
10  * | 2'b10      | PC+4      | Return Address      |
11  * +-----+-----+-----+
12 */
13
14 module Mux3x1 #(parameter DATA_WIDTH = 32)(
15   input wire [1:0]           Sel,
16   input wire [DATA_WIDTH-1:0] In0,
17   input wire [DATA_WIDTH-1:0] In1,
18   input wire [DATA_WIDTH-1:0] In2,
19   output reg [DATA_WIDTH-1:0] Out
20 );
21
22 always @(*)
23 begin
24   #10; // Simple Logic Delay
25   case (Sel)
26     2'b00 : Out = In0;
27     2'b01 : Out = In1;
28     2'b10 : Out = In2;
29     default: Out = {DATA_WIDTH{1'b0}};
30   endcase
31 end
32
33 endmodule
```

1.2 RISC-V Datapath

```
1      `include "Mux2x1.v"
2      `include "PC.v"
3      `include "Add.v"
4      `include "ImmGen.v"
5      `include "RegisterFile.v"
6      `include "BranchComp.v"
7      `include "ALU.v"
8      `include "Mux3x1.v"
9
10     module Data_Path #(parameter DATA_WIDTH = 32) (
11         input  wire          clk,
12         input  wire          rst_n,
13         input  wire          PCSel,
14         input  wire [2:0]     ImmSel,
15         input  wire          RegWEen,
16         input  wire          BrUn,
17         input  wire          BSel,
18         input  wire          ASel,
19         input  wire [3:0]     ALUSel,
20         input  wire          MemRW,
21         input  wire [1:0]     WBSel,
22         input  wire [31:0]    Instr,
23         input  wire [DATA_WIDTH-1:0] Mem,
24         output wire [DATA_WIDTH-1:0] PC_Next,
25         output wire [DATA_WIDTH-1:0] ALU_Result,
26         output wire [DATA_WIDTH-1:0] Rs2,
27         output wire          BrEq,
28         output wire          BrLT
29     );
30
31
32     // Internal Wires
33
34     wire [DATA_WIDTH-1:0] PC_Plus_4;
35
36     wire [DATA_WIDTH-1:0] PC_Mux2x1_Out;
37
38     wire [DATA_WIDTH-1:0] Imm;
39
40     wire [DATA_WIDTH-1:0] Write_Back_Mux3x1;
41
42     wire [DATA_WIDTH-1:0] Rs1;
43
44     wire [DATA_WIDTH-1:0] Rs1_ALU;
45     wire [DATA_WIDTH-1:0] Rs2_ALU;
46
47
```

```

49
50     //*****PC Mux*****///
51     //////////////////// PC Mux ///////////////////////
52     //*****PC Register*****///
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     //*****PC Register*****///
62     //////////////////// PC Register ///////////////////////
63     //*****PC Register*****///
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     //*****PC Adder*****///
73     //////////////////// Adder for PC ///////////////////////
74     //*****PC Adder*****///
75
76     Add #( .DATA_WIDTH(DATA_WIDTH) ) U0_Add (
77         .PC(PC_Next),
78         .PC_Plus_4(PC_Plus_4)
79     );
80
81     //*****Immediate Generator*****///
82     //////////////////// Immediate Generator ///////////////////////
83     //*****Immediate Generator*****///
84
85     ImmGen U0_ImmGen (
86         .Instr(Instr[31:7]),
87         .ImmSel(ImmSel),
88         .Imm(Imm)
89     );
90
91     //*****Register File*****///
92     //////////////////// Register File ///////////////////////
93     //*****Register File*****///
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.

The figure shows the RV32I instruction set mapping from 32-bit binary representations to various RISC-V instructions. The columns represent the 32-bit binary fields: imm[31:12], imm[31:12], imm[20:10:1|11|19:12], imm[11:0], rs1, rs2, rd, and LUI/AUIPC. Red arrows point to specific fields: one arrow points to the imm[31:12] field, another to the imm[20:10:1|11|19:12] field, and a third to the rd field. The rows list instructions like BEQ, BNE, BLT, etc., with their corresponding binary codes.

inst[30]	inst[14:12]	inst[6:2]				
00000000	shamt	rs1	001	rd	0010011	SLLI
00000000	shamt	rs1	101	rd	0010011	SRLI
01000000	shamt	rs1	101	rd	0010011	SRAI
00000000	rs2	rs1	000	rd	0110011	ADD
01000000	rs2	rs1	000	rd	0110011	SUB
00000000	rs2	rs1	001	rd	0110011	SLL
00000000	rs2	rs1	010	rd	0110011	SLT
00000000	rs2	rs1	011	rd	0110011	SLTU
00000000	rs2	rs1	100	rd	0110011	XOR
00000000	rs2	rs1	101	rd	0110011	SRL
01000000	rs2	rs1	101	rd	0110011	SRA
00000000	rs2	rs1	110	rd	0110011	OR
00000000	rs2	rs1	111	rd	0110011	AND
00000000	pred	pred	000	00000	00000	FENCE
00000000	0000	0000	001	00000	00000	FENCE.I
00000000	000000000000	00000	000	00000	1110011	ECALL
00000000	000000000001	00000	000	00000	1110011	EBREAK
00000000	csr	rs1	001	rd	1110011	CSRWR
00000000	csr	rs1	011	rd	1110011	CSRRC
00000000	csr	zimm	101	rd	1110011	CSRRCI
00000000	csr	zimm	110	rd	1110011	CSRRSI
00000000	csr	zimm	111	rd	1110011	CSRRCI

Figure 1-3: RV32I 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 **BSel**: 0 = Reg, 1 = Imm

For **ASel**: 0 = Reg, 1 = PC

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
69
```

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 // srsl
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

```

106
109  **** I_Format ****
110
111  9'bx_000_00100 : begin // addi
112      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;
113      end
114  /*
115  9'bx_010_00100 : begin // slti
116      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;
117      end
118  9'bx_011_00100 : begin // sltiu
119      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;
120      end
121  9'bx_100_00100 : begin // xor
122      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;
123      end
124  /*
125  9'bx_110_00100 : begin // ori
126      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;
127      end
128  9'bx_111_00100 : begin // andi
129      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;
130      end
131  /*
132  9'b0_001_00100 : begin // slli
133      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;
134      end
135  9'b0_101_00100 : begin // srli
136      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;
137      end
138  9'b0_101_00100 : begin // srai
139      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;
140      end
141  /*
142  **** I_Format Contd. Load ****
143  /*
144  9'bx_000_00000 : begin // lb
145      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;
146      end
147  9'bx_001_00000 : begin // lh
148      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;
149      end
150  /*
151  9'bx_010_00000 : begin // lw
152      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;
153      end
154  /*
155  9'bx_100_00000 : begin // lbu
156      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;
157      end
158  9'bx_101_00000 : begin // lhu
159      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;
160      end
161  /*
162  **** I_Format Contd. JLR ****
163
164  9'bx_xxx_11001 : begin // jalr
165      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;
166      end
167

```

1.3.3 S-Format

```

167
168  **** S_Format ****
169  /*
170  9'bx_000_01000 : begin // sb
171      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;
172      end
173  9'bx_001_01000 : begin // sh
174      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;
175      end
176  /*
177  9'bx_010_01000 : begin // sw
178      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;
179      end

```

1.3.4 B-Format

```
183      /***** B_Format *****/
184
185 9'bx_000_11000 : begin // beq
186    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;
187    end
188 9'bx_001_11000 : begin // bne
189    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;
190    end
191  /*
192 9'bx_100_11000 : begin // blt
193    if (BrLt)
194      begin
195        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;
196      end
197    else
198      begin
199        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;
200      end
201    end
202
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
207 9'bx_110_11000 : begin // bltu
208    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;
209
210 9'bx_111_11000 : begin // bgeu
211    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;
212
213
```

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
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 //***** Data Path *****
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 //***** Control Logic *****
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

```
1  `timescale 1ps / 1fs
2
3  module IMEM #(parameter DATA_WIDTH = 32, MEM_SIZE = 256) (
4      input  wire      [31:0] Addr,
5      output reg       [31:0] Instr
6  );
7
8  reg [DATA_WIDTH-1:0] Mem [0:MEM_SIZE-1];
9
10 initial
11 begin
12     $readmemh("Memory/instructions.txt", Mem); // Load instructions from a file
13 end
14
15 // Instruction Memory Delay
16 always @(*) begin
17     #100 Instr = Mem[Addr];
18 end
19
20 endmodule
```

1.5.2 Data Memory

```
1  `timescale 1ps/1fs
2
3  module DMEM #(parameter DATA_WIDTH = 32, MEM_SIZE = 256) (
4      input  wire          clk,
5      input  wire          MemRW,
6      input  wire [7:0]    Addr,
7      input  wire [DATA_WIDTH-1:0] DataW,           // Data to be written
8      output reg [DATA_WIDTH-1:0] DataR            // Data read from memory
9  );
10
11
12 reg [DATA_WIDTH-1:0] memory [0:MEM_SIZE-1];
13
14 // Memory Read Operation (Combinational)
15 always @(*)
16 begin
17     #100;
18     if (!MemRW)
19         DataR = memory[Addr[7:0]];
20     else
21         DataR = 32'bxx;
22 end
23
24 // Memory Write Operation
25 always @ (posedge clk)
26 begin
27     #100
28     if (MemRW)
29         memory[Addr[7:0]] <= DataW;
30 end
31
32
33 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: 0001000000000000000000000000000100
- 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

```
1 .data
2 fib_result: .word 0      # Reserve space for one word to store the result
3
4 .text
5 main:
6     add t0, x0, x0      # t0 = 0 (1st Fibonacci number)
7     addi t1, x0, 1       # t1 = 1 (2nd Fibonacci number)
8     li t4, 6             # Load n = 6 into t4
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 finish:
18    la t3, fib_result    # Load the address of fib_result into t3
19    sw t0, 0(t3)          # Store the final Fibonacci number in fib_result
```

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

t0 (x5) 0x000000008

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 `0x000010000`, adding instructions before the `la` may result in an address like `0x000010004` 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.

```
1 .data
2 fib_result: .word 0      # Reserve space for one word to store the result
3
4 .text
5 main:
6     la t3, fib_result    # Load the address of fib_result into t3
7     add t0, x0, x0        # t0 = 0 (1st Fibonacci number)
8     addi t1, x0, 1        # t1 = 1 (2nd Fibonacci number)
9     li t4, 6              # Load n = 9 into t4
10
11 fib:
12     beq t4, x0, finish   # If n == 0, exit loop
13     add t2, t1, t0        # t2 = t1 + t0 (next Fibonacci number)
14     mv t0, t1              # t0 = t1 (shift Fibonacci sequence)
15     mv t1, t2              # t1 = t2 (shift Fibonacci sequence)
16     addi t4, t4, -1        # Decrement n
17     j fib                 # Repeat the loop
18
19 finish:
20     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

```
1  `include "Memory/IMEM.v"
2  `include "Memory/DMEM.v"
3
4  `timescale 1ps/1fs
5
6  module RISC_V_tb ();
7
8  //////////////////////////////////////////////////////////////////// Parameters
9  ///////////////////////////////////////////////////////////////////
10 ///////////////////////////////////////////////////////////////////
11 parameter CLK_PERIOD = 1000; // 1ns
12 parameter DATA_WIDTH = 32;
13 parameter MEM_SIZE   = 256;
14
15 //////////////////////////////////////////////////////////////////// DUT Signals
16 ///////////////////////////////////////////////////////////////////
17 ///////////////////////////////////////////////////////////////////
18 ///////////////////////////////////////////////////////////////////
19
20 reg          clk_tb;
21 reg          rst_n_tb;
22
23 wire [31:0] Mem;      // used to connect risc_v input of wb_mux to output of DMEM DataR
24
25 wire [31:0] Addr_IMEM;
26 wire [31:0] Addr_DMEM;
27 wire [31:0] Rs2;
28 wire [31:0] Instr;
29 wire        MemRW;
30
31 ///////////////////////////////////////////////////////////////////
32 //////////////////////////////////////////////////////////////////// Clock Generation
33 ///////////////////////////////////////////////////////////////////
34
35 initial
36 begin
37     clk_tb = 0;
38     forever #(CLK_PERIOD / 2) clk_tb = ~clk_tb;
39 end
```

```
41 //////////////////////////////////////////////////////////////////// DUT Instantiations
42 ///////////////////////////////////////////////////////////////////
43
44
45 RISC_V #(DATA_WIDTH(DATA_WIDTH)) DUT (
46     .clk(clk_tb),
47     .rst_n(rst_n_tb),
48     .Instr(Instr),
49     .Mem(Mem),
50     .Addr_IMEM(Addr_IMEM),
51     .Addr_DMEM(Addr_DMEM),
52     .Rs2(Rs2),
53     .MemRW(MemRW)
54 );
55
56 IMEM #(DATA_WIDTH(DATA_WIDTH), .MEM_SIZE(MEM_SIZE)) U_IMEM (
57     .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
58     .Instr(Instr)
59 );
60
61 DMEM #(DATA_WIDTH(DATA_WIDTH), .MEM_SIZE(MEM_SIZE)) U_DMEM (
62     .clk(clk_tb),
63     .MemRW(MemRW),
64     .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.
65     .DataR(Rs2),
66     .DataR(Mem)
67 );
68
69 //////////////////////////////////////////////////////////////////// Initial Block
70 ///////////////////////////////////////////////////////////////////
71
72
73 initial
74 begin
75     // System functions
76     $dumpfile("RISC_V_tb.vcd");
77     $dumpvars;
78
79     reset();
80
81     #(150*CLK_PERIOD)
82
83     $display("Value stored in the register (t0): %d", DUT.U_Data_Path.U0_Register_File.RegFile[5]);
84
85
86     $display("Value stored in the Data Memory (Address 0): %d", U_DMEM.memory[0]);
87
88
89     #(200*CLK_PERIOD)
90     $stop;
91 end
```

3.2 Simulation Results

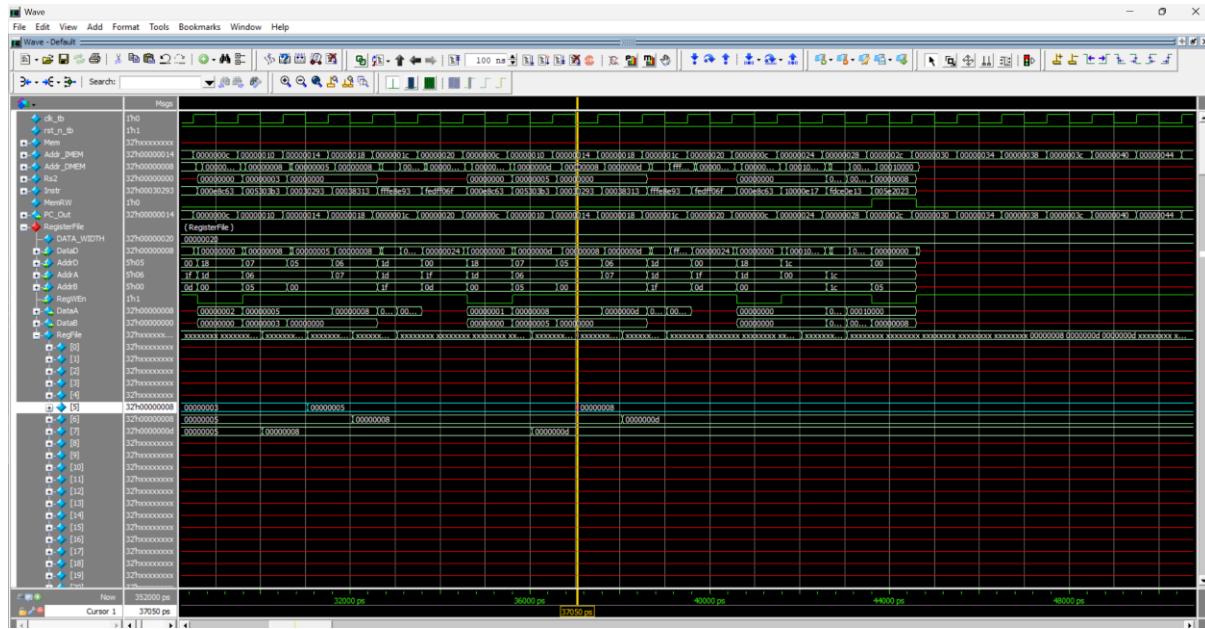


Figure 3-1: 6th Fibonacci number successfully written at register x5.

3.2.1 Testing Rest of the Instructions (store word)

					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	0xFFFFE8E93	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.

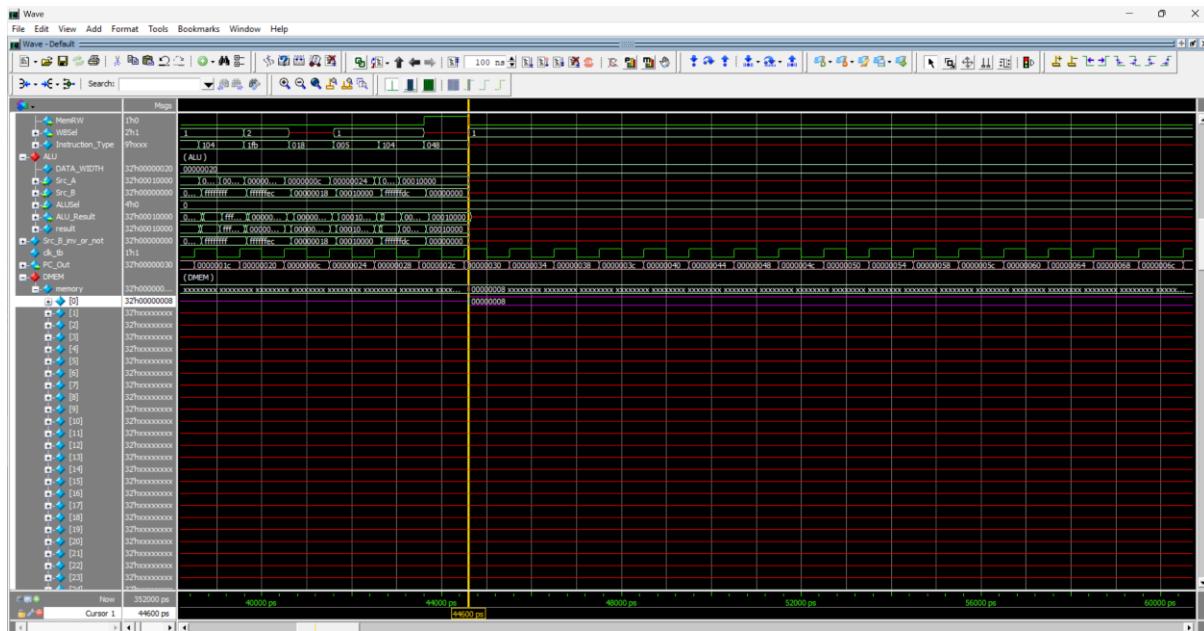


Figure 3-3: 6th Fibonacci number successfully written at memory address 0.

4. Pipelined RISC-V Architecture

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.

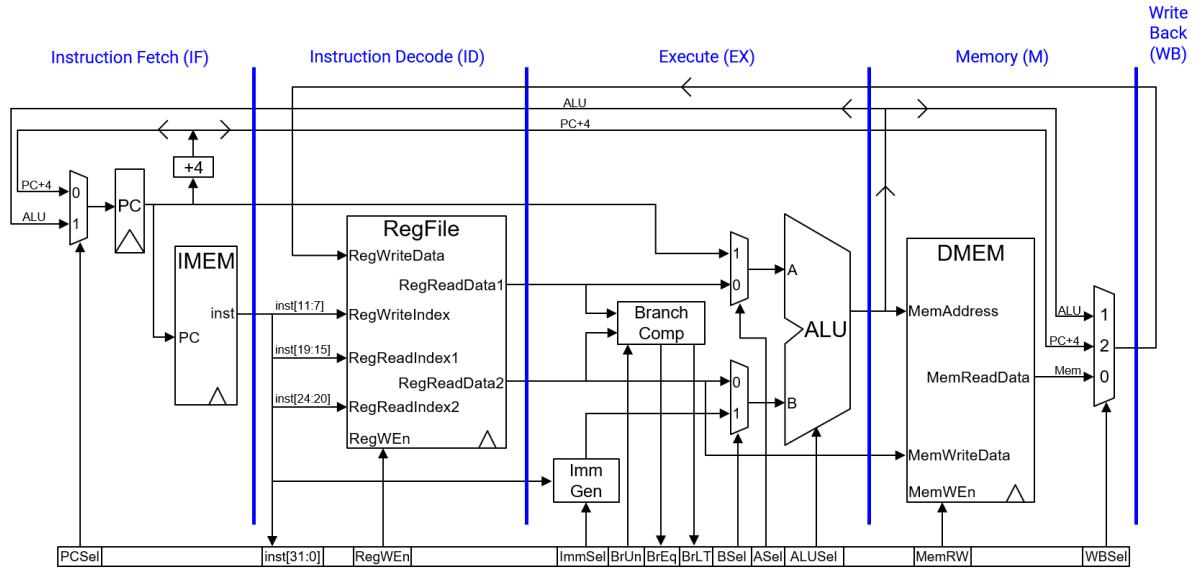


Figure 4-1: Datapath divided into 5 stages. [2]

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

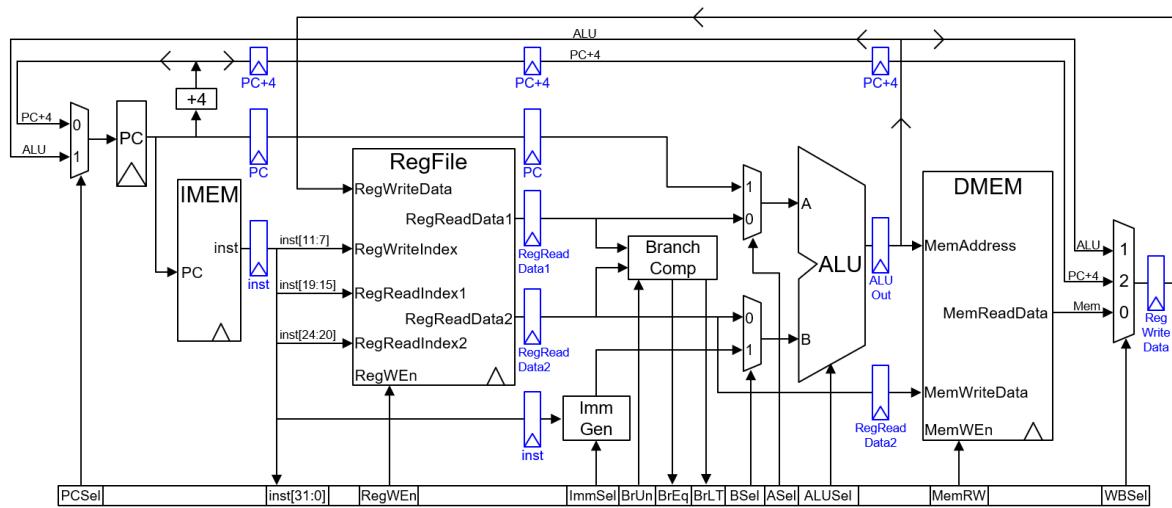


Figure 4-2: Registers added between each stage. [2]

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.

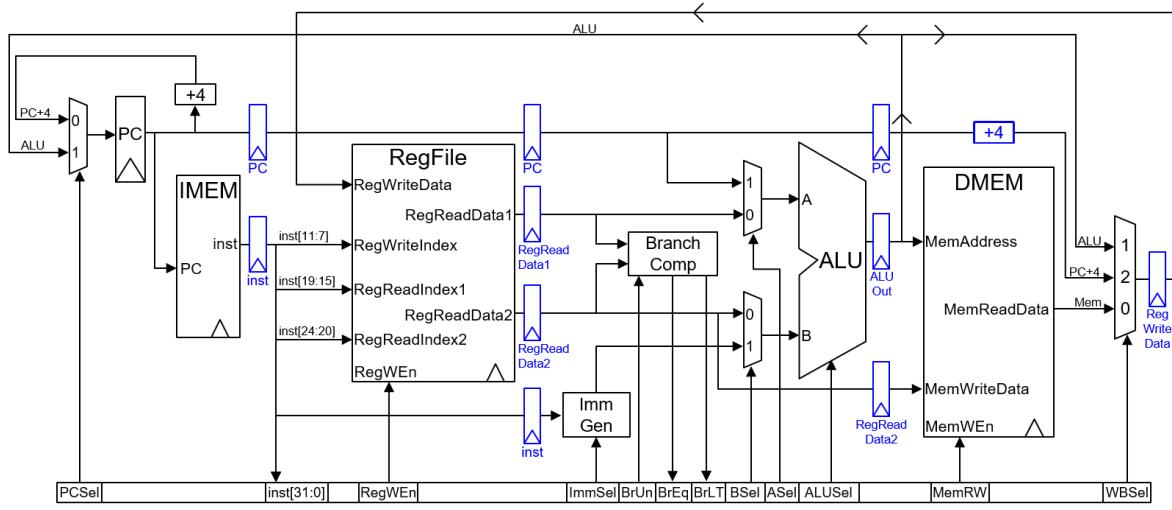


Figure 4-3: Recomputing PC+4 in the memory stage. [2]

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

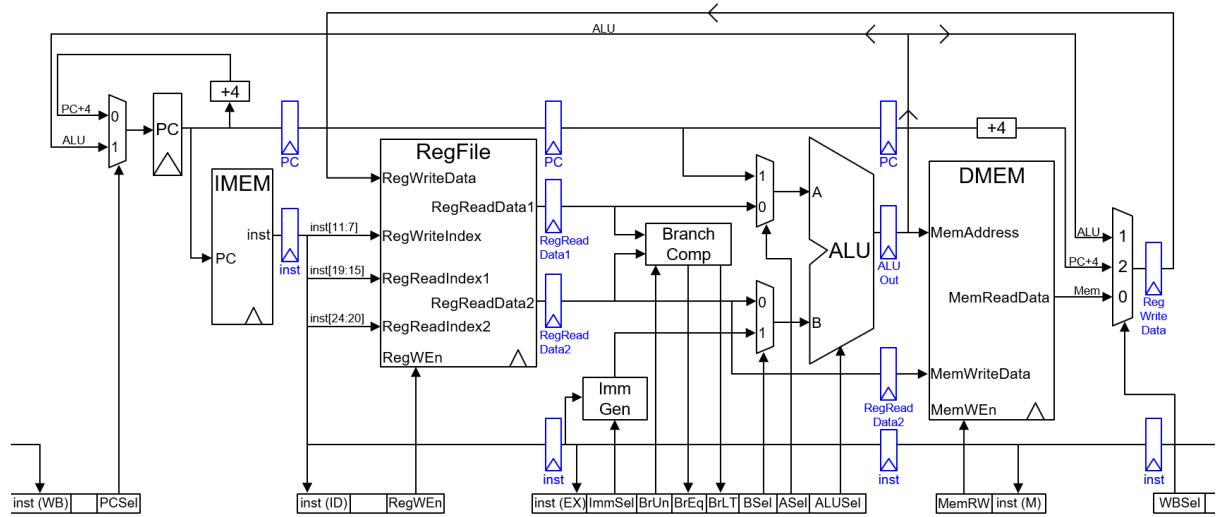


Figure 4-4: Update RegWriteIndex to use the rd field from the WB stage. [2]

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.

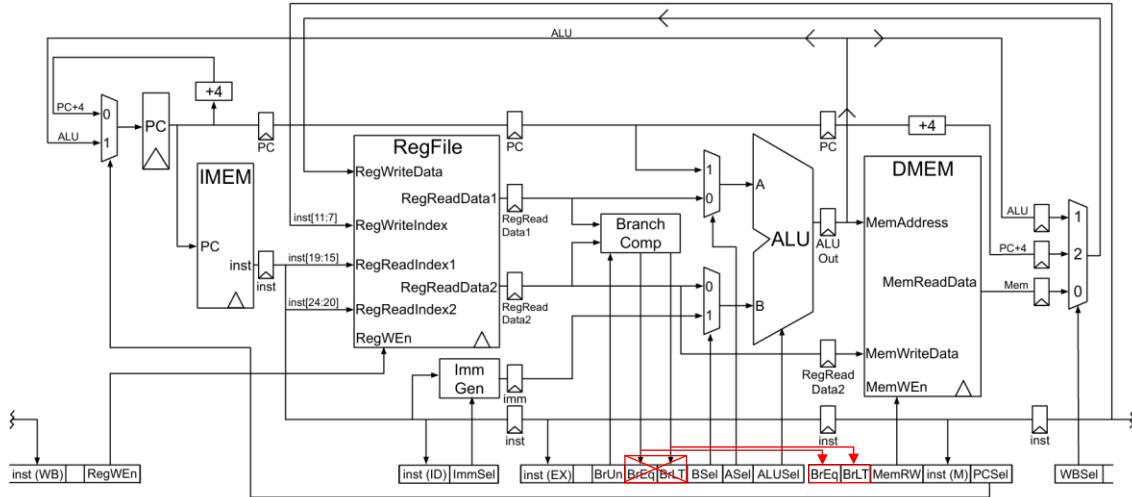


Figure 4-5: Modified pipelined RISC-V architecture. Adapted from [3]

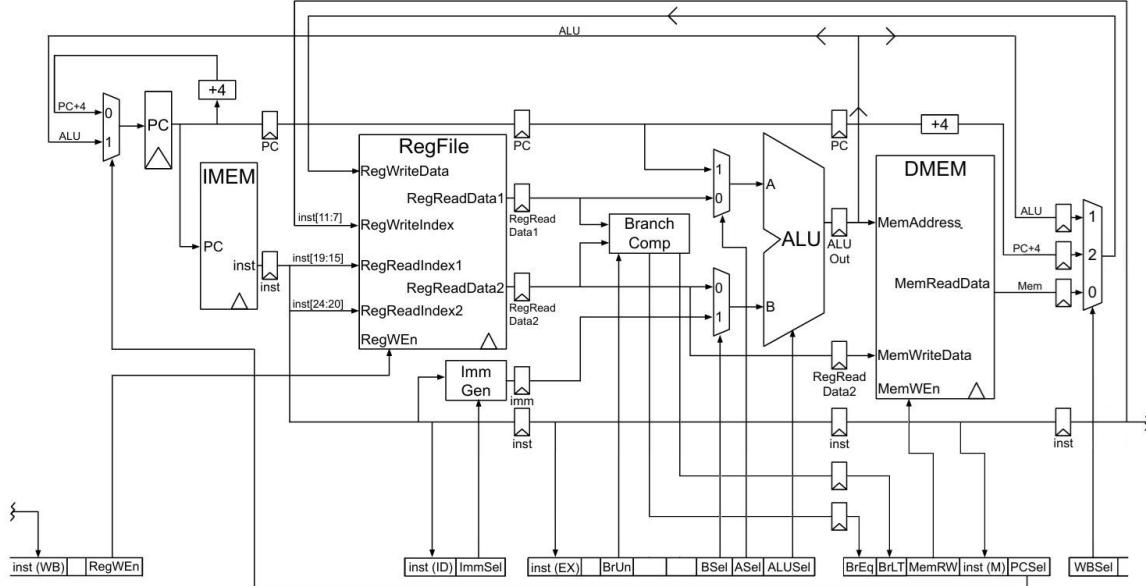


Figure 4-6: Final pipelined RISC-V architecture. Adapted from [3]

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.

31	30	25 24	21 20 19	15 14	12 11	8 7 6	0	
funct7	rs2	rs1	funct3	rd	opcode			R-type
imm[11:0]		rs1	funct3	rd	opcode			I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode			S-type
imm[12:10:5]	rs2	rs1	funct3	imm[4:1 11]	opcode			B-type
	imm[31:12]			rd	opcode			U-type
imm[20 10:1 11]]		imm[19:12]		rd	opcode			J-type

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.

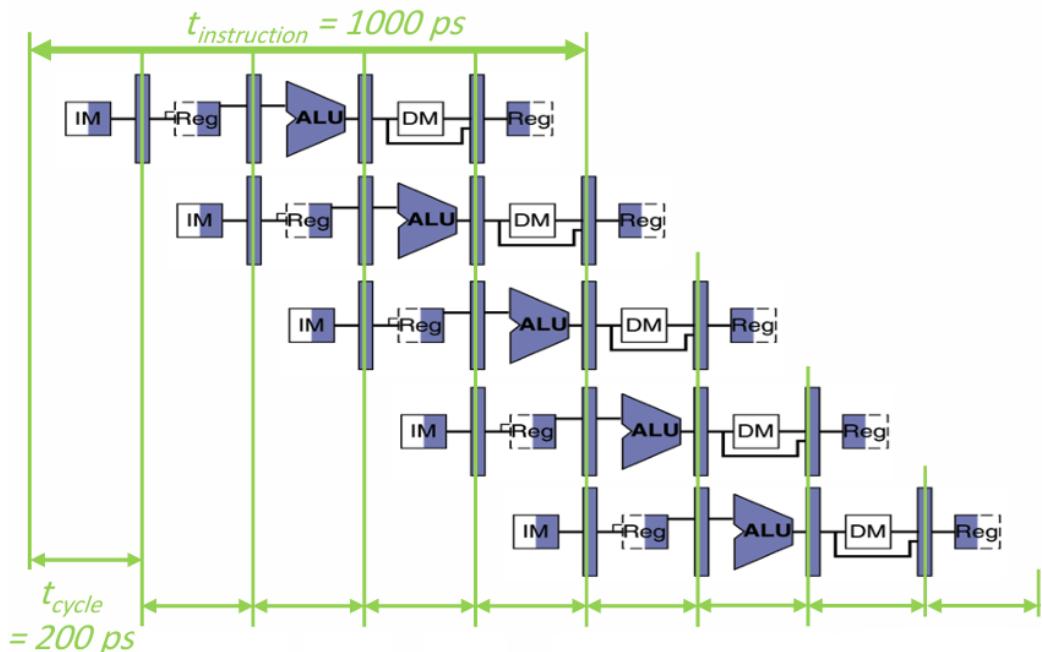


Figure 4-8: Pipeline Stages of the RISC-V. Adapted from [5]

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.

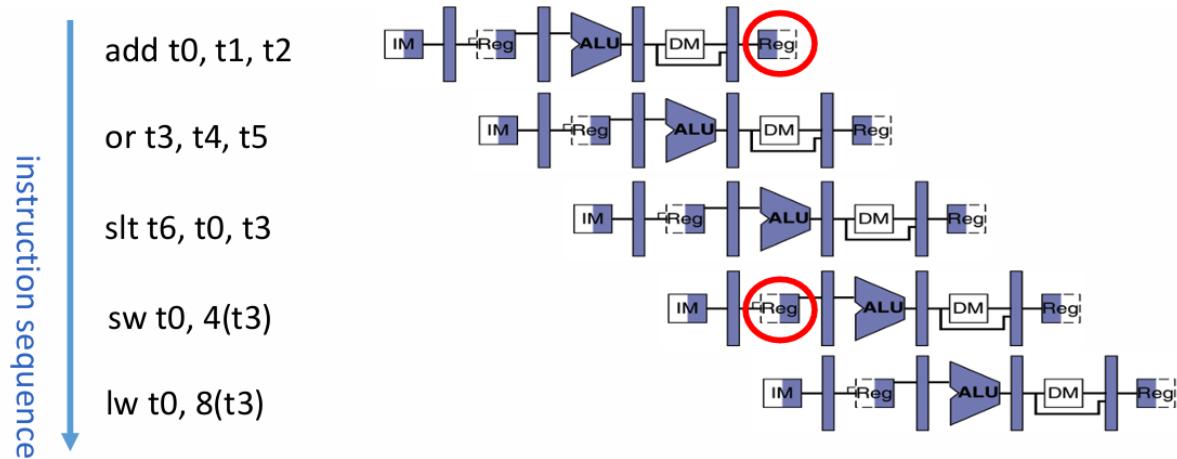


Figure 4-9: Pipelined RISC-V instructions with edge-triggered memory access: write on negative edge, read on positive edge, eliminating structural hazards. [5]

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	0x0000E0E13	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.

```
1 .data
2 fib_result: .word 0      # Reserve space for one word to store the result
3
4 .text
5 main:
6   la t3, fib_result    # Load the address of fib_result into t3 (requires 2 manual stalls between translated instructions)
7   add t0, x0, x0        # t0 = 0 (1st Fibonacci number)
8   addi t1, x0, 1        # t1 = 1 (2nd Fibonacci number)
9   li t4, 6              # Load n = 6 into t4
10  nop                 # Must stall
11  nop                 # Must stall
12
13 fib:
14  beq t4, x0, finish   # If n == 0, exit loop
15  nop                 # Must stall
16  nop                 # Must stall
17  nop                 # Must stall
18  add t2, t1, t0        # t2 = t1 + t0 (next Fibonacci number)
19  addi t4, t4, -1       # Decrement n (rearranged)
20  mv t0, t1              # t0 = t1 (shift Fibonacci sequence)
21  mv t1, t2              # t1 = t2 (shift Fibonacci sequence)
22  j fib                # Repeat the loop
23 finish:
24  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 `instM` 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
<code>add t0 t1 t2</code>	IF	ID	EX	M	WB	
<code>slt t6 t0 t3</code>		IF	ID	EX	M	WB

	1	2	3	4	5	6
<code>add t0 t1 t2</code>	IF	ID	EX	M	WB	
<code>slt t6 t0 t3</code>		IF	ID	EX	M	WB

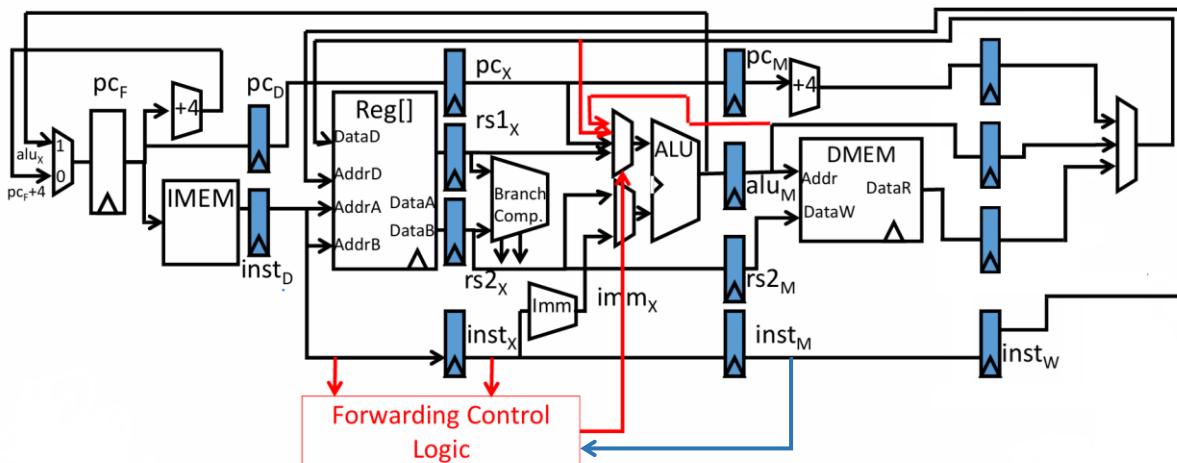


Figure 4-11: Forwarding control logic in hazard unit. Adapted from [5]

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.

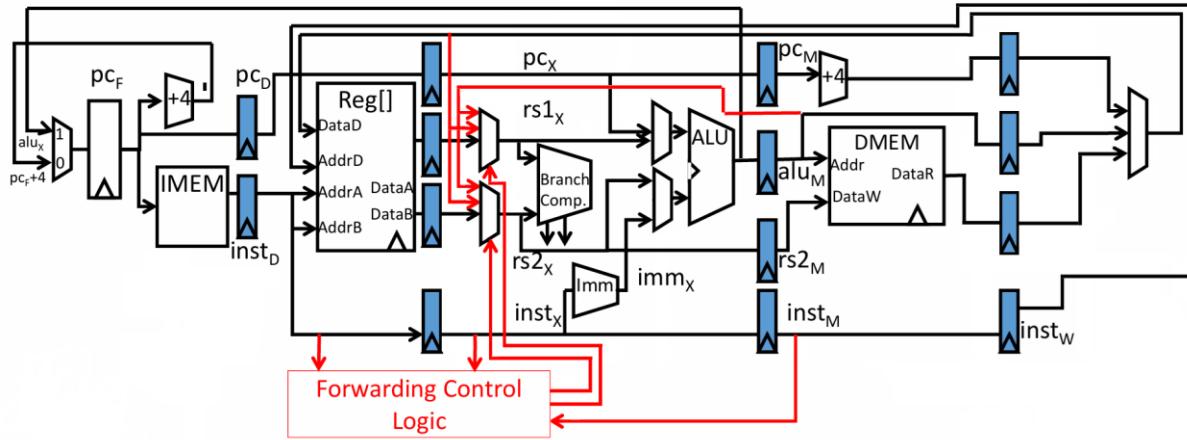


Figure 4-12: Updated forwarding logic ensuring correct data propagation across pipeline stages. Adapted from [5]

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.

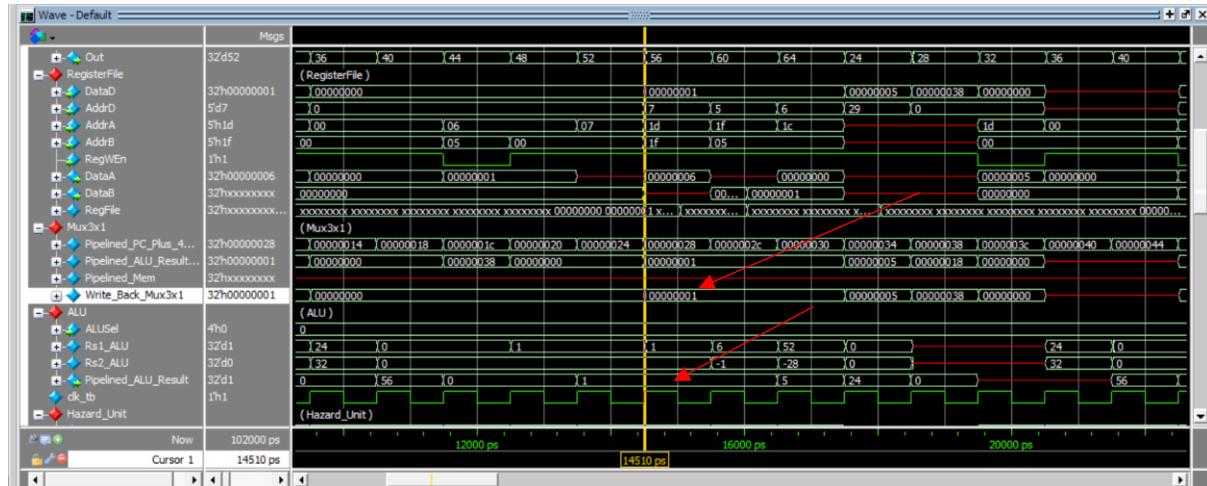


Figure 4-13: ALU result from the previous instruction is available at the write back multiplexer.



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.

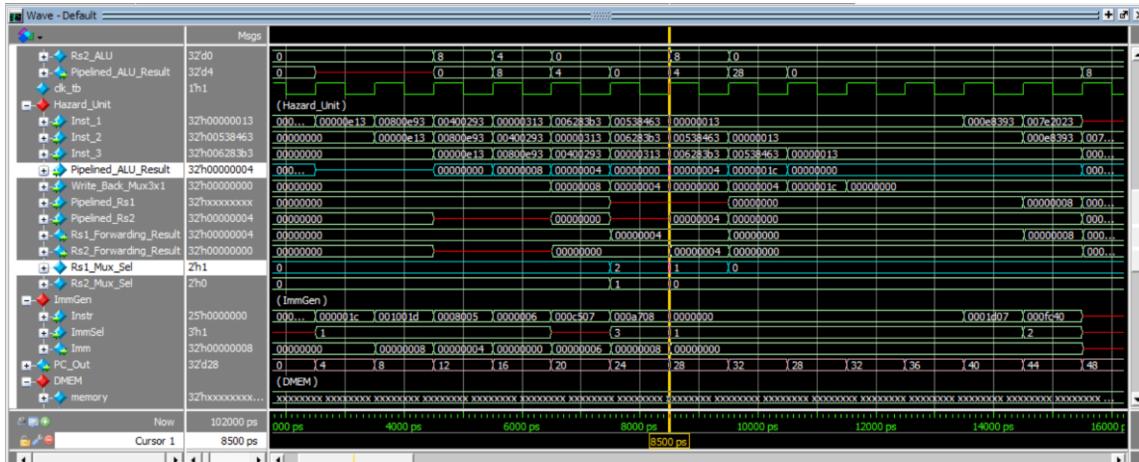


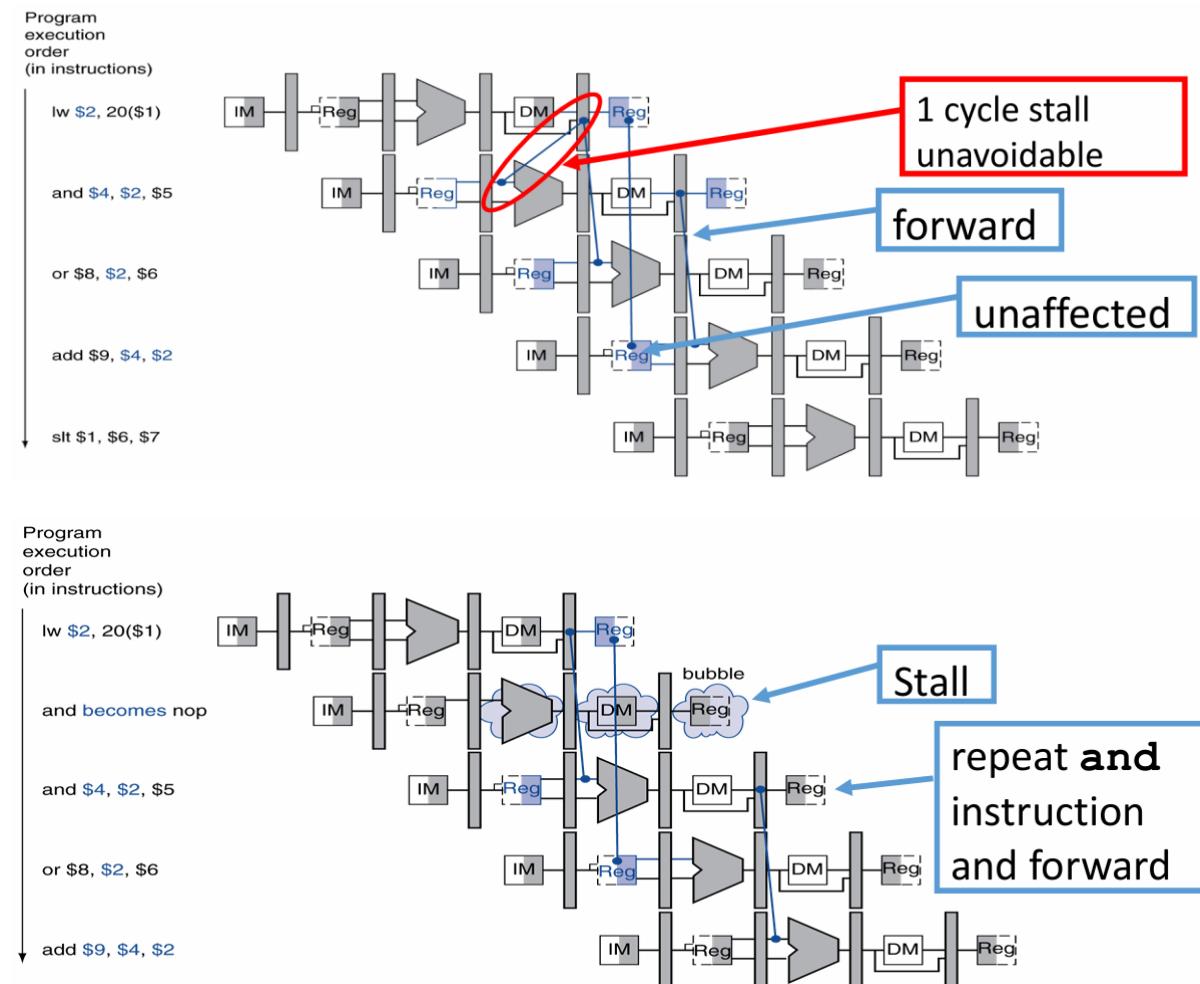
Figure 4-15: ALU result forwarded to the branch comparator for Rs1.



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.

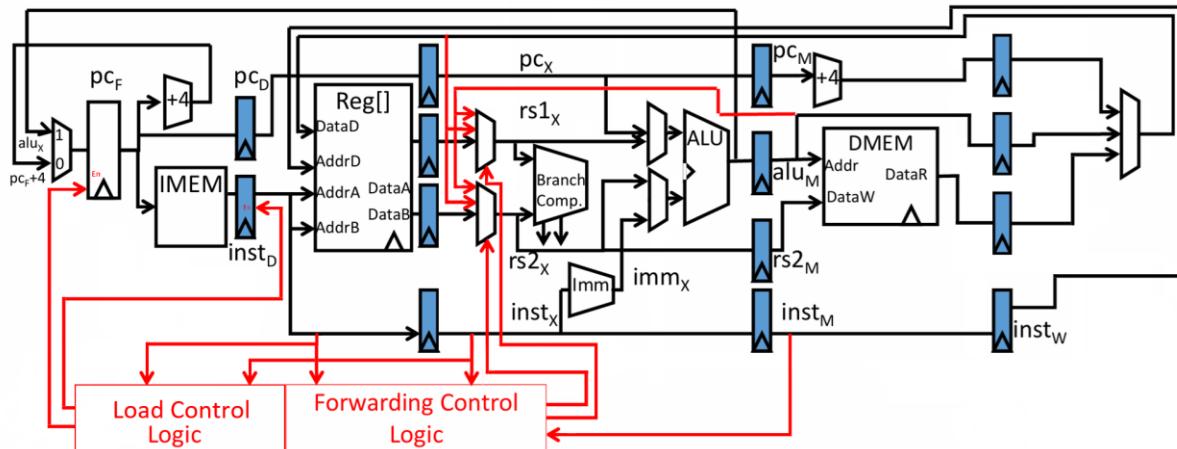


Figure 4-17: Load hazard control: stalling the PC and instruction registers to resolve data dependencies. Adapted from [5]

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

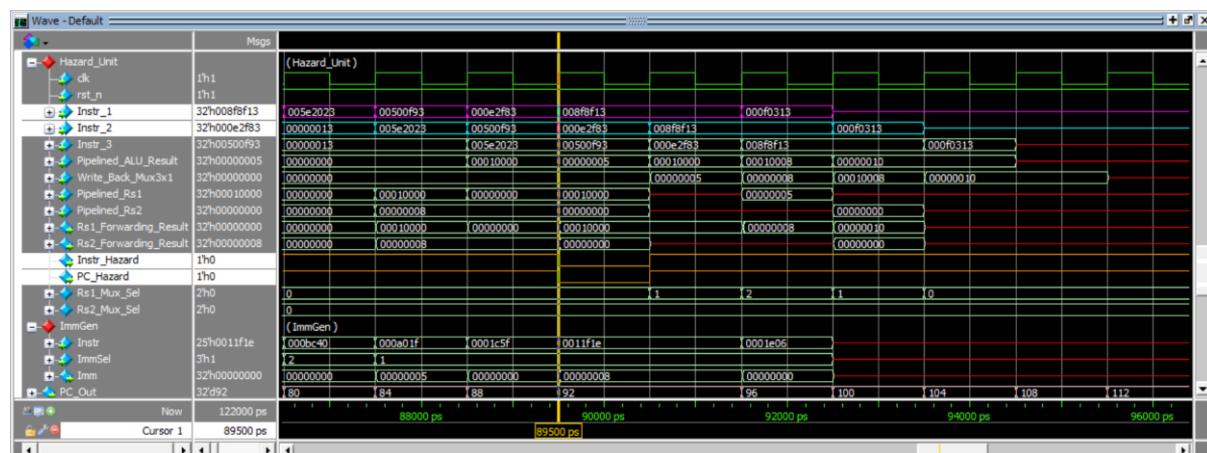


Figure 4-18: Handling load hazards.

4.5 Modified RTL for Pipelined RISC-V

4.5.1 Enabled Register

```
1  module En_Register #(parameter DATA_WIDTH = 32) (
2      input  wire                               clk,
3      input  wire                               rst_n,
4      input  wire                               En,
5      input  wire [DATA_WIDTH-1:0]             In,
6      output reg [DATA_WIDTH-1:0]              Out
7  );
8
9  always@(posedge clk or negedge rst_n)
10 begin
11     if (!rst_n)
12         Out <= 'b0;
13     else if (En)
14         Out <= In;
15 end
16
17 endmodule
```

4.5.2 PC

```
1  `timescale 1ps/1fs
2
3  module PC #(parameter DATA_WIDTH = 32)(
4      input  wire                               clk,
5      input  wire                               rst_n,
6      input  wire                               En,          // Added for Load Hazard
7      input  wire [DATA_WIDTH-1:0]             PC_In,
8      output reg [DATA_WIDTH-1:0]              PC_Out
9  );
10
11 always@(posedge clk or negedge rst_n)
12 begin
13     //#10; // Simple Logic Delay
14     if(!rst_n)
15         PC_Out <= 'b0;
16     else if (En)           // Modified for Hazard
17         PC_Out <= PC_In;
18 end
19
20
21 endmodule
```

4.5.3 Datapath

```

1  `include "Mux2x1.v"
2  `include "PC.v"
3  `include "Add.v"
4  `include "ImmGen.v"
5  `include "RegisterFile.v"
6  `include "BranchComp.v"
7  `include "ALU.v"
8  `include "Mux3x1.v"
9  `include "Register.v"
10 `include "En_Register.v"
11 `include "../Hazard_Unit/Hazard_Unit.v"
12
13 module Data_Path #(parameter DATA_WIDTH = 32) (
14     input  wire          clk,
15     input  wire          rst_n,
16     input  wire          PCSel,
17     input  wire [2:0]     ImmSel,
18     input  wire          RegWEN,
19     input  wire          BrUn,
20     input  wire          BSel,
21     input  wire          ASel,
22     input  wire [3:0]     ALUSel,
23     input  wire          MemRW,
24     input  wire [1:0]     WBSel,
25     input  wire [31:0]    Instr,
26     input  wire [DATA_WIDTH-1:0] Mem,
27     output wire [DATA_WIDTH-1:0] PC_Next,
28     output wire [DATA_WIDTH-1:0] Pipelined_Rs2_2,           // Modified for Pipeline
29     output wire [DATA_WIDTH-1:0] Pipelined_Instr_1,         // ===== Pipelined Instruction ===== //
30     output wire [DATA_WIDTH-1:0] Pipelined_Instr_2,         // ===== Pipelined Instruction ===== //
31     output wire [DATA_WIDTH-1:0] Pipelined_Instr_3,         // ===== Pipelined Instruction ===== //
32     output wire [DATA_WIDTH-1:0] Pipelined_Instr_4,         // ===== Pipelined Instruction ===== //
33     output wire          Pipelined_BrEq,                  // -- To Delay BrEq one clk to be used in Memory Control to generate PCSel correctly -- //
34     output wire          Pipelined_BrLT,                  // -- To Delay and BrLT one clk to be used in Memory Control to generate PCSel correctly -- //
35     output wire [DATA_WIDTH-1:0] Pipelined_ALU_Result,       // Modified for Pipeline
36     output wire          BrEq,
37     output wire          BrLT,
38 );
39

```

```

41 // Internal Wires
42
43 wire [DATA_WIDTH-1:0] PC_Plus_4;
44
45 wire [DATA_WIDTH-1:0] PC_Mux2x1_Out;
46
47 wire [DATA_WIDTH-1:0] Imm;
48
49 wire [DATA_WIDTH-1:0] Write_Back_Mux3x1;
50
51 wire [DATA_WIDTH-1:0] Rs1_ALU;
52 wire [DATA_WIDTH-1:0] Rs2_ALU;
53
54 // Pipeline Wires
55
56 wire [DATA_WIDTH-1:0] PC_Next_Reg1;
57 wire [DATA_WIDTH-1:0] PC_Next_Reg2;
58 wire [DATA_WIDTH-1:0] PC_Next_Reg3;
59 wire [DATA_WIDTH-1:0] Pipelined_PC_Plus_4;
60
61 wire [DATA_WIDTH-1:0] Rs1;
62 wire [DATA_WIDTH-1:0] Rs2;
63 wire [DATA_WIDTH-1:0] Pipelined_Rs1;
64 wire [DATA_WIDTH-1:0] Pipelined_Rs2;
65
66 wire [DATA_WIDTH-1:0] ALU_Result;
67
68 wire [DATA_WIDTH-1:0] Pipelined_PC_Plus_4_Out_Mux;
69 wire [DATA_WIDTH-1:0] Pipelined_ALU_Result_Mux;
70 wire [DATA_WIDTH-1:0] Pipelined_Mem_Mux;
71
72 wire [DATA_WIDTH-1:0] Pipelined_Imm;
73
74
75 // Hazard Unit Forawrd Wires
76
77 wire [DATA_WIDTH-1:0] Rs1_Forwarding_Result;
78 wire [DATA_WIDTH-1:0] Rs2_Forwarding_Result;
79
80 wire Instr_Hazard;
81 wire PC_Hazard;
82
83
84
85
86
87 //////////////////////////////// PC Mux /////////////////////////////////
88 //////////////////////////////// PC Mux /////////////////////////////////
89 //////////////////////////////// PC Mux /////////////////////////////////
90
91 Mux2x1 #(.DATA_WIDTH(DATA_WIDTH)) U0_PC_Mux (
92     .Sel(PCSel),
93     .In0(PC_Plus_4),
94     .In1(Pipelined_ALU_Result),      // Modified for Pipeline
95     .Out(PC_Mux2x1_Out)
96 );
97

```

```

98  //////////////////////////////////////////////////////////////////// PC Register /////////////////////////////////////////////////////////////////////
99  //////////////////////////////////////////////////////////////////// PC Register /////////////////////////////////////////////////////////////////////
100 /////////////////////////////////////////////////////////////////////
101
102 PC #(DATA_WIDTH) U0_PC (
103     .clk(clk),
104     .rst_n(rst_n),
105     .En(PC_Hazard),           // Added for Load Hazard
106     .PC_In(PC_Mux2x1_Out),
107     .PC_Out(PC_Next)
108 );
109
110
111 // ----- Pipelined PC ----- //
112
113 //----- Pipeline_Registered_PC 1 -----//
114 //----- Pipeline_Registered_PC 1 -----//
115 //----- Pipeline_Registered_PC 1 -----//
116
117 Register #(DATA_WIDTH) U1_Pipeline_Registered_PC (
118     .clk(clk),
119     .rst_n(rst_n),
120     .In(PC_Next),
121     .Out(PC_Next_Reg1)
122 );
123
124 //----- Pipeline_Registered_PC 2 -----//
125 //----- Pipeline_Registered_PC 2 -----//
126 //----- Pipeline_Registered_PC 2 -----//
127
128 Register #(DATA_WIDTH) U2_Pipeline_Registered_PC (
129     .clk(clk),
130     .rst_n(rst_n),
131     .In(PC_Next_Reg1),
132     .Out(PC_Next_Reg2)
133 );
134
135 //----- Pipeline_Registered_PC 3 -----//
136 //----- Pipeline_Registered_PC 3 -----//
137 //----- Pipeline_Registered_PC 3 -----//
138
139 Register #(DATA_WIDTH) U3_Pipeline_Registered_PC (
140     .clk(clk),
141     .rst_n(rst_n),
142     .In(PC_Next_Reg2),
143     .Out(PC_Next_Reg3)
144 );
145
146 //----- Adder for Pipeline_Registered_PC -----//
147 //----- Adder for Pipeline_Registered_PC -----//
148 //----- Adder for Pipeline_Registered_PC -----//
149
150 Add #(DATA_WIDTH) U1_Pipeline_Add (
151     .PC(PC_Next_Reg3),
152     .PC_Plus_4(Pipelined_PC_Plus_4)
153 );
154
155 //----- Pipeline_Registered_PC4 -----//
156 //----- Pipeline_Registered_PC4 -----//
157 //----- Pipeline_Registered_PC4 -----//
158
159 Register #(DATA_WIDTH) U1_Pipeline_PC_Plus4 (
160     .clk(clk),
161     .rst_n(rst_n),
162     .In(Pipelined_PC_Plus_4),
163     .Out(Pipelined_PC_Plus_4_Out_Mux)
164 );
165
166
167 //////////////////////////////////////////////////////////////////// Adder for PC ///////////////////////////////////////////////////////////////////
168 //////////////////////////////////////////////////////////////////// Adder for PC ///////////////////////////////////////////////////////////////////
169 //////////////////////////////////////////////////////////////////// Adder for PC ///////////////////////////////////////////////////////////////////
170
171 Add #(DATA_WIDTH) U0_Add (
172     .PC(PC_Next),
173     .PC_Plus_4(PC_Plus_4)
174 );
175
176
177 // ----- Pipelined Instruction ----- //
178
179 //----- Pipeline_Registered_Instruction 1-----//
180 //----- Pipeline_Registered_Instruction 1-----//
181 //----- Pipeline_Registered_Instruction 1-----//
182
183 En_Register #(DATA_WIDTH) U1_Pipeline_Instruction (
184     .clk(clk),
185     .rst_n(rst_n),
186     .En(Inst_Hazard),          // Added for Load Hazard
187     .In(Inst),
188     .Out(Pipelined_Instr_1)
189 );
190
191 //----- Pipeline_Registered_Instruction 2-----//
192 //----- Pipeline_Registered_Instruction 2-----//
193 //----- Pipeline_Registered_Instruction 2-----//
194
195 Register #(DATA_WIDTH) U2_Pipeline_Instruction (
196     .clk(clk),
197     .rst_n(rst_n),
198     .In(Pipelined_Instr_1),
199     .Out(Pipelined_Instr_2)
200 );
201
202 //----- Pipeline_Registered_Instruction 3-----//
203 //----- Pipeline_Registered_Instruction 3-----//
204 //----- Pipeline_Registered_Instruction 3-----//
205
206 Register #(DATA_WIDTH) U3_Pipeline_Instruction (
207     .clk(clk),
208     .rst_n(rst_n),
209     .In(Pipelined_Instr_2),
210     .Out(Pipelined_Instr_3)
211 );

```

```

213 // -- To Delay BrEq and BrLT one clk to be used in Memory Control to generate PCSel correctly -- //
214 Register #(DATA_WIDTH(1)) U1_BrEq (
215     .clk(clk),
216     .rst_n(rst_n),
217     .In(BrEq),
218     .Out(Pipelined_BrEq)
219 );
220
221 Register #(DATA_WIDTH(1)) U1_BrLT (
222     .clk(clk),
223     .rst_n(rst_n),
224     .In(BrLT),
225     .Out(Pipelined_BrLT)
226 );
227
228 //----- Pipeline_Registered_Instruction 4 -----
229 //----- Pipeline_Registered_Instruction 4 -----
230 //----- Pipeline_Registered_Instruction 4 -----
231 //----- Pipeline_Registered_Instruction 4 -----
232 //----- Pipeline_Registered_Instruction 4 -----
233 //----- Pipeline_Registered_Instruction 4 -----
234
235 Register #(DATA_WIDTH(DATA_WIDTH)) U4_Pipeline_Instruction (
236     .clk(clk),
237     .rst_n(rst_n),
238     .In(Pipelined_Instr_3),
239     .Out(Pipelined_Instr_4)
240 );
241
242 // -----
243
244 //***** Register File *****
245 //*****
246 //*****
247 //*****
248 Registerfile #(DATA_WIDTH(DATA_WIDTH)) U0_Register_File (
249     .clk(clk),
250     .Data0(Write_Bus3x1),
251     .AddrA(Pipelined_Instr_4[11:7]),           // Modified for Pipeline
252     .AddrB(Pipelined_Instr_1[19:15]),           // Modified for Pipeline and Hazard
253     .RegWEn(RegWEn),
254     .DataA(Rs1),
255     .DataB(Rs2)
256 );
257
258
259
260 // ----- Pipelined Rs1 and Rs2 -----
261 //----- Pipeline_Registered_Rs1 -----
262 //----- Pipeline_Registered_Rs1 -----
263 //----- Pipeline_Registered_Rs1 -----
264 //----- Pipeline_Registered_Rs1 -----
265 //----- Pipeline_Registered_Rs1 -----
266
267 Register #(DATA_WIDTH(DATA_WIDTH)) U1_Pipeline_Registered_Rs1 (
268     .clk(clk),
269     .rst_n(rst_n),
270     .In(Rs1),
271     .Out(Pipelined_Rs1)
272 );
273
274 //----- Pipeline_Registered_Rs2 -----
275 //----- Pipeline_Registered_Rs2 -----
276 //----- Pipeline_Registered_Rs2 -----
277
278 Register #(DATA_WIDTH(DATA_WIDTH)) U1_Pipeline_Registered_Rs2 (
279     .clk(clk),
280     .rst_n(rst_n),
281     .In(Rs2),
282     .Out(Pipelined_Rs2)
283 );
284
285
286 // ***** Hazard Unit (Forwarding) *****
287
288 //***** Hazard Unit Forwarding *****
289 //*****
290 //*****
291
292 Hazard_Unit #(DATA_WIDTH(DATA_WIDTH)) U0_Hazard_Unit (
293     .clk(clk),
294     .rst_n(rst_n),
295     .Instr_1(Pipelined_Instr_1),
296     .Instr_2(Pipelined_Instr_2),
297     .Instr_3(Pipelined_Instr_3),
298     .Pipelined_ALU_Result(Pipelined_ALU_Result),
299     .Write_Bus3x1(Write_Bus3x1),
300     .Pipelined_Rs1(Pipelined_Rs1),
301     .Pipelined_Rs2(Pipelined_Rs2),
302     .Rs1_Forwarding_Result(Rs1_Forwarding_Result),
303     .Rs2_Forwarding_Result(Rs2_Forwarding_Result),
304     .Instr_Hazard(Instr_Hazard),
305     .PC_Hazard(PC_Hazard)
306 );
307
308
309
310
311 //----- Pipeline_Registered_Rs2_2 -----
312 //----- Pipeline_Registered_Rs2_2 -----
313 //----- Pipeline_Registered_Rs2_2 -----
314
315 Register #(DATA_WIDTH(DATA_WIDTH)) U1_Pipeline_Registered_Rs2_2 (
316     .clk(clk),
317     .rst_n(rst_n),
318     .In(Rs2_Forwarding_Result),           // Modified for Hazard
319     .Out(Pipelined_Rs2_2)
320 );
321
322
323
324
325 //***** Immediate Generator *****
326 //*****
327 //*****
328
329 ImmGen U0_ImmGen (
330     .Instr(Pipelined_Instr_1[31:7]),    // Modified for Pipeline and Hazard
331     .ImmSel(ImmSel),
332     .Imm(Imm)
333 );

```

```

335 // ----- Pipelined Imm ----- //
337 //----- Pipeline_Registered_Imm -----// 
338 //----- Pipeline_Registered_Imm -----// 
339 //----- Pipeline_Registered_Imm -----// 
340 //----- Pipeline_Registered_Imm -----// 
341 //----- Pipeline_Registered_Imm -----// 
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 //***** Branch Comparator ****// 
351 //***** Branch Comparator ****// 
352 //***** Branch Comparator ****// 
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 //***** Rs1 Mux ****// 
363 //***** Rs1 Mux ****// 
364 //***** Rs1 Mux ****// 
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 //***** Rs2 Mux ****// 
374 //***** Rs2 Mux ****// 
375 //***** Rs2 Mux ****// 
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 //***** ALU ****// 
385 //***** ALU ****// 
386 //***** ALU ****// 
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 // ----- Pipelined ALU ----- //
397 //----- Pipeline Registered_ALU 1 -----// 
398 //----- Pipeline Registered_ALU 1 -----// 
399 //----- Pipeline Registered_ALU 1 -----// 
400 //----- Pipeline Registered_ALU 1 -----// 
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 //----- Pipeline_Registered_ALU 2 -----// 
410 //----- Pipeline_Registered_ALU 2 -----// 
411 //----- Pipeline_Registered_ALU 2 -----// 
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 //----- Pipeline_Mem -----// 
421 //----- Pipeline_Mem -----// 
422 //----- Pipeline_Mem -----// 
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 //***** Write Back Mux ****// 
433 //***** Write Back Mux ****// 
434 //***** Write Back Mux ****// 
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 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

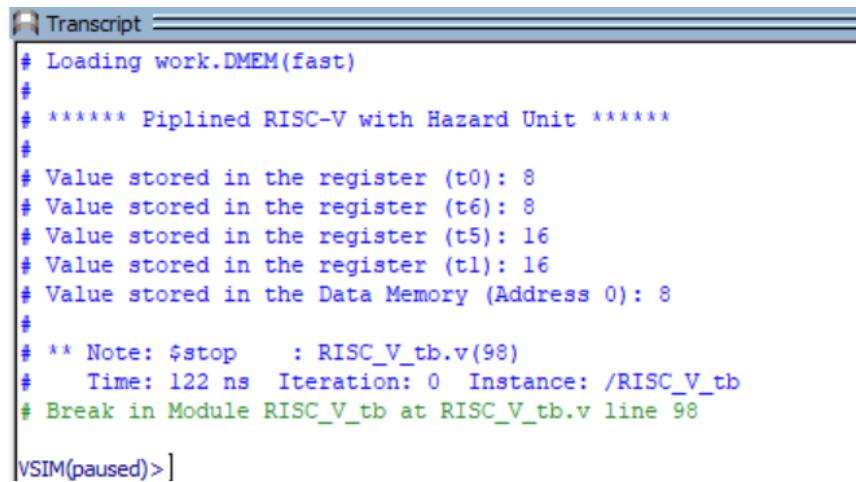
```

1  `include "../Data_Path/Mux3x1.v"
2  `include "../Data_Path/Mux2x1.v"
3
4  module Hazard_Unit #(parameter DATA_WIDTH = 32) (
5      input  wire          clk,
6      input  wire          rst_n,
7      input  wire [DATA_WIDTH-1:0] Instr_1,
8      input  wire [DATA_WIDTH-1:0] Instr_2,
9      input  wire [DATA_WIDTH-1:0] Instr_3,
10     input  wire [DATA_WIDTH-1:0] Pipelined_ALU_Result,
11     input  wire [DATA_WIDTH-1:0] Write_Back_Mux3x1,
12     input  wire [DATA_WIDTH-1:0] Pipelined_Rs1,
13     input  wire [DATA_WIDTH-1:0] Pipelined_Rs2,
14     output wire [DATA_WIDTH-1:0] Rs1_Forwarding_Result,
15     output wire [DATA_WIDTH-1:0] Rs2_Forwarding_Result,
16     output reg           Instr_Hazard,
17     output reg           PC_Hazard
18 );
19
20 // **** Data Hazard ****
21
22 reg [1:0] Rs1_Mux_Sel;
23 reg [1:0] Rs2_Mux_Sel;
24
25
26 // Instr_1 for rs1 or rs2
27 // Instr_2 and Inst_3 for rd
28
29
30
31 // Note: I used a sequential always block here to delay the action to the next clock edge,
32 // ensuring that the multiplexer select signal is ready when the next instruction causing
33 // the hazard reaches the Execute stage.
34
35
36 // Rs1 Forwarding Selection
37 always@(posedge clk or negedge rst_n)
38 begin
39     if (!rst_n)
40         begin
41             Rs1_Mux_Sel <= 2'd0;
42         end
43     else
44         begin
45             if (Instr_1[19:15] == 5'b0 || Instr_2[11:7] == 5'b0 || Instr_3[11:7] == 5'b0)
46                 begin
47                     Rs1_Mux_Sel <= 2'd0; // No forwarding
48                 end
49             else
50                 begin
51                     if (Instr_2[11:7] == Instr_1[19:15])
52                         Rs1_Mux_Sel <= 2'd1; // Take Result from ALU
53                     else if (Instr_3[11:7] == Instr_1[19:15])
54                         Rs1_Mux_Sel <= 2'd2; // Take Result from Write Back
55                     else
56                         Rs1_Mux_Sel <= 2'd0; // No Forwarding
57                 end
58             end
59         end
60     end
61
62 // Rs2 Forwarding Selection
63 always@(posedge clk or negedge rst_n)
64 begin
65     if (!rst_n)
66         begin
67             Rs2_Mux_Sel <= 2'd0;
68         end
69     else
70         begin
71             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
72                 begin
73                     Rs2_Mux_Sel <= 2'd0; // No Forwarding
74                 end
75             else
76                 begin
77                     if (Instr_2[11:7] == Instr_1[24:20])
78                         Rs2_Mux_Sel <= 2'd1; // Take Result from ALU
79                     else if (Instr_3[11:7] == Instr_1[24:20])
80                         Rs2_Mux_Sel <= 2'd2; // Take Result from Write Back
81                     else
82                         Rs2_Mux_Sel <= 2'd0; // No Forwarding
83                 end
84             end
85         end
86     end
87
88 Mux3x1#(1,0)(Pipelined_Rs1), .In1(Pipelined_ALU_Result), .In2(Write_Back_Mux3x1), .Sel(Rs1_Mux_Sel), .Out(Rs1_Forwarding_Result);
89 Mux3x1#(2,0)(Pipelined_Rs2), .In1(Pipelined_ALU_Result), .In2(Write_Back_Mux3x1), .Sel(Rs2_Mux_Sel), .Out(Rs2_Forwarding_Result);
90
91 // **** Load Hazard ****
92
93
94
95 always@(*)
96 begin
97     if (Instr_2[6:0] == 7'b0000011 && (Instr_2[11:7] == Instr_1[19:15] || Instr_2[11:7] == Instr_1[24:20] ))
98         begin
99             Instr_Hazard <= 1'b0; // Stall Instruction
100            PC_Hazard <= 1'b0; // Stall PC
101        end
102    else
103        begin
104            Instr_Hazard <= 1'b1;
105            PC_Hazard <= 1'b1;
106        end
107    end
108
109 endmodule
110
111

```

4.5.6 Simulation Results of Pipelined RISC-V with Hazard Unit

```
1 .data
2 fib_result: .word 0      # Reserve space for one word to store the result
3
4 .text
5 main:
6   la t3, fib_result    # Load the address of fib_result into t3 (requires 2 manual stalls between translated instructions)
7   add t0, x0, x0        # t0 = 0 (1st Fibonacci number)
8   addi t1, x0, 1        # t1 = 1 (2nd Fibonacci number)
9   li t4, 6              # Load n = 6 into t4
10
11 fib:
12   beq t4, x0, finish  # If n == 0, exit loop
13   nop
14   nop
15   nop
16   add t2, t1, t0        # t2 = t1 + t0 (next Fibonacci number)
17   mv t0, t1              # t0 = t1 (shift Fibonacci sequence)
18   mv t1, t2              # t1 = t2 (shift Fibonacci sequence)
19   addi t4, t4, -1        # Decrement n
20   j fib                 # Repeat the loop
21   nop
22   nop
23   nop
24 finish:
25   sw t0, 0(t3)          # Store the final Fibonacci number in fib_result
26   li t6, 5                # t6 = 5
27   lw t6, 0(t3)          # t6 = 8
28   addi t5, t6, 8          # t5 = 16
29   addi t1, t5, 0          # t1 = 16
```



The screenshot shows a simulation transcript window titled "Transcript". The output displays the results of a RISC-V pipeline simulation. It includes the command "# Loading work.DMEM(fast)", followed by a series of "# Value stored in the register (t0): 8", "# Value stored in the register (t6): 8", "# Value stored in the register (t5): 16", "# Value stored in the register (t1): 16", and "# Value stored in the Data Memory (Address 0): 8". Below this, there is a note "# ** Note: \$stop : RISC_V_tb.v(98)" and a break point information "# Time: 122 ns Iteration: 0 Instance: /RISC_V_tb # Break in Module RISC_V_tb at RISC_V_tb.v line 98". The transcript concludes with "[VSIM(paused)]".

```
# Loading work.DMEM(fast)
#
# ***** Pipelined RISC-V with Hazard Unit *****
#
# Value stored in the register (t0): 8
# Value stored in the register (t6): 8
# Value stored in the register (t5): 16
# Value stored in the register (t1): 16
# Value stored in the Data Memory (Address 0): 8
#
# ** Note: $stop : RISC_V_tb.v(98)
# Time: 122 ns Iteration: 0 Instance: /RISC_V_tb
# Break in Module RISC_V_tb at RISC_V_tb.v line 98
[VSIM(paused)]
```

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.