Report for Lab5: Pipelined CPU

20140843 Taekang Eom

20150384 Eunyoung Hyung

1. Introduction

In this lab, we need to understand why the pipelined CPU has better throughput than non-pipelined CPU. To design and implement the pipelined CPU, we also need to understand the data hazard and control hazard and how to solve them. For data hazard, we implement Forwarding and Stall. For control hazard, we take the Always-not-taken Branch Prediction strategy and so we implement Flush.

2. Design

1) Forwarding

We have one 4-way MUX before the ALU. (The MUX is actually work as 3-way but just for convenience we implement this in 4-way.) If the rs1 in the ID stage is same as the rd in EX or MEM stages, then we set the ForwardA bit differently. In the same way, for rs2, we set the ForwardB.

According to these Forward bit, this MUX chooses the operand (e.g. rs which is just from RF, forwarded from prior ALU result, from DM or earlier result).

2) Stall

Even with Forwarding, a stall is unavoidable in the case of Load instructions. So we need to stall when the instruction is Load in EX stage and the rs1 or rs2 in ID stage are same as rd in EX stage.

With this bit for stall, we do not process IF, ID and EX stages.

3) Flush

In the EX stage, if the pipeline figures out that this instruction is Jump or Branch (with satisfied condition) instruction then Flush bit is set to 1. If this Flush bit is set, the target address of Control instructions which is in ALUResult are fetched to PC and the stages after the branch is flushed while the stages before the branch are processed as nothing happens.

4) The structure of the registers

We use two 85-bit length registers as below.

		reg_EX	reg_MEM
bit	signal	reg_EX	reg_MEM
0	Jump	Decoded signal	Decoded signal
1	JALorJALR	Decoded signal	Decoded signal
2	Branch	Decoded signal	Decoded signal
3	MemRead	Decoded signal	Decoded signal
4	MemWrite	Decoded signal	Decoded signal
5	RegWrite	Decoded signal	Decoded signal
6	MemtoReg	Decoded signal	Decoded signal
7	PCtoReg	Decoded signal	Decoded signal
8	OpenPort	Decoded signal	Decoded signal
9	Halt	Decoded signal	Decoded signal
10	ALUOp	Decoded signal	Decoded signal
11	ALUOp	Decoded signal	Decoded signal
12	ALUOp	Decoded signal	Decoded signal
13	ALUOp	Decoded signal	Decoded signal
14	rs1	Decoded signal	Decoded signal
15	rs1	Decoded signal	Decoded signal
16	rs2	Decoded signal	Decoded signal
17	rs2	Decoded signal	Decoded signal
18	rd	Decoded signal	Decoded signal
19	rd	Decoded signal	Decoded signal
20	ALUSrc	Forwarded rs1Data	Forwarded rs1Data
21		Forwarded rs1Data	Forwarded rs1Data
22		Forwarded rs1Data	Forwarded rs1Data
23		Forwarded rs1Data	Forwarded rs1Data

24		Forwarded rs1Data	Forwarded rs1Data
25		Forwarded rs1Data	Forwarded rs1Data
26		Forwarded rs1Data	Forwarded rs1Data
27		Forwarded rs1Data	Forwarded rs1Data
28		Forwarded rs1Data	Forwarded rs1Data
29		Forwarded rs1Data	Forwarded rs1Data
30		Forwarded rs1Data	Forwarded rs1Data
31		Forwarded rs1Data	Forwarded rs1Data
32		Forwarded rs1Data	Forwarded rs1Data
33		Forwarded rs1Data	Forwarded rs1Data
34		Forwarded rs1Data	Forwarded rs1Data
35		Forwarded rs1Data	Forwarded rs1Data
36		Forwarded rs2Data	Forwarded rs2Data
37		Forwarded rs2Data	Forwarded rs2Data
38		Forwarded rs2Data	Forwarded rs2Data
39		Forwarded rs2Data	Forwarded rs2Data
40		Forwarded rs2Data	Forwarded rs2Data
41		Forwarded rs2Data	Forwarded rs2Data
42		Forwarded rs2Data	Forwarded rs2Data
43		Forwarded rs2Data	Forwarded rs2Data
44		Forwarded rs2Data	Forwarded rs2Data
45		Forwarded rs2Data	Forwarded rs2Data
46		Forwarded rs2Data	Forwarded rs2Data
47		Forwarded rs2Data	Forwarded rs2Data
48		Forwarded rs2Data	Forwarded rs2Data
49		Forwarded rs2Data	Forwarded rs2Data
50		Forwarded rs2Data	Forwarded rs2Data
51		Forwarded rs2Data	Forwarded rs2Data
52		Current pc + 1	Current pc + 1
53		Current pc + 1	Current pc + 1
54		Current pc + 1	Current pc + 1
55		Current pc + 1	Current pc + 1
56		Current pc + 1	Current pc + 1
57		Current pc + 1	Current pc + 1
58		Current pc + 1	Current pc + 1
59		Current pc + 1	Current pc + 1
60		Current pc + 1	Current pc + 1
61		Current pc + 1	Current pc + 1

62		Current pc + 1	Current pc + 1
63		Current pc + 1	Current pc + 1
64		Current pc + 1	Current pc + 1
65		Current pc + 1	Current pc + 1
66		Current pc + 1	Current pc + 1
67		Current pc + 1	Current pc + 1
68		isInst	isInst
69		instruction	ALUResult
70		instruction	ALUResult
71		instruction	ALUResult
72		instruction	ALUResult
73		instruction	ALUResult
74		instruction	ALUResult
75		instruction	ALUResult
76		instruction	ALUResult
77		instruction	ALUResult
78		instruction	ALUResult
79		instruction	ALUResult
80		instruction	ALUResult
81		instruction(opcode)	ALUResult
82		instruction(opcode)	ALUResult
83		instruction(opcode)	ALUResult
84		instruction(opcode)	ALUResult

3. Implementation

We only address the newly added or updated modules. We have a new unit for forwarding and data hazard detection in one module called "Data_Hazard_Unit". We define some numbers in "opcodes" for better readability. Of course "CPU" is functionally changed into pipelined-CPU. However, "control" is slightly changed but it's functionally same as before and "ALU" is neither changed so we do not address them here.

1) Opcodes.v

The last two parts with the comment "Bit for signal" and "Bit for pipeline register" is based on the table above in [2. Design]

```
// Opcode
```

```
`define ALU_OP  4'd15
`define ADI_OP  4'd4
`define ORI_OP  4'd5
`define LHI_OP  4'd6
`define LWD_OP  4'd7
`define SWD_OP  4'd8
`define BNE_OP  4'd0
`define BEQ_OP  4'd1
`define BGZ_OP  4'd2
`define BLZ_OP  4'd3
`define JMP_OP  4'd9
`define JAL_OP  4'd10
`define JPR_OP  4'd15
`define JRL_OP  4'd15

// ALU Function Codes
`define FUNC_ADD    3'b000
`define FUNC_SUB    3'b001
`define FUNC_AND    3'b010
`define FUNC_ORR    3'b011
`define FUNC_NOT    3'b100
`define FUNC_TCP    3'b101
`define FUNC_SHL    3'b110
`define FUNC_SHR    3'b111

// ALU instruction function codes
`define INST_FUNC_ADD 6'd0
`define INST_FUNC_SUB 6'd1
`define INST_FUNC_AND 6'd2
`define INST_FUNC_ORR 6'd3
`define INST_FUNC_NOT 6'd4
`define INST_FUNC_TCP 6'd5
`define INST_FUNC_SHL 6'd6
`define INST_FUNC_SHR 6'd7
`define INST_FUNC_JPR 6'd25
`define INST_FUNC_JRL 6'd26
`define INST_FUNC_WWD 6'd28
`define INST_FUNC_HALT 6'd29

`define WORD_SIZE   16
`define NUM_REGS    4

// Bit for signal
`define Jump 0
`define JALorJALR 1
`define Branch 2
`define MemRead 3
`define MemWrite 4
```

```
`define RegWrite 5
`define MemtoReg 6
`define PCtoReg 7
`define OpenPort 8
`define Halt 9
`define ALUOp 10
`define rs1 14
`define rs2 16
`define rd 18
`define ALUSrc 20

// Bit for pipeline register
`define rs1Data 20
`define rs2Data 36
`define expPC 52
`define isInst 68
`define instruction 69
`define ALUResult 69
```

2) Data Hazard Unit

You can refer the comments.

```
`include "opcodes.v"

module Data_Hazard_Unit(
    // CPU calls this like the comments below
    input wire isInst_ID_EX, // .isInst_ID_EX(reg_EX[`isInst])
    input wire isInst_EX_MEM, // .isInst_EX_MEM(reg_MEM[`isInst])
    input wire [1:0] rs1, // rs1(signal[`rs1+1:`rs1])
    input wire [1:0] rs2, // .rs2(signal[`rs2+1:`rs2])
    input wire [1:0] rd_ID_EX, // .rd_ID_EX(reg_EX[`rd+1:`rd])
    input wire [1:0] rd_EX_MEM, // .rd_EX_MEM(reg_MEM[`rd+1:`rd])
    input wire MemRead_ID_EX, // .MemRead_ID_EX(reg_EX[`MemRead])
    input wire RegWrite_ID_EX, // .RegWrite_ID_EX(reg_EX[`RegWrite])
    input wire RegWrite_EX_MEM, // .RegWrite_EX_MEM(reg_MEM[`RegWrite])
    input wire Jump, // .Jump(signal[`Jump])
    input wire Branch, // .Branch(signal[`Branch])
    input wire ALUSrc, // .ALUSrc(signal[`ALUSrc])
    output reg [1:0] ForwardA, // .ForwardA(ForwardA)
    output reg [1:0] ForwardB, // .ForwardB(ForwardB)
    output reg Stall // .Stall(Stall))
);
    always @(*) begin
        // rs1
        if(isInst_ID_EX && RegWrite_ID_EX && (rs1 == rd_ID_EX)) begin
            ForwardA <= 2'b01; // rs1 is fowarded from prior ALU result
        end
        else if(isInst_EX_MEM && RegWrite_EX_MEM && (rs1 == rd_EX_MEM)) begin
            ForwardA <= 2'b10; // rs1 is forwarded from DM or earlier ALU
result
        end
        else begin
            ForwardA <= 2'b00; // rs1 is from RF
        end
        // rs2 is similar to the way of rs1
        if(isInst_ID_EX && RegWrite_ID_EX && (rs2 == rd_ID_EX)) begin
            ForwardB <= 2'b01;
        end
        else if(isInst_EX_MEM && RegWrite_EX_MEM && (rs2 == rd_EX_MEM)) begin
            ForwardB <= 2'b10;
        end
        else begin
            ForwardB <= 2'b00;
        end

        // stall for LD
        // if rs1 or rs2 in ID stage are same as rd in EX stage
        // && the instruction is LD operation in EX stage (MemRead_ID_EX)
        if(isInst_ID_EX && MemRead_ID_EX && (rs1 == rd_ID_EX || rs2 ==
rd_ID_EX)) begin
            Stall <= 1'b1;
        end
        else begin
            Stall <= 1'b0;
        end
    end
endmodule
```

3) Control.v

We omit some modules which are same as before labs.

① Mux4Way

```
module Mux4Way(Src,A,B,C,D,result);
    input wire [1:0] Src; // .Src(ForwardA) or .Src(ForwardB)
    input wire [`WORD_SIZE-1:0] A; // .A(regFile[signal[`rs1+1:`rs1]])
or .A(regFile[signal[`rs2+1:`rs2]])
    input wire [`WORD_SIZE-1:0] B; // .B(ALUResult)
    input wire [`WORD_SIZE-1:0] C; // .C(WriteData)
    input wire [`WORD_SIZE-1:0] D; // This is not used because Src is only 00,
01, 10
    output wire [`WORD_SIZE-1:0] result; // .result(wireA) or .result(wireB)
    // 00 -> A rs1 or rs2 (rs is from RF)
    // 10 -> C WriteData (rs is forwarded from DM or earlier ALU result)
    // 01 -> B ALUResult (rs is fowarded from prior ALU result)
    // 11 -> D
    wire [`WORD_SIZE-1:0] temp1;
    wire [`WORD_SIZE-1:0] temp2;
    assign temp1 = Src[0]? B:A;
    assign temp2 = Src[0]? D:C;
    assign result = Src[1]? temp2:temp1;
endmodule
```

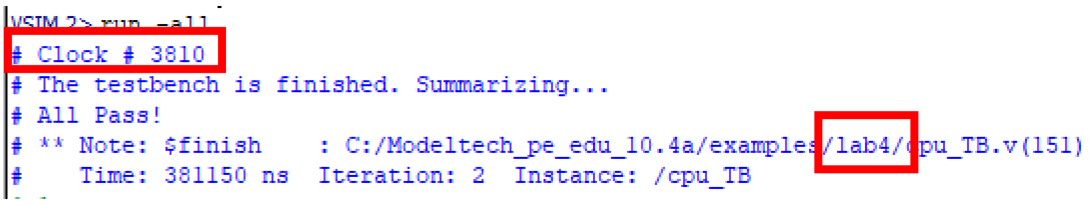
② Cpu

```
module cpu(Clk, Reset_N, readM1, address1, data1, readM2, writeM2, address2,
data2, num_inst, output_port, is_halted);
    input Clk;
    wire Clk;
    input Reset_N;
    wire Reset_N;
    output readM1;
    output [`WORD_SIZE-1:0] address1; // address to load instruction
    output readM2;
    output writeM2;
    output [`WORD_SIZE-1:0] address2; // address to load data
    input [`WORD_SIZE-1:0] data1; // it will be instruction
    wire [`WORD_SIZE-1:0] data1;
    inout [`WORD_SIZE-1:0] data2;//it will be data from memory
    wire [`WORD_SIZE-1:0] data2;
    output [`WORD_SIZE-1:0] num_inst;
    output [`WORD_SIZE-1:0] output_port;
    output is_halted;
    // TODO : Implement your pipelined CPU!(copied from Lab4, so we need to
modify)

    reg [`WORD_SIZE-1:0] num_inst;
    reg [`WORD_SIZE-1:0] output_port;
    wire [`WORD_SIZE-1:0] data;
    reg [`WORD_SIZE-1:0] instruction;
    reg [`WORD_SIZE-1:0] pc;
    reg [`WORD_SIZE-1:0] regFile[`NUM_REGS-1:0];
    reg [`WORD_SIZE-1:0] A;
    reg [`WORD_SIZE-1:0] B;
    reg [`WORD_SIZE-1:0] address1;
    reg [`WORD_SIZE-1:0] address2;
    reg readM1;
    reg readM2;
    reg writeM2;
    reg readData2;
    reg is_halted;

    //IF_ID
    reg isInst_IF_ID;
    reg [`WORD_SIZE-1:0] expPC_IF_ID; // expected next pc(exactly pc+4 because
the predictor is always not-taken)
    //ID_EX
    reg [84:0] reg_EX;
    //EX_MEM
    reg [84:0] reg_MEM;
    //HALT
    reg is_halted_MEM_WB;
    //wires
    wire [`WORD_SIZE-1:0] Imm_next_pc;
    wire [`WORD_SIZE-1:0] Next_pc;
    wire [20:0] signal;
    wire [1:0] ForwardA;
    wire [1:0] ForwardB;
    wire [`WORD_SIZE-1:0] ALUResult;
    wire [`WORD_SIZE-1:0] Immediate;
    wire [`WORD_SIZE-1:0] wireA;
    wire [`WORD_SIZE-1:0] wireB;
    wire Flush;
    wire Stall;
    //tristate of data
    wire data_write;
    wire [`WORD_SIZE-1:0] WriteData;
    assign Imm_next_pc = pc + 16'b1;

    control cont (.instruction(instruction), .signal(signal)); // According to
the instruction, control unit gives proper control signal as before labs
    Immediate_Generator imm
(.instruction(instruction), .Immediate(Immediate)); // According to the
instruction, it generates immediate value
    ALU alu (.A(A), .B(B), .FuncCode(reg_EX[`ALUOp+3:`ALUOp]), .C(ALUResult));
// According to the ALU operation based on the instruction, it calculates
    condition cond (.a(reg_EX[`rs1Data+`WORD_SIZE-
1:`rs1Data]), .b(reg_EX[`rs2Data+`WORD_SIZE-
1:`rs2Data]), .condcode(reg_EX[82:81]),.result(Branch_cond)); // checks
whether the Branch condition is satisfied
    // Below three are mentioned in their own part.
    Mux4Way muxa
(.Src(ForwardA), .A(regFile[signal[`rs1+1:`rs1]]), .B(ALUResult), .C(WriteData
), .D(expPC_IF_ID), .result(wireA));
```

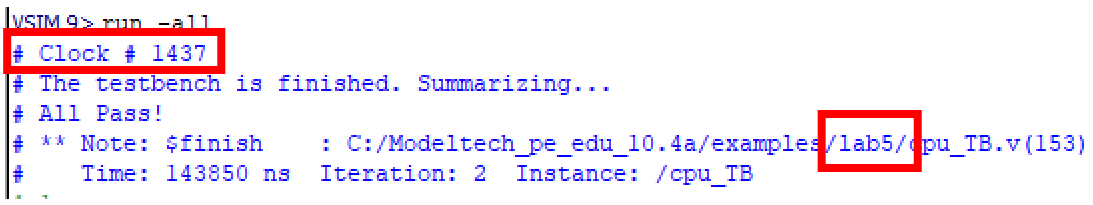
```
    Mux4Way muxb
(.Src(ForwardB), .A(regFile[signal[`rs2+1:`rs2]]), .B(ALUResult), .C(WriteData
), .D(Immediate), .result(wireB));
    Data_Hazard_Unit dhu
(.isInst_ID_EX(reg_EX[`isInst]), .isInst_EX_MEM(reg_MEM[`isInst]), .rs1(signal
[`rs1+1:`rs1]), .rs2(signal[`rs2+1:`rs2]), .rd_ID_EX(reg_EX[`rd+1:`rd]), .rd_E
X_MEM(reg_MEM[`rd+1:`rd]), .MemRead_ID_EX(reg_EX[`MemRead]), .RegWrite_ID_EX(r
eg_EX[`RegWrite]),.RegWrite_EX_MEM(reg_MEM[`RegWrite]), .Jump(signal[`Jump]),
.Branch(signal[`Branch]),.ALUSrc(signal[`ALUSrc]), .ForwardA(ForwardA), .Forwa
rdB(ForwardB), .Stall(Stall));
    // "data2" is "inout" type, so when readData2 is 1, "data2" plays as input
else output.
    assign data2 = (readData2)?  16'bz : reg_MEM[`rs2Data + `WORD_SIZE -
1:`rs2Data];
    // If Load or Store, rt(=rs2) should be taken, else ALUResult is used.
    assign WriteData = (reg_MEM[`MemtoReg])? data2 : reg_MEM[`ALUResult +
`WORD_SIZE -1:`ALUResult];
    // If the instruction in reg_EX is Jump or Branch type(and it satisfies
the branch condition) then flush bit is set.
    assign Flush = reg_EX[`isInst] && (reg_EX[`Jump] || (reg_EX[`Branch] &&
Branch_cond));

    initial begin
        pc <= 16'b0;
        readM1 <= 1'b1;
        readM2 <= 1'b1;
        writeM2 <= 1'b0;
        address1 <= 16'b0;
        num_inst <= 16'b0;
        regFile[0] <= 16'b0;
        regFile[1] <= 16'b0;
        regFile[2] <= 16'b0;
        regFile[3] <= 16'b0;
        readData2 <= 1'b0;
        reg_EX <= 85'b0;
        reg_MEM <=85'b0;
        isInst_IF_ID <= 1'b0;
        is_halted_MEM_WB <= 1'b0;
    end

    always @(posedge Clk) begin
        if(!Reset_N) begin
            pc <= 16'b0;
            readM1 <= 1'b1;
            readM2 <= 1'b1;
            writeM2 <= 1'b0;
            address1 <= 16'b0;
            num_inst <= 16'b0;
            regFile[0] <= 16'b0;
            regFile[1] <= 16'b0;
            regFile[2] <= 16'b0;
            regFile[3] <= 16'b0;
            readData2 <= 1'b0;
            reg_EX <= 85'b0;
            reg_MEM <=85'b0;
            isInst_IF_ID <= 1'b0;
            is_halted_MEM_WB <= 1'b0;
        end
        else begin
            if(!Stall) begin // Just in normal case
                //IF
                // In the case of Flush(the target address of Control
instructions is ALUResult), pc should be target address. Otherwise,
Imm_next_pc = pc + 16'b1;
                pc <= Flush? ALUResult:Imm_next_pc;
                address1 <= Flush? ALUResult:Imm_next_pc;
                //ID
                isInst_IF_ID <= !Flush;
                expPC_IF_ID <= Imm_next_pc;
                instruction <= data1;
                //EXE
                reg_EX <= Flush?
85'b0:{instruction,isInst_IF_ID,expPC_IF_ID,wireB,wireA,signal[19:0]};
                // A and B are used in ALU module. Only for the case of Jump
and Branch, it uses expected PC as first operand in ALU
                A <= (signal[`Jump] && !signal[`JALorJALR]) ||
signal[`Branch]? expPC_IF_ID : wireA;
                // if ALUSrc signal is on, we need to use Immediate value for
ALU calculation
                B <= signal[`ALUSrc]? Immediate : wireB;
            end
            else begin
                reg_EX <= 85'b0; // if stall, do nothing.
            end
            // But the other stages need to keep going even in the case of
stall and flush
            //MEM
            reg_MEM <= {ALUResult,reg_EX[68:0]};
            readM2 <= reg_EX[`isInst]? reg_EX[`MemRead] : 1'b0; // if Load, it
is set to 1
            writeM2 <= reg_EX[`isInst] && reg_EX[`MemWrite]; // if Store, it
is set to 1
            readData2 <= !reg_EX[`MemWrite]; // "data2" is "inout" type, so
when readData2 is 1, "data2" plays as input else output.
            address2 <= ALUResult;
            //WB
            is_halted_MEM_WB <= reg_MEM[`isInst]? reg_MEM[`Halt]:1'b0; // Just
indicate the Halt signal
            if(reg_MEM[`isInst] && reg_MEM[`OpenPort]) begin
                output_port <= reg_MEM[`rs1Data + `WORD_SIZE - 1:`rs1Data];
            end
            if(reg_MEM[`isInst] && reg_MEM[`RegWrite]) begin
                regFile[reg_MEM[`rd+1:`rd]] <= (reg_MEM[`PCtoReg])?
reg_MEM[`expPC+`WORD_SIZE-1:`expPC] : WriteData; // write back the value for
PCtoReg or just a data to RF.
            end
            num_inst <= (reg_MEM[`isInst])? num_inst+1:num_inst;
            //HALT
            is_halted <= is_halted_MEM_WB; // Signal the is_halted after one
cylce later than open_port
        end
    end
endmodule
```

4. Discussion

1) Multi cycle cpu in Lab4 needs 3810 clock cycles while pipelined cpu in this lab needs 1437 clock cycles. Overall pipelined cpu 2.7times faster than muiti cycle cpu. The reasons of that it gets not full improvements like 5X are unavoidable stall and flush caused by load and control instructions respectively. Also, ideal pipelined cpu is not 2.7 times faster because not every instruction consume 5 cycles in the multi cycle cpu but between 3 and 5 cycles.



```
VSIM 2> run -all
# Clock # 3810
# The testbench is finished. Summarizing...
# All Pass!
# ** Note: $finish    : C:/Modeltech_pe_edu_10.4a/examples/lab4/cpu_TB.v(151)
#    Time: 381150 ns  Iteration: 2  Instance: /cpu_TB
```



```
VSIM 9> run -all
# Clock # 1437
# The testbench is finished. Summarizing...
# All Pass!
# ** Note: $finish    : C:/Modeltech_pe_edu_10.4a/examples/lab5/cpu_TB.v(153)
#    Time: 143850 ns  Iteration: 2  Instance: /cpu_TB
```

2) First, we did not know how to "load" in one cycle. It took quite a long time to find out

that the CPU clock is opposite to the memory clock so that after the half of the cycle, we can access the memory and get the value and then after again half of the cycle we can write it to the Register in CPU (so, totally, it takes one cycle).

5. Conclusion

We understand why the pipelined CPU has better throughput than non-pipelined CPU. All the time during the execution, pipelined CPU tries to utilize the all stages so it makes one instruction finish at every cycle. Only for the case of Stall and Flush it delays for few cycles therefore this pipelined CPU is better in performance. Of course because we implement the all required, we totally understand how pipelined CPU works with the data hazard detection, forwarding, stall and flush.