台科大

PA3

30 頁真的寫不完

周 柏宇

2024/5/20

內容

Part 1

Adder

```verilog
module Adder(
    // Outputs
    output [31:0] OutputAddr,
    // Inputs
    input [31:0] InputAddr,
    input [31:0] Offset
);

    assign OutputAddr = InputAddr + Offset;

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  `define ADDU 6'b001011
2  `define SUBU 6'b001101
3  `define SLL 6'b100110
4  `define SLLV 6'b110110
5
6  module ALU_Control(
7      // Outputs
8      output reg [5:0] funct,
9      // Inputs
10     input [1:0] ALU_op,
11     input [5:0] funct_ctrl
12 );
13
14     always @(*) begin
15         case(ALU_op)
16             2'b00: funct = 6'b001001;
17             2'b01: funct = 6'b001010;
18             2'b10: begin
19                 case(funct_ctrl)
20                     `ADDU: funct = 6'b001001;
21                     `SUBU: funct = 6'b001010;
22                     `SLL: funct = 6'b100001;
23                     `SLLV: funct = 6'b110101;
24                     default: funct = 6'b000000;
25                 endcase
26             end
27             2'b11: funct = 6'b101010;
28             default: funct = 6'b000000;
29         endcase
30     end
31
32 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  `define ADDU 6'b001001
2  `define SUBU 6'b001010
3  `define SLL 6'b100001
4  `define SLLV 6'b110101
5  `define SLTI 6'b101010
6
7  module ALU(
8      // Outputs
9      output reg [31:0] ALU_result,
10     // Inputs
11     input [31:0] data1,
12     input [31:0] data2,
13     input [4:0] shamt,
14     input [5:0] funct
15 );
16
17     wire [31:0] ADDU_result, SUBU_result, SLL_result, S
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
37
```

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

```
`define DATA_MEM_SIZE   128 // Bytes

/*
 * Declaration of Data Memory for this project.
 * CAUTION: DONT MODIFY THE NAME.
 */
module DM(
    // Outputs
    output [31:0] Mem_r_data,
    // Inputs
    input [31:0] Mem_addr,
    input [31:0] Mem_w_data,
    input Mem_w,
    input Mem_r,
    input clk
);

    /*
     * Declaration of data memory.
     * CAUTION: DONT MODIFY THE NAME AND SIZE.
     */
    reg [7:0]DataMem[0:`DATA_MEM_SIZE - 1];

    // write data to memory
    always @(negedge clk) begin
        if(Mem_w) begin
            DataMem[Mem_addr[6:0]] <= Mem_w_data[31:24];
            DataMem[Mem_addr[6:0] + 1] <= Mem_w_data[23:16];
            DataMem[Mem_addr[6:0] + 2] <= Mem_w_data[15:8];
            DataMem[Mem_addr[6:0] + 3] <= Mem_w_data[7:0];
        end
    end

    // read data from memory
    assign Mem_r_data = {DataMem[Mem_addr[6:0]], DataMem[Mem_addr[6:0] + 1]
                , DataMem[Mem_addr[6:0] + 2], DataMem[Mem_addr[6:0] + 3]};

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

```
module EX_MEM (
    // Outputs
    output reg [31:0] ALU_result_out,
    output reg [31:0] Rt_data_out,
    output reg [4:0] RdAddr_out,
    output reg MemW_out,
    output reg MemR_out,
    output reg Mem2Reg_out,
    output reg RegWrite_out,
    // Inputs
    input [31:0] ALU_result,
    input [31:0] Rt_data,
    input [4:0] RdAddr,
    input MemW,
    input MemR,
    input Mem2Reg,
    input RegWrite,
    input clk
);
    always @(negedge clk) begin
        ALU_result_out <= ALU_result;
        Rt_data_out <= Rt_data;
        RdAddr_out <= RdAddr;
        MemW_out <= MemW;
        MemR_out <= MemR;
        Mem2Reg_out <= Mem2Reg;
        RegWrite_out <= RegWrite;
    end

endmodule

```

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 bits	Data from 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

```
module GPRMUX(
    // Outputs
    output [31:0] data,
    // Inputs
    input [31:0] src1,
    input [31:0] src2,
    input ctrl
);

    assign data = ctrl ? src2 : src1;

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

```
module ID_EX (
    output reg RegDst_out,
    output reg RegWrite_out,
    output reg [1:0] ALU_op_out,
    output reg ALU_src_out,
    output reg Mem_w_out,
    output reg Mem_r_out,
    output reg Mem_to_Reg_out,

    output reg [31:0] rs_out,
    output reg [31:0] rt_out,
    output reg [4:0] rt_addr_out,
    output reg [4:0] rd_addr_out,
    output reg [31:0] imm_out,

    input RegDst,
    input RegWrite,
    input [1:0] ALU_op,
    input ALU_src,
    input Mem_w,
    input Mem_r,
    input Mem_to_Reg,

    input [31:0] rs,
    input [31:0] rt,
    input [4:0] rt_addr,
    input [4:0] rd_addr,
    input [31:0] imm,

    input clk
);
    always @(negedge clk) begin
        rs_out <= rs;
        rt_out <= rt;
        rt_addr_out <= rt_addr;
        rd_addr_out <= rd_addr;
        imm_out <= imm;

        RegDst_out <= RegDst;
        RegWrite_out <= RegWrite;
        ALU_op_out <= ALU_op;
        ALU_src_out <= ALU_src;
        Mem_w_out <= Mem_w;
        Mem_r_out <= Mem_r;
        Mem_to_Reg_out <= Mem_to_Reg;
    end

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 desination.
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

```
module IF_ID (
    // Outputs
    output reg [31:0] instr_out,
    // Inputs
    input [31:0] instr,
    input clk
);
    reg init = 0;

    always @(negedge clk) begin
        if(init == 0) begin
            instr_out <= 32'bz;
            init <= 1;
        end
        else begin
            instr_out <= instr;
        end
    end

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

```
`define INSTR_MEM_SIZE  128 // Bytes

/*
 * Declaration of Instruction Memory for this project.
 * CAUTION: DONT MODIFY THE NAME.
 */
module IM(
    // Outputs
    output [31:0] Instruction,
    // Inputs
    input [31:0] InputAddr
);

    /*
     * Declaration of instruction memory.
     * CAUTION: DONT MODIFY THE NAME AND SIZE.
     */
    reg [7:0]InstrMem[0:`INSTR_MEM_SIZE - 1];

    // fetch instruction from memory with big-endian
    assign Instruction = {InstrMem[InputAddr[6:0]], InstrMem[InputAddr[6:0] + 1],
                          InstrMem[InputAddr[6:0] + 2], InstrMem[InputAddr[6:0] + 3]};

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

```
module MEM_WB (
    // Outputs
    output reg [31:0] ALUResult_out,
    output reg [31:0] Mem_r_data_out,
    output reg [4:0] RdAddr_out,
    output reg Mem2Reg_out,
    output reg RegWrite_out,
    // Inputs
    input [31:0] ALUResult,
    input [31:0] Mem_r_data,
    input [4:0] RdAddr,
    input Mem2Reg,
    input RegWrite,
    input clk
);
    always @(negedge clk) begin
        ALUResult_out <= ALUResult;
        Mem_r_data_out <= Mem_r_data;
        RdAddr_out <= RdAddr;
        Mem2Reg_out <= Mem2Reg;
        RegWrite_out <= RegWrite;
    end

endmodule
```

Description

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

IO

Inputs:

Port	Widt	Descriptio

	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.

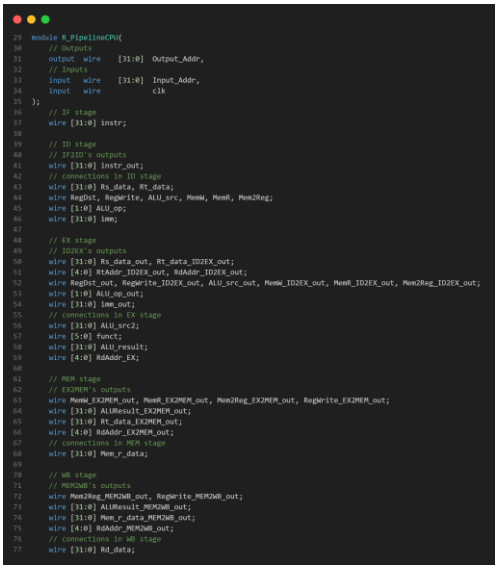
Outputs:

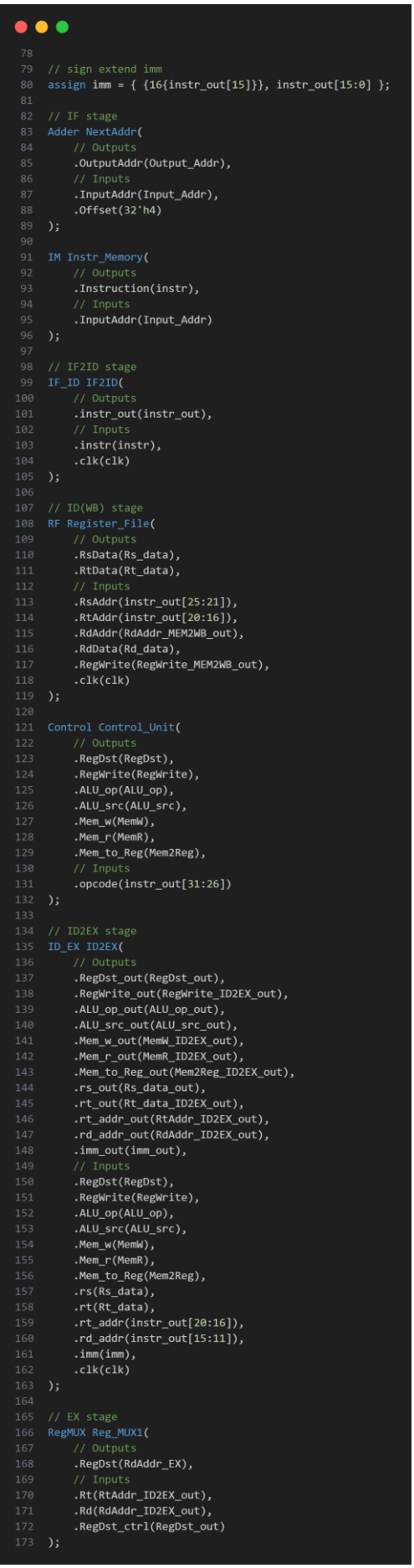
Port	Width	Description
ALUResult_out	32 bits	Output result from the ALU to be used in the WB stage.
Mem_r_data_out	32 bits	Output data read from memory to be used in the WB stage.
RdAddr_out	5 bits	Output address of the destination register.
Mem2Reg_out	1 bit	Output control signal for memory to register write.
RegWrite_out	1 bit	Output control signal enabling the register write.

Explanation

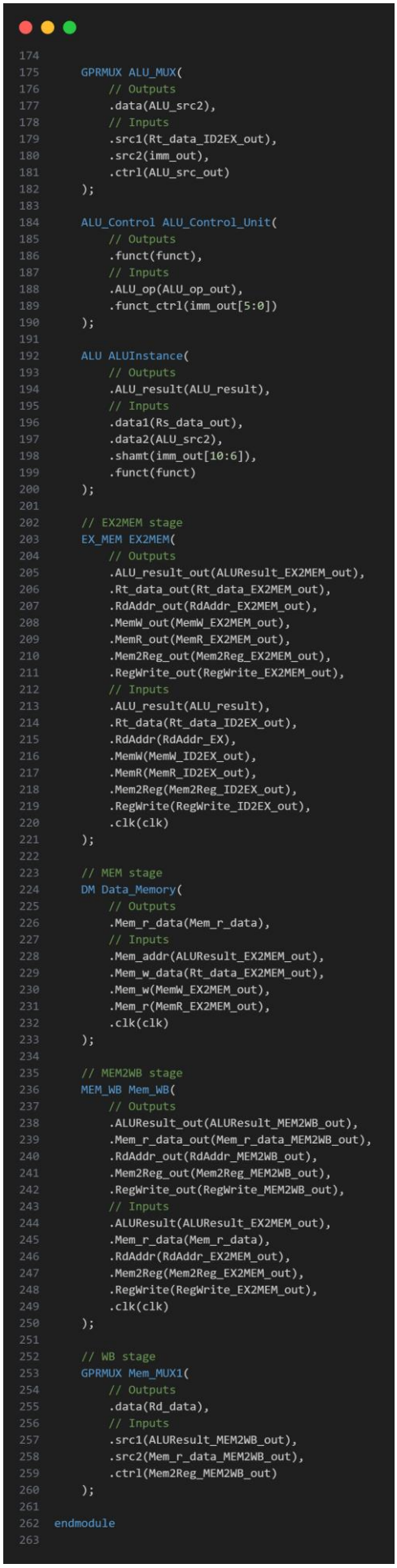
Holds the input values and control signals on the falling edge of the clock (**negedge clk**).

R_PipelineCPU





```
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

```v
module RegMUX(
    // Outputs
    output [4:0] RegDst,
    // Inputs
    input [4:0] Rt,
    input [4:0] Rd,
    input RegDst_ctrl
);

    assign RegDst = (RegDst_ctrl) ? Rd : Rt;

endmodule

```

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

```v
`define REG_MEM_SIZE    32  // Words

/*
 * Declaration of Register File for this project.
 * CAUTION: DONT MODIFY THE NAME.
 */
module RF(
    // Outputs
    output [31:0] RsData,
    output [31:0] RtData,
    // Inputs
    input [4:0] RsAddr,
    input [4:0] RtAddr,
    input [4:0] RdAddr,
    input [31:0] RdData,
    input RegWrite,
    input clk
);

    /*
     * Declaration of inner register.
     * CAUTION: DONT MODIFY THE NAME AND SIZE.
     */
    reg [31:0]R[0:`REG_MEM_SIZE - 1];

    assign RsData = R[RsAddr];
    assign RtData = R[RtAddr];

    always @(*) begin
        if(RegWrite && !clk)
            R[RdAddr] <= RdData;
    end

endmodule

```

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.”

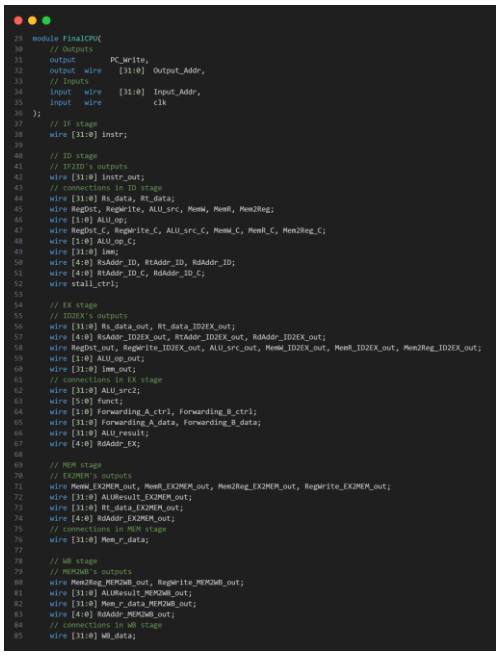
Part 3

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



```
module FinalCPU(
    // Outputs
    output        PC_Write,
    output  wire  [31:0]  Output_Addr,
    // Inputs
    input   wire  [31:0]  Input_Addr,
    input   wire          clk
);
    // IF stage
    wire [31:0] instr;

    // ID stage
    // IF2ID's outputs
    wire [31:0] instr_out;
    // connections in ID stage
    wire [31:0] Rs_data, Rt_data;
    wire RegDst, RegWrite, ALU_src, MemW, MemR, Mem2Reg;
    wire [1:0] ALU_op;
    wire RegDst_C, RegWrite_C, ALU_src_C, MemW_C, MemR_C, Mem2Reg_C;
    wire [1:0] ALU_op_C;
    wire [31:0] imm;
    wire [4:0] RsAddr_ID, RtAddr_ID, RdAddr_ID;
    wire [4:0] RtAddr_ID_C, RdAddr_ID_C;
    wire stall_ctrl;

    // EX stage
    // ID2EX's outputs
    wire [31:0] Rs_data_out, Rt_data_ID2EX_out;
    wire [4:0] RsAddr_ID2EX_out, RtAddr_ID2EX_out, RdAddr_ID2EX_out;
    wire RegDst_out, RegWrite_ID2EX_out, ALU_src_out, MemW_ID2EX_out, MemR_ID2EX_out, Mem2Reg_ID2EX_out;
    wire [1:0] ALU_op_out;
    wire [31:0] imm_out;
    // connections in EX stage
    wire [31:0] ALU_src2;
    wire [5:0] funct;
    wire [1:0] Forwarding_A_ctrl, Forwarding_B_ctrl;
    wire [31:0] Forwarding_A_data, Forwarding_B_data;
    wire [31:0] ALU_result;
    wire [4:0] RdAddr_EX;

    // MEM stage
    // EX2MEM's outputs
    wire MemW_EX2MEM_out, MemR_EX2MEM_out, Mem2Reg_EX2MEM_out, RegWrite_EX2MEM_out;
    wire [31:0] ALUResult_EX2MEM_out;
    wire [31:0] Rt_data_EX2MEM_out;
    wire [4:0] RdAddr_EX2MEM_out;
    // connections in MEM stage
    wire [31:0] Mem_r_data;

    // WB stage
    // MEM2WB's outputs
    wire Mem2Reg_MEM2WB_out, RegWrite_MEM2WB_out;
    wire [31:0] ALUResult_MEM2WB_out;
    wire [31:0] Mem_r_data_MEM2WB_out;
    wire [4:0] RdAddr_MEM2WB_out;
    // connections in WB stage
    wire [31:0] WB_data;
```

```
// IF stage
Adder NextAddr(
    // Outputs
    .OutputAddr(Output_Addr),
    // Inputs
    .InputAddr(Input_Addr),
    .Offset(32'h4)
);

IM Instr_Memory(
    // Outputs
    .Instruction(instr),
    // Inputs
    .InputAddr(Input_Addr)
);

// IF2ID stage
IF_ID IF2ID(
    // Outputs
    .instr_out(instr_out),
    // Inputs
    .instr(stall_ctrl ? instr_out : instr),
    .clk(clk)
);

// ID(WB) stage
assign imm = { {16{instr_out[15]}}, instr_out[15:0] };
assign RsAddr_ID = instr_out[25:21];
assign RtAddr_ID_C = instr_out[20:16];
assign RdAddr_ID_C = instr_out[15:11];
assign PC_Write = ~stall_ctrl;

RF Register_File(
    // Outputs
    .RsData(Rs_data),
    .RtData(Rt_data),
    // Inputs
    .RsAddr(RsAddr_ID),
    .RtAddr(RtAddr_ID_C),
    .RdAddr(RdAddr_MEM2WB_out),
    .RdData(WB_data),
    .RegWrite(RegWrite_MEM2WB_out),
    .clk(clk)
);
```

```
HD_Unit HD_Unit(
    // Outputs
    .ctrl(stall_ctrl),
    // Inputs
    .Mem_Read(MemR_ID2EX_out),
    .RtAddr_EX(RtAddr_ID2EX_out),
    .RsAddr_ID(RsAddr_ID),
    .RtAddr_ID(RtAddr_ID_C)
);

Control Control_Unit(
    // Outputs
    .RegDst(RegDst_C),
    .RegWrite(RegWrite_C),
    .ALU_op(ALU_op_C),
    .ALU_src(ALU_src_C),
    .Mem_w(MemW_C),
    .Mem_r(MemR_C),
    .Mem_to_Reg(Mem2Reg_C),
    // Inputs
    .opcode(instr_out[31:26])
);

Stall_MUX Stall_MUX_Control(
    // Outputs
    .RegDst_out(RegDst),
    .RegWrite_out(RegWrite),
    .ALU_op_out(ALU_op),
    .ALU_src_out(ALU_src),
    .Mem_w_out(MemW),
    .Mem_r_out(MemR),
    .Mem_to_Reg_out(Mem2Reg),
    // Inputs
    .RegDst(RegDst_C),
    .RegWrite(RegWrite_C),
    .ALU_op(ALU_op_C),
    .ALU_src(ALU_src_C),
    .Mem_w(MemW_C),
    .Mem_r(MemR_C),
    .Mem_to_Reg(Mem2Reg_C),
    .stall_ctrl(stall_ctrl)
);

RegMUX Stall_MUX_Rt(
    // Outputs
    .RegDst(RtAddr_ID),
    // Inputs
    .addr1(RtAddr_ID_C),
    .addr2(5'b00000),
    .RegDst_ctrl(stall_ctrl)
);

RegMUX Stall_MUX_Rd(
    // Outputs
    .RegDst(RdAddr_ID),
    // Inputs
    .addr1(RdAddr_ID_C),
    .addr2(5'b00000),
    .RegDst_ctrl(stall_ctrl)
);
```

```
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

```
module Forwarding_Unit(
    // output
    output reg [1:0] Forwarding_A_ctrl,
    output reg [1:0] Forwarding_B_ctrl,
    // input
    input [4:0] RsAddr_ID2EX,
    input [4:0] RtAddr_ID2EX,
    input [4:0] RdAddr_EX2MEM,
    input [4:0] RdAddr_MEM2WB,
    input RegWrite_EX2MEM,
    input RegWrite_MEM2WB
);

    // Forwarding A
    always @(*) begin
        // EX hazard
        if (RegWrite_EX2MEM && (RdAddr_EX2MEM != 0)) begin
            // Forwarding A
            if (RdAddr_EX2MEM == RsAddr_ID2EX) begin
                Forwarding_A_ctrl <= 2'b10;
            end
            else begin
                Forwarding_A_ctrl <= 2'b00;
            end
            // Forwarding B
            if (RdAddr_EX2MEM == RtAddr_ID2EX) begin
                Forwarding_B_ctrl <= 2'b10;
            end
            else begin
                Forwarding_B_ctrl <= 2'b00;
            end
        end
        // MEM hazard
        else if (RegWrite_MEM2WB && (RdAddr_MEM2WB != 0)) begin
            // Forwarding A
            if (RdAddr_MEM2WB == RsAddr_ID2EX) begin
                Forwarding_A_ctrl <= 2'b01;
            end
            else begin
                Forwarding_A_ctrl <= 2'b00;
            end
            // Forwarding B
            if (RdAddr_MEM2WB == RtAddr_ID2EX) begin
                Forwarding_B_ctrl <= 2'b01;
            end
            else begin
                Forwarding_B_ctrl <= 2'b00;
            end
        end
        else begin
            Forwarding_A_ctrl <= 2'b00;
            Forwarding_B_ctrl <= 2'b00;
        end
    end
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 forwardin

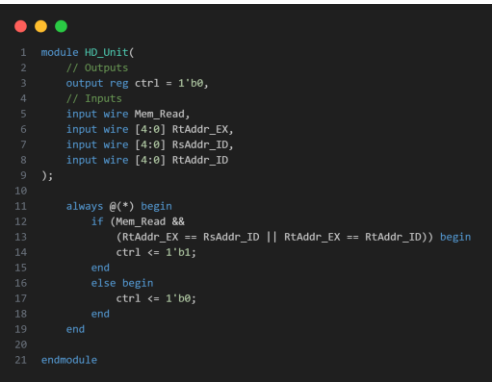
		g data to operand B in the instructio n EX stage.

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



```
module HD_Unit(
    // Outputs
    output reg ctrl = 1'b0,
    // Inputs
    input wire Mem_Read,
    input wire [4:0] RtAddr_EX,
    input wire [4:0] RsAddr_ID,
    input wire [4:0] RtAddr_ID
);

    always @(*) begin
        if (Mem_Read &&
            (RtAddr_EX == RsAddr_ID || RtAddr_EX == RtAddr_ID)) begin
            ctrl <= 1'b1;
        end
        else begin
            ctrl <= 1'b0;
        end
    end

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**).

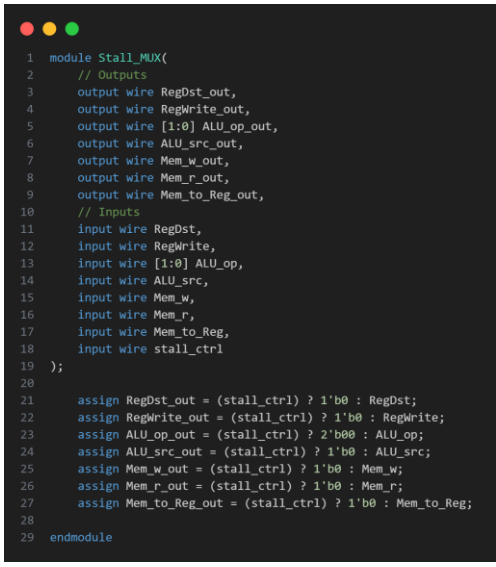
Outputs:

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



```
module Stall_MUX(
    // Outputs
    output wire RegDst_out,
    output wire RegWrite_out,
    output wire [1:0] ALU_op_out,
    output wire ALU_src_out,
    output wire Mem_w_out,
    output wire Mem_r_out,
    output wire Mem_to_Reg_out,
    // Inputs
    input wire RegDst,
    input wire RegWrite,
    input wire [1:0] ALU_op,
    input wire ALU_src,
    input wire Mem_w,
    input wire Mem_r,
    input wire Mem_to_Reg,
    input wire stall_ctrl
);

    assign RegDst_out = (stall_ctrl) ? 1'b0 : RegDst;
    assign RegWrite_out = (stall_ctrl) ? 1'b0 : RegWrite;
    assign ALU_op_out = (stall_ctrl) ? 2'b00 : ALU_op;
    assign ALU_src_out = (stall_ctrl) ? 1'b0 : ALU_src;
    assign Mem_w_out = (stall_ctrl) ? 1'b0 : Mem_w;
    assign Mem_r_out = (stall_ctrl) ? 1'b0 : Mem_r;
    assign Mem_to_Reg_out = (stall_ctrl) ? 1'b0 : Mem_to_Reg;

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

```v
module TMUX(
    // Outputs
    output [31:0] data,
    // Inputs
    input [31:0] src1,
    input [31:0] src2,
    input [31:0] src3,
    input [1:0] ctrl
);

    assign data = (ctrl == 2'b00) ? src1 : (ctrl == 2'b01) ? src2 : src3;

endmodule

```

Description

Triple MUX. Selecting among 3 sources.

IO

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
		OPCode Rs Rt Rd Shamt Funct	
0x0000_0000	addu $30, $31	0b000000_11110_11111_10100_00000_001011	0x03_DF_A0_0B
0x0000_0004	subu $21, $9, $8	0b000000_01001_01000_10101_00000_001011	0x01_28_A8_0D
0x0000_0008	sllv $22, $8, 0x4	0b000000_01000_00000_10110_00100_100110	0x01_00_B1_26
0x0000_000C	sllv $23, $30, $2	0b000000_11110_00010_10111_00000_110110	0x03_C2_B8_36

1. R[20]=R[30]+R[31]=FFFF_FF00+FFFF_FFFF=FFFF_FEFF
2. R[21]=R[9]-R[8]=0000_FFFF-FFFF_0000=0001_FFFF
3. R[22]=R[8]<<shamt=FFFF_0000<<4=FFF0_0000
4. R[23]=R[30]<<R[2][4:0]=FFFF_FF00<<02=FFFF_FC00

RF

```
21  ffffeff
22  0001ffff
23  fff00000
24  fffffc00
```

Part 2 Test

Program

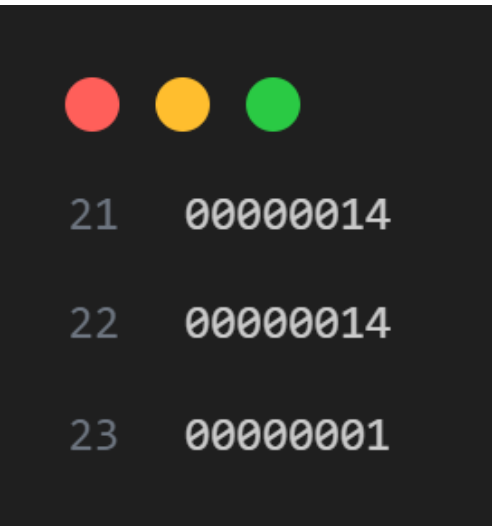
InstrAddr	Instruction	Binary Code	Hexadecimal
		OPCode Rs Rt Immediate	
		OPCode Rs Rt Rd Shamt Funct	
0x0000_0000	subi $20, $0, 65516	0b001101_00000_10100_11111_11111_101100	0x3414FFEC
0x0000_0004	addu $0, $0, $0	0b000000_00000_00000_00000_00000_001011	0x0000000B
0x0000_0008	addu $0, $0, $0	0b000000_00000_00000_00000_00000_001011	0x0000000B
0x0000_000c	sw $20, 0($20)	0b010000_10100_10100_00000_00000_000000	0x42940000
0x0000_0010	addu $0, $0, $0	0b000000_00000_00000_00000_00000_001011	0x0000000B
0x0000_0014	addu $0, $0, $0	0b000000_00000_00000_00000_00000_001011	0x0000000B
0x0000_0018	lw $21, 0($20)	0b100001_10100_10101_00000_00000_000000	0x46950000
0x0000_001c	addu $0, $0, $0	0b000000_00000_00000_00000_00000_001011	0x0000000B
0x0000_0020	addu $0, $0, $0	0b000000_00000_00000_00000_00000_001011	0x0000000B
0x0000_0024	slti $22, $21, 21	0b101010_10101_10110_00000_00000_010101	0xAAB60015

1. R[20]=R[0]-65516=0+0x14=0x14
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[0x14]=$20=0x14
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[0x14]=0x14
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

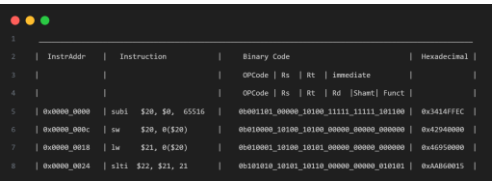
```
21  00
22  00
23  00
24  14
```

RF



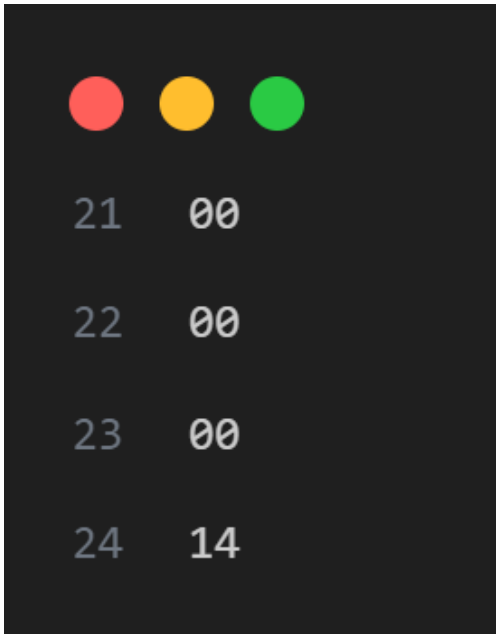
Part 3 Test

Program

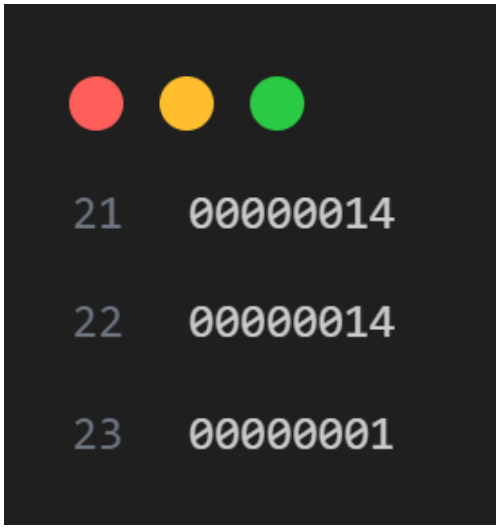


1. R[20]=R[0]-65516=0+0x14=0x14
2. Mem[0+R[20]]=Mem[0x14]=$20 =0x14
3. R[21]=Mem[0+R[20]]=Mem[0x14]=0x14
4. R[22]=(R[21]<21)?1:0=1

DM



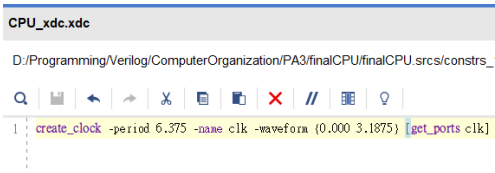
RF



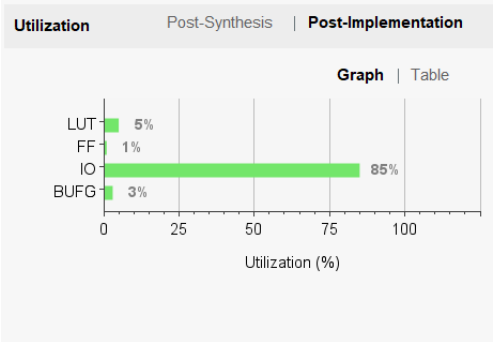
Simulation Result

Timing

Clock period = 6.375 ns



Utilization



Power