

台科大

PA3

30 頁真的寫不完

周 柏宇

2024/5/20

Part 1	1
Adder.....	1
Description	1
IO	1
Explanation	2
ALU_Control	2
Description	2
IO	2
Explanation	2
ALU	2
Description	2
IO	2
Explanation	3
Control.....	3
Description	3
IO	3
Explanation	4
DM.....	4
Description	4
IO	4
Explanation	4
EX_MEM.....	5
Description	5
IO	5
Explanation	5
GPRMUX.....	6
Description	6
IO	6
Explanation	6
ID_EX.....	6
Description	6
IO	6
Explanation	7
IF_ID	8
Description	8
IO	8
Explanation	8
IM	8

Description	8
IO	8
Explanation	8
MEM_WB	8
Description	8
IO	8
Explanation	9
R_PipelineCPU	9
Description	10
IO	10
Explanation	10
RegMUX	11
Description	11
IO	11
Explanation	11
RF	11
Description	11
IO	11
Explanation	11
Part 2	12
Changes	12
Part 3	12
Changes	12
New modules.....	12
FinalCPU.....	12
Forwarding_Unit.....	14
HD_Unit	15
Stall_MUX	15
TMUX	16
Part 1 Test Program	16
RF	16
Part 2 Test Program	16
DM	16
RF	17
Part 3 Test Program	17
DM	17
RF	17
Simulation Result.....	17
Timing	17
Utilization	17

Power	17
-------------	----

Part 1

Adder

```

1 module Adder(
2     // Outputs
3     output [31:0] OutputAddr,
4     // Inputs
5     input [31:0] InputAddr,
6     input [31:0] Offset
7 );
8
9     assign OutputAddr = InputAddr + Offset;
10
11 endmodule

```

Description

The purpose of the Adder module is to compute the sum of two 32-bit input values, InputAddr and Offset, and provide the result as a 32-bit output value, OutputAddr.

IO

Inputs:

Port	Width	Description
InputAddr	31 bits	Original pc addr.
Offset	31 bits	How many to be added.

Outputs:

Port	Width	Description
OutputAddr	31 bits	New pc addr.

Explanation

line 9

Using assign statement to perform a continuous addition of the inputs in.

ALU_Control

```
1 module ALU_Control(
2   // Outputs
3   output reg [5:0] funct,
4   // Inputs
5   input [1:0] ALU_op,
6   input [5:0] funct_ctrl
7 );
8   parameter ADDU = 6'b001001;
9   parameter SUBU = 6'b001010;
10  parameter SLL = 6'b100001;
11  parameter SLLV = 6'b110101;
12
13  always @(*) begin
14    case(ALU_op)
15      2'b00: funct <= 6'b001001;
16      2'b01: funct <= 6'b001010;
17      2'b10: begin
18        case(funct_ctrl)
19          ADDU: funct <= 6'b001001;
20          SUBU: funct <= 6'b001010;
21          SLL:  funct <= 6'b100001;
22          SLLV: funct <= 6'b110101;
23          default: funct <= 6'b000000;
24        endcase
25      end
26      2'b11: funct <= 6'b101010;
27      default: funct <= 6'b000000;
28    endcase
29  end
30
31 endmodule
```

Description

Generates a function code (**funct**) based on the ALU operation code (**ALU_op**) and a function control code (**funct_ctrl**).

IO

Input:

Port	Width	Description
ALU_op	2 bits	ALU operation code that

		determines the type of operation.
funct_ctrl	6 bits	Function control code used for specific operations when ALU_op is 2'b10 .

Outputs:

Port	Width	Description
funct	6 bits	Function code output used to control the ALU operation.

Explanation

- When **ALU_op** is **2'b00**, **funct** is set to **6'b001001**.
- When **ALU_op** is **2'b01**, **funct** is set to **6'b001010**.
- When **ALU_op** is **2'b11**, **funct** is set to **6'b101010**.
- When **ALU_op** is **2'b10**
 - ADDU: **funct** is set to **6'b001001**.
 - SUBU: **funct** is set to **6'b001010**.
 - SLL: **funct** is set to **6'b100001**.
 - SLLV: **funct** is set to **6'b110101**.
 - For any other **funct_ctrl** value, **funct** defaults to **6'b000000**.

ALU

```
1 module ALU(
2   // Outputs
3   output reg [31:0] ALU_result,
4   // Inputs
5   input [31:0] data1,
6   input [31:0] data2,
7   input [4:0] shamt,
8   input [5:0] funct
9 );
10
11   parameter ADDU = 6'b001001;
12   parameter SUBU = 6'b001010;
13   parameter SLL = 6'b100001;
14   parameter SLLV = 6'b110101;
15   parameter SLTI = 6'b101010;
16
17   wire [31:0] ADDU_result, SUBU_result, SLL_result, SLLV_result, SLTI_result;
18
19   assign ADDU_result = data1 + data2;
20   assign SUBU_result = data1 - data2;
21   assign SLL_result = data1 << shamt;
22   assign SLLV_result = data1 << data2[4:0];
23   assign SLTI_result = data1 < data2;
24
25   always @(*) begin
26     case(funct)
27       ADDU: ALU_result <= ADDU_result;
28       SUBU: ALU_result <= SUBU_result;
29       SLL:  ALU_result <= SLL_result;
30       SLLV: ALU_result <= SLLV_result;
31       SLTI: ALU_result <= SLTI_result;
32       default: ALU_result <= 32'b0;
33     endcase
34   end
35
36 endmodule
```

Description

Performs arithmetic and logical operations based on the function code (**funct**).

IO

Inputs:

Port	Width	Description
data1	32 bits	First operand for the ALU operation.
data2	32 bits	Second operand for the ALU operation or shift amount in SLLV.
shamt	5 bits	Shift amount for the shift left logical (SLL) operation.
funct	6 bits	Function code that

		determines the specific ALU operation to perform.
--	--	---

Outputs:

Port	Width	Description
ALU_result	32 bits	Result of the ALU operation.

Explanation

Line 19~23

Using wire to get all result at the same time

Line 25~33

Put the value to ALU_result designated by funct.

Control

```

1  `define R_TYPE 6'b000000
2  `define SUBIU 6'b001101
3  `define SW 6'b010000
4  `define LW 6'b010001
5  `define SLTI 6'b101010
6
7  module Control(
8      // Outputs
9      output reg RegDst,
10     output reg RegWrite,
11     output reg [1:0] ALU_op,
12     output reg ALU_src,
13     output reg Mem_w,
14     output reg Mem_r,
15     output reg Mem_to_Reg,
16     // Inputs
17     input [5:0] opcode
18 );
19
20     always @(*) begin
21         case(opcode)
22             `R_TYPE: begin
23                 RegDst <= 1'b1;
24                 RegWrite <= 1'b1;
25                 ALU_op <= 2'b10;
26                 ALU_src <= 1'b0;
27                 Mem_w <= 1'b0;
28                 Mem_r <= 1'b0;
29                 Mem_to_Reg <= 1'b0;
30             end
31             `SUBIU: begin
32                 RegDst <= 1'b0;
33                 RegWrite <= 1'b1;
34                 ALU_op <= 2'b01;
35                 ALU_src <= 1'b1;
36                 Mem_w <= 1'b0;
37                 Mem_r <= 1'b0;
38                 Mem_to_Reg <= 1'b0;
39             end
40             `SW: begin
41                 RegDst <= 1'b0;
42                 RegWrite <= 1'b0;
43                 ALU_op <= 2'b00;
44                 ALU_src <= 1'b1;
45                 Mem_w <= 1'b1;
46                 Mem_r <= 1'b0;
47                 Mem_to_Reg <= 1'b0;
48             end
49             `LW: begin
50                 RegDst <= 1'b0;
51                 RegWrite <= 1'b1;
52                 ALU_op <= 2'b00;
53                 ALU_src <= 1'b1;
54                 Mem_w <= 1'b0;
55                 Mem_r <= 1'b1;
56                 Mem_to_Reg <= 1'b1;
57             end

```

```

58             `SLTI: begin
59                 RegDst <= 1'b0;
60                 RegWrite <= 1'b1;
61                 ALU_op <= 2'b11;
62                 ALU_src <= 1'b1;
63                 Mem_w <= 1'b0;
64                 Mem_r <= 1'b0;
65                 Mem_to_Reg <= 1'b0;
66             end
67         default: begin
68             RegDst <= 1'b0;
69             RegWrite <= 1'b0;
70             ALU_op <= 2'b00;
71             ALU_src <= 1'b0;
72             Mem_w <= 1'b0;
73             Mem_r <= 1'b0;
74             Mem_to_Reg <= 1'b0;
75         end
76     endcase
77 end
78 endmodule

```

Description

Generates control signals based on the opcode of an instruction.

IO

Inputs:

Port	Width	Description
opcode	6 bits	Opcode of the instruction to be decoded.

Outputs:

Port	Width	Description
RegDst	1 bit	Determines the destination register for the write operation.
RegWrite	1 bit	Enables

		writing to the register file.
ALU_op	2 bits	Selects the ALU operation to be performed.
ALU_src	1 bit	Selects the second operand for the ALU.
Mem_w	1 bit	Enables writing to memory.
Mem_r	1 bit	Enables reading from memory.
Mem_to_Reg	1 bit	Selects the data source for writing to the register file.

Explanation

uses an **always @(*)** block to generate control signals based on the input **opcode**.

DM

```

29 define DATA_MEM_SIZE 128 // Bytes
30
31 /*
32  * Declaration of Data Memory for this project.
33  * CAUTION: DONT MODIFY THE NAME.
34  */
35 module DM(
36     // Outputs
37     output [31:0] Mem_r_data,
38     // Inputs
39     input [31:0] Mem_addr,
40     input [31:0] Mem_w_data,
41     input Mem_w,
42     input Mem_r,
43     input clk
44 );
45
46 /*
47  * Declaration of data memory.
48  * CAUTION: DONT MODIFY THE NAME AND SIZE.
49  */
50 reg [7:0]DataMem[0:DATA_MEM_SIZE - 1];
51
52 // write data to memory
53 always @(negedge clk) begin
54     if(Mem_w) begin
55         DataMem[Mem_addr[6:0]] <= Mem_w_data[31:24];
56         DataMem[Mem_addr[6:0] + 1] <= Mem_w_data[23:16];
57         DataMem[Mem_addr[6:0] + 2] <= Mem_w_data[15:8];
58         DataMem[Mem_addr[6:0] + 3] <= Mem_w_data[7:0];
59     end
60 end
61
62 // read data from memory
63 assign Mem_r_data = {DataMem[Mem_addr[6:0]], DataMem[Mem_addr[6:0] + 1]
64 , DataMem[Mem_addr[6:0] + 2], DataMem[Mem_addr[6:0] + 3]};
65
66 endmodule

```

Description

Simulate a data memory component that allows reading from and writing to memory locations.
Not used in R type.

IO

Inputs:

Port	Width	Description
Mem_addr	32 bits	Address from which to read or write data.
Mem_w_data	32 bits	Data to be written to the memory.
Mem_w	1 bit	Write enable signal; when high, data is written to memory.

Mem_r	1 bit	Read enable signal. Not used in this implementation
clk	1 bit	Clock signal. Not used in this implementation.

Outputs:

Port	Width	Description
Mem_r_data	32 bits	Data read from the memory.

Explanation

Using **always @(*)** block ensures that writing occurs whenever **Mem_w** is high.
Using **assign** statement continuously updates **Mem_r_data** based on the current memory contents at the specified address.

EX_MEM

```
1 module EX_MEM (
2     // Outputs
3     output reg [31:0] ALU_result_out,
4     output reg [31:0] Rt_data_out,
5     output reg [4:0] RdAddr_out,
6     output reg MemW_out,
7     output reg MemR_out,
8     output reg Mem2Reg_out,
9     output reg RegWrite_out,
10    // Inputs
11    input [31:0] ALU_result,
12    input [31:0] Rt_data,
13    input [4:0] RdAddr,
14    input MemW,
15    input MemR,
16    input Mem2Reg,
17    input RegWrite,
18    input clk
19 );
20 always @(negedge clk) begin
21     ALU_result_out <= ALU_result;
22     Rt_data_out <= Rt_data;
23     RdAddr_out <= RdAddr;
24     MemW_out <= MemW;
25     MemR_out <= MemR;
26     Mem2Reg_out <= Mem2Reg;
27     RegWrite_out <= RegWrite;
28 end
29 endmodule
30
31
```

Description

Stores the outputs from the execution stage and passes them to the memory stage in a pipelined processor.

IO

Inputs:

Port	Width	Description
ALU_result	32 bits	Result of the ALU operation from the execution stage.

Rt_data	32 bits	Data from register Rt from the execution stage.
RdAddr	5 bits	Register destination address from the execution stage.
MemW	1 bit	Signal indicating memory write operation.
MemR	1 bit	Signal indicating memory read operation.
Mem2Reg	1 bit	Signal indicating whether memory data should be written back to a register.
RegWrite	1 bit	Signal enabling register write operation.
clk	1 bit	Clock signal.

Outputs:

Port	Width	Description
ALU_result_out	32 bits	Result of the ALU operation.
Rt_data_out	32	Data from

	bits	register Rt .
RdAddr_out	5 bits	Register destination address for write-back.
MemW_out	1 bit	Signal indicating memory write operation.
MemR_out	1 bit	Signal indicating memory read operation.
Mem2Reg_out	1 bit	Signal indicating whether memory data is written back to a register.
RegWrite_out	1 bit	Signal enabling register write operation.

Explanation

Use registers to store the results and control signals from the EX stage on the falling edge of the clock.

GPRMUX

```
1 module GPRMUX(  
2     // Outputs  
3     output [31:0] data,  
4     // Inputs  
5     input [31:0] src1,  
6     input [31:0] src2,  
7     input ctrl  
8 );  
9  
10    assign data = ctrl ? src2 : src1;  
11  
12 endmodule
```

Description

2-to-1 multiplexer that selects one of two 32-bit input data signals based on a control signal.

IO

Inputs:

Port	Width	Description
src1	32 bits	First input data.
src2	32 bits	Second input data.
ctrl	1 bit	Control signal to select between src1 and src2 .

Outputs:

Port	Width	Description
data	32 bits	Selected output data based on control signal.

Explanation

- When **ctrl** is **0**, the output **data** is equal to **src1**.

- When **ctrl** is **1**, the output **data** is equal to **src2**.

ID_EX

```
1 module ID_EX (  
2     output reg RegDst_out,  
3     output reg RegWrite_out,  
4     output reg [1:0] ALU_op_out,  
5     output reg ALU_src_out,  
6     output reg Mem_w_out,  
7     output reg Mem_r_out,  
8     output reg Mem_to_Reg_out,  
9  
10    output reg [31:0] rs_out,  
11    output reg [31:0] rt_out,  
12    output reg [4:0] rt_addr_out,  
13    output reg [4:0] rd_addr_out,  
14    output reg [31:0] imm_out,  
15  
16    input RegDst,  
17    input RegWrite,  
18    input [1:0] ALU_op,  
19    input ALU_src,  
20    input Mem_w,  
21    input Mem_r,  
22    input Mem_to_Reg,  
23  
24    input [31:0] rs,  
25    input [31:0] rt,  
26    input [4:0] rt_addr,  
27    input [4:0] rd_addr,  
28    input [31:0] imm,  
29  
30    input clk  
31 );  
32    always @(negedge clk) begin  
33        rs_out <= rs;  
34        rt_out <= rt;  
35        rt_addr_out <= rt_addr;  
36        rd_addr_out <= rd_addr;  
37        imm_out <= imm;  
38  
39        RegDst_out <= RegDst;  
40        RegWrite_out <= RegWrite;  
41        ALU_op_out <= ALU_op;  
42        ALU_src_out <= ALU_src;  
43        Mem_w_out <= Mem_w;  
44        Mem_r_out <= Mem_r;  
45        Mem_to_Reg_out <= Mem_to_Reg;  
46    end  
47  
48 endmodule
```

Description

Stores intermediate values and control signals between the Instruction Decode (ID) stage and the Execution (EX) stage.

IO

Inputs:

Port	Width	Description
RegDst	1 bit	Control signal selecting the destination
RegWrite	1 bit	Control signal enabling the register write operation.
ALU_op	2 bits	Control signals for ALU operation type.
ALU_src	1 bit	Control signal selecting the ALU source operand.
Mem_w	1 bit	Control signal enabling memory write operation.
Mem_r	1 bit	Control signal enabling memory read

		operation.
Mem_to_Reg	1 bit	Control signal for selecting data source to write back to the register.
rs	32 bits	Data from source register rs .
rt	32 bits	Data from source register rt .
rt_addr	5 bits	Address of the rt register.
rd_addr	5 bits	Address of the rd register.
imm	32 bits	Immediate value.
clk	1 bit	Clock signal.

Outputs:

Port	Width	Description
RegDst_out	1 bit	Output control signal selecting the destination.
RegWrite_out	1 bit	Output control signal enabling

		the register write operation.
ALU_op_out	2 bits	Output control signals for ALU operation type.
ALU_src_out	1 bit	Output control signal selecting the ALU source operand.
Mem_w_out	1 bit	Output control signal enabling memory write operation.
Mem_r_out	1 bit	Output control signal enabling memory read operation.
Mem_to_Reg_out	1 bit	Output control signal for selecting data source to

		write back to the register.
rs_out	32 bits	Output data from source register rs .
rt_out	32 bits	Output data from source register rt .
rt_addr_out	5 bits	Output address of the rt register.
rd_addr_out	5 bits	Output address of the rd register.
imm_out	32 bits	Output immediate value.

Explanation

On the falling edge of the clock, the module transfers the input data and control signals to their respective output ports.

IF_ID

```
1 module IF_ID (
2     // Outputs
3     output reg [31:0] instr_out,
4     // Inputs
5     input [31:0] instr,
6     input clk
7 );
8 reg init = 0;
9
10 always @(negedge clk) begin
11     if(init == 0) begin
12         instr_out <= 32'bz;
13         init <= 1;
14     end
15     else begin
16         instr_out <= instr;
17     end
18 end
19
20 endmodule
```

Description

Captures and holds the instruction fetched from the instruction memory in the Instruction Fetch (IF) stage.

IO

Inputs:

Port	Width	Description
instr	32 bits	Input instruction from the instruction memory.
clk	1 bit	Clock signal.

Outputs:

Port	Width	Description
instr_out	32 bits	Output instruction to be sent to the

	ID stage.
--	-----------

Explanation

Captures the **instr** input on the falling edge of the clock and updates the **instr_out** output with this value. Using init reg to manually deal with **WRONG** input for the specific case in this PA.

IM

```
20 `define INSTR_MEM_SIZE 128 // Bytes
21 /*
22  * Declaration of Instruction Memory for this project.
23  * CAUTION: DON'T MODIFY THE NAME.
24  */
25 module IM(
26     // Outputs
27     output [31:0] Instruction,
28     // Inputs
29     input [31:0] InputAddr
30 );
31
32 /*
33  * Declaration of instruction memory.
34  * CAUTION: DON'T MODIFY THE NAME AND SIZE.
35  */
36 reg [7:0] InstrMem[0: INSTR_MEM_SIZE - 1];
37
38 // fetch instruction from memory with big-endian
39 assign Instruction = {InstrMem[InputAddr[6:0]], InstrMem[InputAddr[6:0] + 1],
40                     InstrMem[InputAddr[6:0] + 2], InstrMem[InputAddr[6:0] + 3]};
41
42 endmodule
```

Description

Fetches 32-bit instructions from memory based on a given input address.

IO

Inputs:

Port	Width	Description
InputAddr	32 bits	Input address used to fetch the instruction from memory.

Outputs:

Port	Width	Description
Instruction	32 bits	Output instruction fetched from memory.

Explanation

Concatenates four bytes from the memory array (big endian) to form a 32-bit instruction.

MEM_WB

```
1 module MEM_WB (
2     // Outputs
3     output reg [31:0] ALUResult_out,
4     output reg [31:0] Mem_r_data_out,
5     output reg [4:0] RdAddr_out,
6     output reg Mem2Reg_out,
7     output reg RegWrite_out,
8     // Inputs
9     input [31:0] ALUResult,
10    input [31:0] Mem_r_data,
11    input [4:0] RdAddr,
12    input Mem2Reg,
13    input RegWrite,
14    input clk
15 );
16
17 always @(negedge clk) begin
18     ALUResult_out <= ALUResult;
19     Mem_r_data_out <= Mem_r_data;
20     RdAddr_out <= RdAddr;
21     Mem2Reg_out <= Mem2Reg;
22     RegWrite_out <= RegWrite;
23 end
24
25 endmodule
```

Description

Hold the data and control signals between the Memory (MEM) stage and the Write-Back (WB) stage.

IO

Inputs:

Port	Width	Description
------	-------	-------------

	h	n
ALUResult	32 bits	Input result from the ALU.
Mem_r_data	32 bits	Input data read from memory.
RdAddr	5 bits	Input address of the destination register.
Mem2Reg	1 bit	Input control signal for memory to register write.
RegWrite	1 bit	Input control signal enabling the register write.
clk	1 bit	Clock signal.

Outputs:

Port	Width	Description
ALUResult_out	32 bits	Output result from the ALU to be used in the WB stage.
Mem_r_data_out	32 bits	Output data read from

		memory to be used in the WB stage.
RdAddr_out	5 bits	Output address of the destination register.
Mem2Reg_out	1 bit	Output control signal for memory to register write.
RegWrite_out	1 bit	Output control signal enabling the register write.

Explanation

Holds the input values and control signals on the falling edge of the clock (**negedge clk**).

R_PipelineCPU

```

29 module R_PipelineCPU(
30     // Outputs
31     output wire [31:0] Output_Addr,
32     // Inputs
33     input wire [31:0] Input_Addr,
34     input wire clk
35 );
36
37 // 1st stage
38 wire [31:0] Instr;
39
40 // 1st stage
41 wire [31:0] Instr_out;
42 // connections to 1st stage
43 wire [31:0] Rs_data, Rn_data;
44 wire RegDst, RegWrite, ALU_src, MemR, Mem2Reg;
45 wire [31:0] ALU_op;
46 wire [31:0] Imm;
47
48 // 1st stage
49 // 1st stage outputs
50 wire [31:0] Rs_data_out, Rn_data_out;
51 wire [4:0] RADDR_IDEX_out, RADDR_IDEX_out;
52 wire RegDst_out, RegWrite_IDEX_out, ALU_src_out, Mem_IDEX_out, Mem2Reg_IDEX_out;
53 wire [1:0] ALU_op_out;
54 wire [31:0] Imm_out;
55 // connections to 2nd stage
56 wire [31:0] ALU_src2;
57 wire [5:0] funct;
58 wire [31:0] ALU_result;
59 wire [4:0] RADDR_EX;
60
61 // 2nd stage
62 // EXMEM's outputs
63 wire MemR_EXMEM_out, MemR_EXMEM_out, Mem2Reg_EXMEM_out, RegWrite_EXMEM_out;
64 wire [31:0] ALUResult_EXMEM_out;
65 wire [31:0] Rn_data_EXMEM_out;
66 wire [4:0] RADDR_EXMEM_out;
67 // connections to 3rd stage
68 wire [31:0] Mem_r_data;
69
70 // 3rd stage
71 // MEMWB's outputs
72 wire Mem2Reg_MEMWB_out, RegWrite_MEMWB_out;
73 wire [31:0] ALUResult_MEMWB_out;
74 wire [31:0] Mem_r_data_MEMWB_out;
75 wire [4:0] RADDR_MEMWB_out;
76 // connections to 4th stage
77 wire [31:0] Rd_data;

```

```
78
79 // sign extend imm
80 assign imm = { {16{instr_out[15]}}, instr_out[15:0] };
81
82 // IF stage
83 Adder NextAddr(
84 // Outputs
85 .OutputAddr(Output_Addr),
86 // Inputs
87 .InputAddr(Input_Addr),
88 .Offset(32'h4)
89 );
90
91 IM Instr_Memory(
92 // Outputs
93 .Instruction(instr),
94 // Inputs
95 .InputAddr(Input_Addr)
96 );
97
98 // IF2ID stage
99 IF_ID IF2ID(
100 // Outputs
101 .instr_out(instr_out),
102 // Inputs
103 .instr(instr),
104 .clk(clk)
105 );
106
107 // ID(WB) stage
108 RF Register_File(
109 // Outputs
110 .RsData(Rs_data),
111 .RtData(Rt_data),
112 // Inputs
113 .RsAddr(instr_out[25:21]),
114 .RtAddr(instr_out[20:16]),
115 .RdAddr(RdAddr_MEM2WB_out),
116 .RdData(Rd_data),
117 .RegWrite(RegWrite_MEM2WB_out),
118 .clk(clk)
119 );
120
121 Control Control_Unit(
122 // Outputs
123 .RegDst(RegDst),
124 .RegWrite(RegWrite),
125 .ALU_op(ALU_op),
126 .ALU_src(ALU_src),
127 .Mem_w(MemW),
128 .Mem_r(MemR),
129 .Mem_to_Reg(Mem2Reg),
130 // Inputs
131 .opcode(instr_out[31:26])
132 );
133
134 // ID2EX stage
135 ID_EX ID2EX(
136 // Outputs
137 .RegDst_out(RegDst_out),
138 .RegWrite_out(RegWrite_ID2EX_out),
139 .ALU_op_out(ALU_op_out),
140 .ALU_src_out(ALU_src_out),
141 .Mem_w_out(MemW_ID2EX_out),
142 .Mem_r_out(MemR_ID2EX_out),
143 .Mem_to_Reg_out(Mem2Reg_ID2EX_out),
144 .rs_out(Rs_data_out),
145 .rt_out(Rt_data_ID2EX_out),
146 .rt_addr_out(RtAddr_ID2EX_out),
147 .rd_addr_out(RdAddr_ID2EX_out),
148 .imm_out(imm_out),
149 // Inputs
150 .RegDst(RegDst),
151 .RegWrite(RegWrite),
152 .ALU_op(ALU_op),
153 .ALU_src(ALU_src),
154 .Mem_w(MemW),
155 .Mem_r(MemR),
156 .Mem_to_Reg(Mem2Reg),
157 .rs(Rs_data),
158 .rt(Rt_data),
159 .rt_addr(instr_out[20:16]),
160 .rd_addr(instr_out[15:11]),
161 .imm(imm),
162 .clk(clk)
163 );
164
165 // EX stage
166 RegMUX Reg_MUX1(
167 // Outputs
168 .RegDst(RdAddr_EX),
169 // Inputs
170 .Rt(RtAddr_ID2EX_out),
171 .Rd(RdAddr_ID2EX_out),
172 .RegDst_ctrl(RegDst_out)
173 );
```

```
174
175 GPRMUX ALU_MUX(
176 // Outputs
177 .data(ALU_src2),
178 // Inputs
179 .src1(Rt_data_ID2EX_out),
180 .src2(imm_out),
181 .ctrl(ALU_src_out)
182 );
183
184 ALU_Control ALU_Control_Unit(
185 // Outputs
186 .funct(funct),
187 // Inputs
188 .ALU_op(ALU_op_out),
189 .funct_ctrl(imm_out[5:0])
190 );
191
192 ALU ALUInstance(
193 // Outputs
194 .ALU_result(ALU_result),
195 // Inputs
196 .data1(Rs_data_out),
197 .data2(ALU_src2),
198 .shamt(imm_out[10:6]),
199 .funct(funct)
200 );
201
202 // EX2MEM stage
203 EX_MEM EX2MEM(
204 // Outputs
205 .ALU_result_out(ALUResult_EX2MEM_out),
206 .Rt_data_out(Rt_data_EX2MEM_out),
207 .RdAddr_out(RdAddr_EX2MEM_out),
208 .MemW_out(MemW_EX2MEM_out),
209 .MemR_out(MemR_EX2MEM_out),
210 .Mem2Reg_out(Mem2Reg_EX2MEM_out),
211 .RegWrite_out(RegWrite_EX2MEM_out),
212 // Inputs
213 .ALU_result(ALU_result),
214 .Rt_data(Rt_data_ID2EX_out),
215 .RdAddr(RdAddr_EX),
216 .MemW(MemW_ID2EX_out),
217 .MemR(MemR_ID2EX_out),
218 .Mem2Reg(Mem2Reg_ID2EX_out),
219 .RegWrite(RegWrite_ID2EX_out),
220 .clk(clk)
221 );
222
223 // MEM stage
224 DM Data_Memory(
225 // Outputs
226 .Mem_r_data(Mem_r_data),
227 // Inputs
228 .Mem_addr(ALUResult_EX2MEM_out),
229 .Mem_w_data(Rt_data_EX2MEM_out),
230 .Mem_w(MemW_EX2MEM_out),
231 .Mem_r(MemR_EX2MEM_out),
232 .clk(clk)
233 );
234
235 // MEM2WB stage
236 MEM_WB Mem_WB(
237 // Outputs
238 .ALUResult_out(ALUResult_MEM2WB_out),
239 .Mem_r_data_out(Mem_r_data_MEM2WB_out),
240 .RdAddr_out(RdAddr_MEM2WB_out),
241 .Mem2Reg_out(Mem2Reg_MEM2WB_out),
242 .RegWrite_out(RegWrite_MEM2WB_out),
243 // Inputs
244 .ALUResult(ALUResult_EX2MEM_out),
245 .Mem_r_data(Mem_r_data),
246 .RdAddr(RdAddr_EX2MEM_out),
247 .Mem2Reg(Mem2Reg_EX2MEM_out),
248 .RegWrite(RegWrite_EX2MEM_out),
249 .clk(clk)
250 );
251
252 // WB stage
253 GPRMUX Mem_MUX1(
254 // Outputs
255 .data(Rd_data),
256 // Inputs
257 .src1(ALUResult_MEM2WB_out),
258 .src2(Mem_r_data_MEM2WB_out),
259 .ctrl(Mem2Reg_MEM2WB_out)
260 );
261
262 endmodule
263
```

Description

A pipelined CPU architecture with five pipeline stages: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory (MEM), and Write-Back (WB)

IO

Inputs:

Port	Width	Description
Input_Addr	32	PC
clk	1	Clock signal

Outputs:

Port	Width	Description
Output_Addr	32	PC + 4

Explanation

Line 36~77

Wires that connecting the components. They are sorted by their corresponding stage.

Line 78~

Components’ instances. They are mostly sorted by their corresponding stage.

Each pipeline stage has its corresponding registers (IF_ID, ID_EX, EX_MEM, MEM_WB) to store intermediate results and control signals between pipeline stages.

RegMUX

```
1 module RegMUX(  
2     // Outputs  
3     output [4:0] RegDst,  
4     // Inputs  
5     input [4:0] Rt,  
6     input [4:0] Rd,  
7     input RegDst_ctrl  
8 );  
9  
10    assign RegDst = (RegDst_ctrl) ? Rd : Rt;  
11  
12 endmodule  
13
```

Description

For selecting the destination register address.

IO

Inputs:

Port	Width	Description
Rt	5 bits	Input register address representing the source register.
Rd	5 bits	Input register address representing the destination register.
RegDst_ctrl	1 bit	Input control signal for selecting the destination register.

Outputs:

Port	Width	Description
RegDst	5 bits	Output selected register address.

Explanation

Using RegDst_ctrl to selects either the source register address (**Rt**) or the destination register address (**Rd**) as the output.

RF

```
29 `define REG_MEM_SIZE    32 // Words  
30  
31 /*  
32  * Declaration of Register File for this project.  
33  * CAUTION: DONT MODIFY THE NAME.  
34  */  
35 module RF(  
36     // Outputs  
37     output [31:0] RsData,  
38     output [31:0] RtData,  
39     // Inputs  
40     input [4:0] RsAddr,  
41     input [4:0] RtAddr,  
42     input [4:0] RdAddr,  
43     input [31:0] RdData,  
44     input RegWrite,  
45     input clk  
46 );  
47  
48 /*  
49  * Declaration of inner register.  
50  * CAUTION: DONT MODIFY THE NAME AND SIZE.  
51  */  
52 reg [31:0] R[0:`REG_MEM_SIZE - 1];  
53  
54 assign RsData = R[RsAddr];  
55 assign RtData = R[RtAddr];  
56  
57 always @(*) begin  
58     if(RegWrite && !clk)  
59         R[RdAddr] <= RdData;  
60 end  
61  
62 endmodule  
63
```

Description

Register file.

IO

Inputs:

Port	Width	Description
RsAddr	5 bits	Input address

		for reading data from the register file (RsData).
RtAddr	5 bits	Input address for reading data from the register file (RtData).
RdAddr	5 bits	Input address for writing data to the register file.
RdData	32 bits	Input data to be written to the register specified by RdAddr .
RegWrite	1 bit	Input control signal for enabling register write.
clk	1 bit	Clock signal.

Outputs:

Port	Width	Description
RsData	32 bits	Output data from the register specified by RsAddr .
RtData	32 bits	Output data from the register specified by RtAddr .

Explanation

- Constantly read data
- Write only when **RegWrite** is asserted and **clk** is deasserted.

Part 2

Changes

It's the same as Part 1. Except
"R_PipelineCPU" is renamed to
"I_PipelineCPU."

Part 3

Changes

1. Rename "GPRMUX" to "DMUX"
which stands for double sources
mux.
2. Rename RegMUX's input
addresses as addr1 and addr2
respectively for more accurate
discription.
3. "ID_EX" add **rs_addr** as input
and **rs_addr_out** as output for
detecting data hazard.

New modules

FinalCPU

```
21 module FinalCPU(  
22     // Outputs  
23     output PC_Write,   
24     output wire [31:0] Output_Addr,  
25     // Inputs  
26     input wire [31:0] Input_Addr,  
27     input wire      clk  
28 );  
29 // IF stage  
30 wire [31:0] Instr;  
31  
32 // ID stage  
33 // IF2ID's outputs  
34 wire [31:0] Instr_out;  
35 // connections in ID stage  
36 wire [31:0] Rs_data, Rt_data;  
37 wire RegDst, RegWrite, ALU_src, MemW, MemR, Mem2Reg;  
38 wire [1:0] ALU_op;  
39 wire RegDst_C, RegWrite_C, ALU_src_C, MemW_C, MemR_C, Mem2Reg_C;  
40 wire [31:0] Imm;  
41 wire [4:0] RstAddr_ID, RstAddr_ID_C, RdAddr_ID;  
42 wire [4:0] RdAddr_ID_C, RdAddr_ID_C;  
43 wire stall_ctrl;  
44  
45 // EX stage  
46 // EX2EX's outputs  
47 wire [31:0] Rs_data_out, Rt_data_IDEX_out;  
48 wire [4:0] RstAddr_IDEX_out, RstAddr_IDEX_out;  
49 wire RegDst_out, RegWrite_IDEX_out, ALU_src_out, MemW_IDEX_out, MemR_IDEX_out, Mem2Reg_IDEX_out;  
50 wire [1:0] ALU_op_out;  
51 wire [31:0] Imm_out;  
52 // connections in EX stage  
53 wire [31:0] ALU_src2;  
54 wire [5:0] funct;  
55 wire [1:0] forwarding_A_ctrl, forwarding_R_ctrl;  
56 wire [31:0] forwarding_A_data, forwarding_R_data;  
57 wire [31:0] ALU_result;  
58 wire [4:0] RdAddr_EX;  
59  
60 // MEM stage  
61 // EX2MEM's outputs  
62 wire MemW_EXMEM_out, MemR_EXMEM_out, Mem2Reg_EXMEM_out, RegWrite_EXMEM_out;  
63 wire [31:0] ALUResult_EXMEM_out;  
64 wire [31:0] Rt_data_EXMEM_out;  
65 wire [4:0] RdAddr_EXMEM_out;  
66 // connections in MEM stage  
67 wire [31:0] MemR_data;  
68  
69 // WB stage  
70 // MEM2WB's outputs  
71 wire Mem2Reg_MEMWB_out, RegWrite_MEMWB_out;  
72 wire [31:0] ALUResult_MEMWB_out;  
73 wire [31:0] MemR_data_MEMWB_out;  
74 wire [4:0] RdAddr_MEMWB_out;  
75 // connections in WB stage  
76 wire [31:0] WB_data;  
77
```

```
87 // IF stage  
88 Addr NextAddr(  
89     // Outputs  
90     .OutputAddr(Output_Addr),  
91     // Inputs  
92     .InputAddr(Input_Addr),  
93     .Offset(32'h4)  
94 );  
95  
96 IM Instr_Memory(  
97     // Outputs  
98     .Instruction(instr),  
99     // Inputs  
100    .InputAddr(Input_Addr)  
101 );  
102  
103 // IF2ID stage  
104 IF_ID IF2ID(  
105     // Outputs  
106     .instr_out(instr_out),  
107     // Inputs  
108     .instr(stall_ctrl ? instr_out : instr),  
109     .clk(clk)  
110 );  
111  
112 // ID(WB) stage  
113 assign imm = { 16{instr_out[15]}}, instr_out[15:0] };  
114 assign RsAddr_ID = instr_out[25:21];  
115 assign RtAddr_ID_C = instr_out[20:16];  
116 assign RdAddr_ID_C = instr_out[15:11];  
117 assign PC_Write = ~stall_ctrl;  
118  
119 RF Register_File(  
120     // Outputs  
121     .RsData(Rs_data),  
122     .RtData(Rt_data),  
123     // Inputs  
124     .RsAddr(RsAddr_ID),  
125     .RtAddr(RtAddr_ID_C),  
126     .RdAddr(RdAddr_MEM2WB_out),  
127     .RdData(WB_data),  
128     .RegWrite(RegWrite_MEM2WB_out),  
129     .clk(clk)  
130 );
```

```
132 HD_Unit HD_Unit(  
133     // Outputs  
134     .ctrl(stall_ctrl),  
135     // Inputs  
136     .Mem_Read(MemR_ID2EX_out),  
137     .RtAddr_EX(RtAddr_ID2EX_out),  
138     .RsAddr_ID(RsAddr_ID),  
139     .RtAddr_ID(RtAddr_ID_C)  
140 );  
141  
142 Control Control_Unit(  
143     // Outputs  
144     .RegDst(RegDst_C),  
145     .RegWrite(RegWrite_C),  
146     .ALU_op(ALU_op_C),  
147     .ALU_src(ALU_src_C),  
148     .Mem_w(MemW_C),  
149     .Mem_r(MemR_C),  
150     .Mem_to_Reg(Mem2Reg_C),  
151     // Inputs  
152     .opcode(instr_out[31:26])  
153 );  
154  
155 Stall_MUX Stall_MUX_Control(  
156     // Outputs  
157     .RegDst_out(RegDst),  
158     .RegWrite_out(RegWrite),  
159     .ALU_op_out(ALU_op),  
160     .ALU_src_out(ALU_src),  
161     .Mem_w_out(MemW),  
162     .Mem_r_out(MemR),  
163     .Mem_to_Reg_out(Mem2Reg),  
164     // Inputs  
165     .RegDst(RegDst_C),  
166     .RegWrite(RegWrite_C),  
167     .ALU_op(ALU_op_C),  
168     .ALU_src(ALU_src_C),  
169     .Mem_w(MemW_C),  
170     .Mem_r(MemR_C),  
171     .Mem_to_Reg(Mem2Reg_C),  
172     .stall_ctrl(stall_ctrl)  
173 );  
174  
175 RegMUX Stall_MUX_Rt(  
176     // Outputs  
177     .RegDst(RtAddr_ID),  
178     // Inputs  
179     .addr1(RtAddr_ID_C),  
180     .addr2(5'b00000),  
181     .RegDst_ctrl(stall_ctrl)  
182 );  
183  
184 RegMUX Stall_MUX_Rd(  
185     // Outputs  
186     .RegDst(RdAddr_ID),  
187     // Inputs  
188     .addr1(RdAddr_ID_C),  
189     .addr2(5'b00000),  
190     .RegDst_ctrl(stall_ctrl)  
191 );  
192
```



```
193 // ID2EX stage
194 ID_EX ID2EX(
195     // Outputs
196     .RegDst_out(RegDst_out),
197     .RegWrite_out(RegWrite_ID2EX_out),
198     .ALU_op_out(ALU_op_out),
199     .ALU_src_out(ALU_src_out),
200     .Mem_w_out(MemW_ID2EX_out),
201     .Mem_r_out(MemR_ID2EX_out),
202     .Mem_to_Reg_out(Mem2Reg_ID2EX_out),
203     .rs_out(Rs_data_out),
204     .rt_out(Rt_data_ID2EX_out),
205     .rs_addr_out(RsAddr_ID2EX_out),
206     .rt_addr_out(RtAddr_ID2EX_out),
207     .rd_addr_out(RdAddr_ID2EX_out),
208     .imm_out(imm_out),
209     // Inputs
210     .RegDst(RegDst),
211     .RegWrite(RegWrite),
212     .ALU_op(ALU_op),
213     .ALU_src(ALU_src),
214     .Mem_w(MemW),
215     .Mem_r(MemR),
216     .Mem_to_Reg(Mem2Reg),
217     .rs(Rs_data),
218     .rt(Rt_data),
219     .rs_addr(RsAddr_ID),
220     .rt_addr(RtAddr_ID),
221     .rd_addr(RdAddr_ID),
222     .imm(imm),
223     .clk(clk)
224 );
225
226 // EX stage
227 RegMUX Reg_MUX1(
228     // Outputs
229     .RegDst(RdAddr_EX),
230     // Inputs
231     .addr1(RtAddr_ID2EX_out),
232     .addr2(RdAddr_ID2EX_out),
233     .RegDst_ctrl(RegDst_out)
234 );
235
236 Forwarding_Unit Forwarding_Unit(
237     // Outputs
238     .Forwarding_A_ctrl(Forwarding_A_ctrl),
239     .Forwarding_B_ctrl(Forwarding_B_ctrl),
240     // Inputs
241     .RsAddr_ID2EX(RsAddr_ID2EX_out),
242     .RtAddr_ID2EX(RtAddr_ID2EX_out),
243     .RdAddr_EX2MEM(RdAddr_EX2MEM_out),
244     .RdAddr_MEM2WB(RdAddr_MEM2WB_out),
245     .RegWrite_EX2MEM(RegWrite_EX2MEM_out),
246     .RegWrite_MEM2WB(RegWrite_MEM2WB_out)
247 );
248
249 TMUX Forwarding_A(
250     // Outputs
251     .data(Forwarding_A_data),
252     // Inputs
253     .src1(Rs_data_out),
254     .src2(WB_data),
255     .src3(ALUResult_EX2MEM_out),
256     .ctrl(Forwarding_A_ctrl)
257 );
258
259 TMUX Forwarding_B(
260     // Outputs
261     .data(Forwarding_B_data),
262     // Inputs
263     .src1(Rt_data_ID2EX_out),
264     .src2(WB_data),
265     .src3(ALUResult_EX2MEM_out),
266     .ctrl(Forwarding_B_ctrl)
267 );
```

```
269 DMUX IMM_MUX(
270     // Outputs
271     .data(ALU_src2),
272     // Inputs
273     .src1(Forwarding_B_data),
274     .src2(imm_out),
275     .ctrl(ALU_src_out)
276 );
277
278 ALU_Control ALU_Control_Unit(
279     // Outputs
280     .funct(funct),
281     // Inputs
282     .ALU_op(ALU_op_out),
283     .funct_ctrl(imm_out[5:0])
284 );
285
286 ALU ALUInstance(
287     // Outputs
288     .ALU_result(ALU_result),
289     // Inputs
290     .data1(Forwarding_A_data),
291     .data2(ALU_src2),
292     .shamt(imm_out[10:6]),
293     .funct(funct)
294 );
295
296 // EX2MEM stage
297 EX_MEM EX2MEM(
298     // Outputs
299     .ALU_result_out(ALUResult_EX2MEM_out),
300     .Rt_data_out(Rt_data_EX2MEM_out),
301     .RdAddr_out(RdAddr_EX2MEM_out),
302     .MemW_out(MemW_EX2MEM_out),
303     .MemR_out(MemR_EX2MEM_out),
304     .Mem2Reg_out(Mem2Reg_EX2MEM_out),
305     .RegWrite_out(RegWrite_EX2MEM_out),
306     // Inputs
307     .ALU_result(ALU_result),
308     .Rt_data(Forwarding_B_data),
309     .RdAddr(RdAddr_EX),
310     .MemW(MemW_ID2EX_out),
311     .MemR(MemR_ID2EX_out),
312     .Mem2Reg(Mem2Reg_ID2EX_out),
313     .RegWrite(RegWrite_ID2EX_out),
314     .clk(clk)
315 );
316
317 // MEM stage
318 DM Data_Memory(
319     // Outputs
320     .Mem_r_data(Mem_r_data),
321     // Inputs
322     .Mem_addr(ALUResult_EX2MEM_out),
323     .Mem_w_data(Rt_data_EX2MEM_out),
324     .Mem_w(MemW_EX2MEM_out),
325     .Mem_r(MemR_EX2MEM_out),
326     .clk(clk)
327 );
328
```

```
329 // MEM2WB stage
330 MEM_WB Mem_WB(
331     // Outputs
332     .ALUResult_out(ALUResult_MEM2WB_out),
333     .Mem_r_data_out(Mem_r_data_MEM2WB_out),
334     .RdAddr_out(RdAddr_MEM2WB_out),
335     .Mem2Reg_out(Mem2Reg_MEM2WB_out),
336     .RegWrite_out(RegWrite_MEM2WB_out),
337     // Inputs
338     .ALUResult(ALUResult_EX2MEM_out),
339     .Mem_r_data(Mem_r_data),
340     .RdAddr(RdAddr_EX2MEM_out),
341     .Mem2Reg(Mem2Reg_EX2MEM_out),
342     .RegWrite(RegWrite_EX2MEM_out),
343     .clk(clk)
344 );
345
346 // WB stage
347 DMUX Mem_MUX1(
348     // Outputs
349     .data(WB_data),
350     // Inputs
351     .src1(ALUResult_MEM2WB_out),
352     .src2(Mem_r_data_MEM2WB_out),
353     .ctrl(Mem2Reg_MEM2WB_out)
354 );
355
356 endmodule
357
```

Description

A pipelined CPU architecture with five pipeline stages: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory (MEM), and Write-Back (WB)

IO

Inputs:

Port	Width	Description
Input_Addr	32	PC
clk	1	Clock signal

Outputs:

Port	Width	Description
PC_Write	1	~stall
Output_Addr	32	PC + 4

Explanation

Line 37~85

Wires that connecting the components. They are sorted by their corresponding stage.

Line 86~

Components’ instances. They are mostly sorted by their corresponding stage.

Each pipeline stage has its corresponding registers (**IF_ID**, **ID_EX**, **EX_MEM**, **MEM_WB**) to store intermediate results and control signals between pipeline stages.

Comparing to previous CPUs, this one uses data forwarding and stall to avoid data hazard.

Forwarding_Unit

```
1 module Forwarding_Unit(  
2     // output  
3     output reg [1:0] Forwarding_A_ctrl,  
4     output reg [1:0] Forwarding_B_ctrl,  
5     // input  
6     input [4:0] RsAddr_ID2EX,  
7     input [4:0] RtAddr_ID2EX,  
8     input [4:0] RdAddr_EX2MEM,  
9     input [4:0] RdAddr_MEM2WB,  
10    input RegWrite_EX2MEM,  
11    input RegWrite_MEM2WB  
12 );  
13  
14 // Forwarding A  
15 always @(*) begin  
16     // EX hazard  
17     if (RegWrite_EX2MEM && (RdAddr_EX2MEM != 0)) begin  
18         // Forwarding A  
19         if (RdAddr_EX2MEM == RsAddr_ID2EX) begin  
20             Forwarding_A_ctrl <= 2'b10;  
21         end  
22         else begin  
23             Forwarding_A_ctrl <= 2'b00;  
24         end  
25         // Forwarding B  
26         if (RdAddr_EX2MEM == RtAddr_ID2EX) begin  
27             Forwarding_B_ctrl <= 2'b10;  
28         end  
29         else begin  
30             Forwarding_B_ctrl <= 2'b00;  
31         end  
32     end  
33     // MEM hazard  
34     else if (RegWrite_MEM2WB && (RdAddr_MEM2WB != 0)) begin  
35         // Forwarding A  
36         if (RdAddr_MEM2WB == RsAddr_ID2EX) begin  
37             Forwarding_A_ctrl <= 2'b01;  
38         end  
39         else begin  
40             Forwarding_A_ctrl <= 2'b00;  
41         end  
42         // Forwarding B  
43         if (RdAddr_MEM2WB == RtAddr_ID2EX) begin  
44             Forwarding_B_ctrl <= 2'b01;  
45         end  
46         else begin  
47             Forwarding_B_ctrl <= 2'b00;  
48         end  
49     end  
50     else begin  
51         Forwarding_A_ctrl <= 2'b00;  
52         Forwarding_B_ctrl <= 2'b00;  
53     end  
54 end  
55 endmodule
```

Description

generates control signals (**Forwarding_A_ctrl** and **Forwarding_B_ctrl**) to indicate whether to forward data from the execution stage or memory stage to the instruction EX stage.

IO

Inputs:

Port	Width	Description
RsAddr_ID2EX	5 bits	Input address of source register (Rs) in the instruction decode/execute stage.
RtAddr_ID2EX	5 bits	Input address of target register (Rt) in the instruction decode/execute stage.
RdAddr_EX2MEM	5 bits	Input address of destination register (Rd) in the execution stage.

RdAddr_MEM2WB	5 bits	Input address of destination register (Rd) in the memory stage.
RegWrite_EX2MEM	1 bit	Input control signal indicating register write in the MEM stage.
RegWrite_MEM2WB	1 bit	Input control signal indicating register write in the WB stage.

Outputs:

Port	Width	Description
Forwarding_A_ctrl	2 bits	Output control signal for forwarding data to operand A in the instruction EX stage.
Forwarding_B_ctrl	2 bits	Output control signal for forwarding

		g data to operand B in the instruction EX stage.
--	--	--

Explanation

Detects data hazards by comparing the destination register addresses.

- ID/EX: no forwarding, just from the register file (00)
- EX/MEM: forwarded data from the prior ALU results (10)
- MEM/WB: from data memory or an earlier ALU result (01)

HD_Unit

```

1 module HD_Unit(
2     // Outputs
3     output reg ctrl = 1'b0,
4     // Inputs
5     input wire Mem_Read,
6     input wire [4:0] RtAddr_EX,
7     input wire [4:0] RsAddr_ID,
8     input wire [4:0] RtAddr_ID
9 );
10
11 always @(*) begin
12     if (Mem_Read &&
13         (RtAddr_EX == RsAddr_ID || RtAddr_EX == RtAddr_ID)) begin
14         ctrl <= 1'b1;
15     end
16     else begin
17         ctrl <= 1'b0;
18     end
19 end
20
21 endmodule

```

Description

Hazard Detection Unit identifies the hazard that cannot be solved by data forwarding.

IO

Inputs:

Port	Width	Description
------	-------	-------------

Mem_Read	1 bit	Input signal indicating a memory read operation in the EX stage.
RtAddr_EX	5 bits	Input address of the target register (Rt) in the execution stage (EX).
RsAddr_ID	5 bits	Input address of the source register (Rs) in the instruction decode stage (ID).
RtAddr_ID	5 bits	Input address of the target register (Rt) in the instruction decode stage (ID).

Outputs:

Port	Width	Description
ctrl	1 bit	Output control signal indicating the presence of a hazard.

Explanation

Sets the control signal (**ctrl**) to indicate the presence of a hazard,

enabling stall operation.

Stall_MUX

```

1 module Stall_MUX(
2     // Outputs
3     output wire RegDst_out,
4     output wire RegWrite_out,
5     output wire [1:0] ALU_op_out,
6     output wire ALU_src_out,
7     output wire Mem_w_out,
8     output wire Mem_r_out,
9     output wire Mem_to_Reg_out,
10    // Inputs
11    input wire RegDst,
12    input wire RegWrite,
13    input wire [1:0] ALU_op,
14    input wire ALU_src,
15    input wire Mem_w,
16    input wire Mem_r,
17    input wire Mem_to_Reg,
18    input wire stall_ctrl
19 );
20
21 assign RegDst_out = (stall_ctrl) ? 1'b0 : RegDst;
22 assign RegWrite_out = (stall_ctrl) ? 1'b0 : RegWrite;
23 assign ALU_op_out = (stall_ctrl) ? 2'b00 : ALU_op;
24 assign ALU_src_out = (stall_ctrl) ? 1'b0 : ALU_src;
25 assign Mem_w_out = (stall_ctrl) ? 1'b0 : Mem_w;
26 assign Mem_r_out = (stall_ctrl) ? 1'b0 : Mem_r;
27 assign Mem_to_Reg_out = (stall_ctrl) ? 1'b0 : Mem_to_Reg;
28
29 endmodule

```

Description

Control multiplexer outputs based on a stall control signal (**stall_ctrl**)

IO

Inputs:

All outputs of Control

Outputs:

Control signals that have the same amount to Control's output.

Explanation

Utilizes **assign** statements to control the outputs of multiplexers based on the stall control signal (**stall_ctrl**). When a stall is required, all outputs are forced to a default state.

TMUX

```

1 module TMUX(
2     // Outputs
3     output [31:0] data,
4     // Inputs
5     input [31:0] src1,
6     input [31:0] src2,
7     input [31:0] src3,
8     input [1:0] ctrl
9 );
10
11     assign data = (ctrl == 2'b00) ? src1 : (ctrl == 2'b01) ? src2 : src3;
12
13 endmodule
14

```

Description

Triple MUX. Selecting among 3 sources.

10

Inputs:

Port	Width	Description
src1	32 bits	Input data source 1.
src2	32 bits	Input data source 2.
src3	32 bits	Input data source 3.
ctrl	2 bits	Control signal determining the selected input data source.

Outputs:

Port	Width	Description
data	32 bits	Output data resulting from the multiplexing operation.

Explanation

- ctrl == 0, select src1
- ctrl == 1, select src2

- ctrl == 2, select src3


Part 1 Test

Program

	InstrAddr	Instruction	Binary Code	Hexadecimal
			SFCODE Rn Rt Rd Shift Imm	
4	0x0000_0000	addi \$t0, \$t0, \$t1	00000000_1110_1111_10100_0000_001101	0x01_0F_00_00
5	0x0000_0004	sdui \$t1, \$0, \$0	00000000_0100_0100_10101_00000_001101	0x01_28_0A_00
6	0x0000_0008	sll \$t2, \$0, 0x4	00000000_0100_0000_10110_00100_100110	0x01_00_01_25
7	0x0000_000C	sllv \$t3, \$t0, \$t1	00000000_1110_00010_10111_00000_110110	0x01_C7_08_36

1. $R[20] = R[30] + R[31] = \text{FFFF_FF00} + \text{FFFF_FFFF} = \text{FFFF_FEFF}$
2. $R[21] = R[9] - R[8] = \text{0000_FFFF} - \text{FFFF_0000} = \text{0001_FFFF}$
3. $R[22] = R[8] \ll \text{shamt} = \text{FFFF_0000} \ll 4 = \text{FFFF0_0000}$
4. $R[23] = R[30] \ll R[2][4:0] = \text{FFFF_FF00} \ll 02 = \text{FFFF_FC00}$

RF



21	ffffffeff
22	0001ffff
23	fff00000
24	ffffffc00

Part 2 Test

Program

	InstAddr	Instruction	Binary Code	Hexadecimal
			OpCode Rs Rt Immediate	
			OpCode Rs Rt Rd Shift Funct	
5	0x0000_0000	subl \$20, %0	0001101_00000_10100_1111_1111_10100	0x34FFFC
6	0x0000_0004	addu \$0, \$0	00000000_0000_00000_0000_0000_00101	0x00000000
7	0x0000_0008	addu \$0, \$0	00000000_0000_00000_0000_0000_00101	0x00000000
8	0x0000_000c	sw \$0, 0(\$20)	00100000_10100_10100_00000_00000_00000	0x42500000
9	0x0000_0010	addu \$0, \$0	00000000_0000_00000_0000_0000_00101	0x00000000
10	0x0000_0014	addu \$0, \$0	00000000_0000_00000_0000_0000_00101	0x00000000
11	0x0000_0018	lw \$21, 0(\$20)	0010001_10100_10101_00000_00000_00000	0x46670000
12	0x0000_001c	addu \$0, \$0	00000000_0000_00000_0000_0000_00101	0x00000000
13	0x0000_0020	addu \$0, \$0	00000000_0000_00000_0000_0000_00101	0x00000000
14	0x0000_0024	sll \$22, \$21, 21	00101010_10101_10110_00000_00000_00101	0xA8A00015

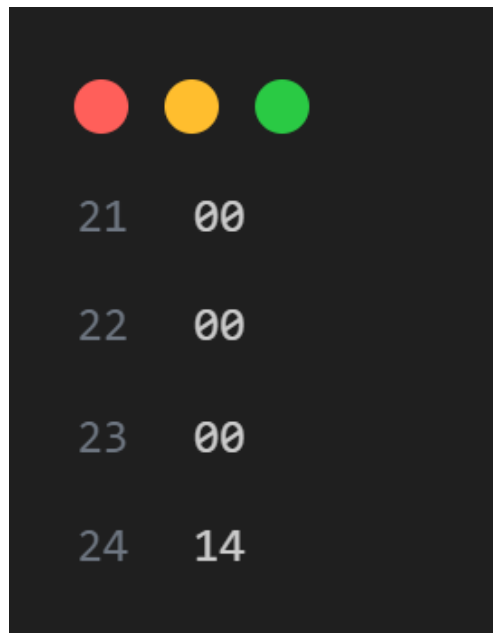
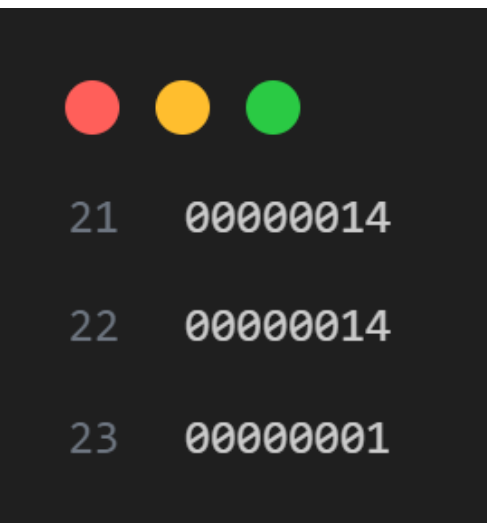
1. $R[20] = R[0] - 65516 = 0 + 0 \times 14 = 0 \times 14$
2. $R[0] = R[0] + R[0] = 0 + 0 = 0$, dummy op for avoiding data hazard.
3. $R[0] = R[0] + R[0] = 0 + 0 = 0$
4. $Mem[0 + R[20]] = Mem[0 \times 14] = \$20 = 0 \times 14$
5. $R[0] = R[0] + R[0] = 0 + 0 = 0$
6. $R[0] = R[0] + R[0] = 0 + 0 = 0$
7. $R[21] = Mem[0 + R[20]] = Mem[0 \times 14] = 0 \times 14$
8. $R[0] = R[0] + R[0] = 0 + 0 = 0$
9. $R[0] = R[0] + R[0] = 0 + 0 = 0$
10. $R[22] = (R[21] < 21) ? 1 : 0 = 1$

DM

A 4x2 grid of numbers on a dark gray background. Above the grid are three colored circles: red, yellow, and green. The numbers in the grid are:

21	00
22	00
23	00
24	14

RF DM

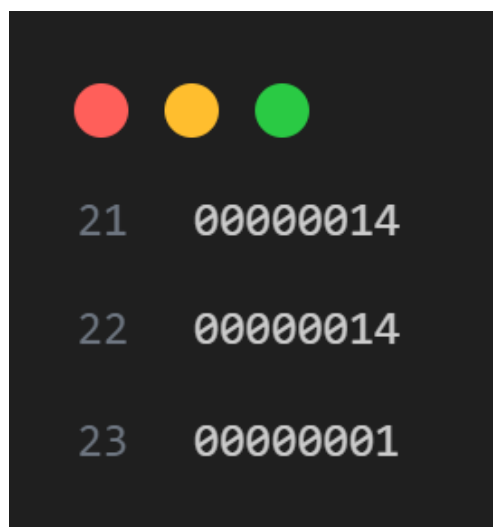


Part 3 Test Program

InstrAddr	Instruction	Binary Code	Hexadecimal
		OPCode Rs Rt Immediate	
		OPCode Rs Rt Rd Shift Funct	
0x0000_0000	subi \$20, \$0, 65516	0002100_0000_10100_11111_11111_10100	0x3410FFEC
0x0000_000c	sw \$20, 0(\$20)	0001000_10100_10100_00000_00000_00000	0x42540000
0x0000_0018	lw \$21, 0(\$20)	0001000_10100_10101_00000_00000_00000	0x46550000
0x0000_0024	slii \$22, \$21, 21	00101010_10101_10110_00000_010101	0xAAD00015

1. $R[20] = R[0] - 65516 = 0 + 0x14 = 0x14$
2. $Mem[0 + R[20]] = Mem[0x14] = \$20 = 0x14$
3. $R[21] = Mem[0 + R[20]] = Mem[0x14] = 0x14$
4. $R[22] = (R[21] < 21) ? 1 : 0 = 1$

RF



Simulation Result

Timing

Clock period = 6.375 ns

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 0.029 ns	Worst Hold Slack (WHS): 0.287 ns	Worst Pulse Width Slack (WPWS): 2.687 ns
Total Negative Slack (TNS): 0.000 ns	Total Hold Slack (THS): 0.000 ns	Total Pulse Width Negative Slack (TPWS): 0.000 ns
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 1087	Total Number of Endpoints: 1087	Total Number of Endpoints: 1047

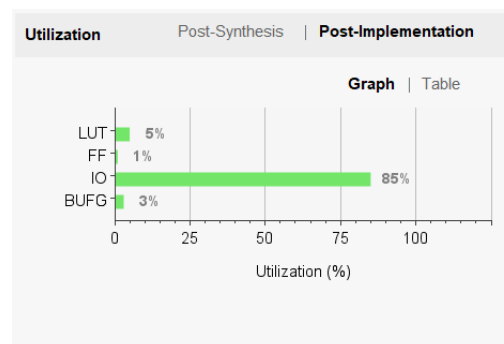
All user specified timing constraints are met.

```
CPU_xdc.xdc

D:/Programming/Verilog/ComputerOrganization/PA3/finalCPU/finalCPU.srcs/constrs_

1 create_clock -period 6.375 -name clk -waveform {0.000 3.1875} [get_ports clk]
```

Utilization



Power

Power	Summary	On-Chip
Total On-Chip Power:	0.17 W	
Junction Temperature:	27.0 °C	
Thermal Margin:	58.0 °C (4.9 W)	
Effective SJA:	11.5 °C/W	
Power supplied to off-chip devices:	0 W	
Confidence level:	Low	
Implemented Power Report		