实验项目名称：Pipelined CPU resolving control hazard

一、 实验目的和要求

1. Improve the design of Datapath of 5-stages Pipelined CPU to implement 1-cycle stall when CPU takes Predict-taken policy.
    1) Bring forward calculation of condition & branch address
    2) Bring forward bypass unit
2. Verify the Pipeline CPU with program and observe the execution of program

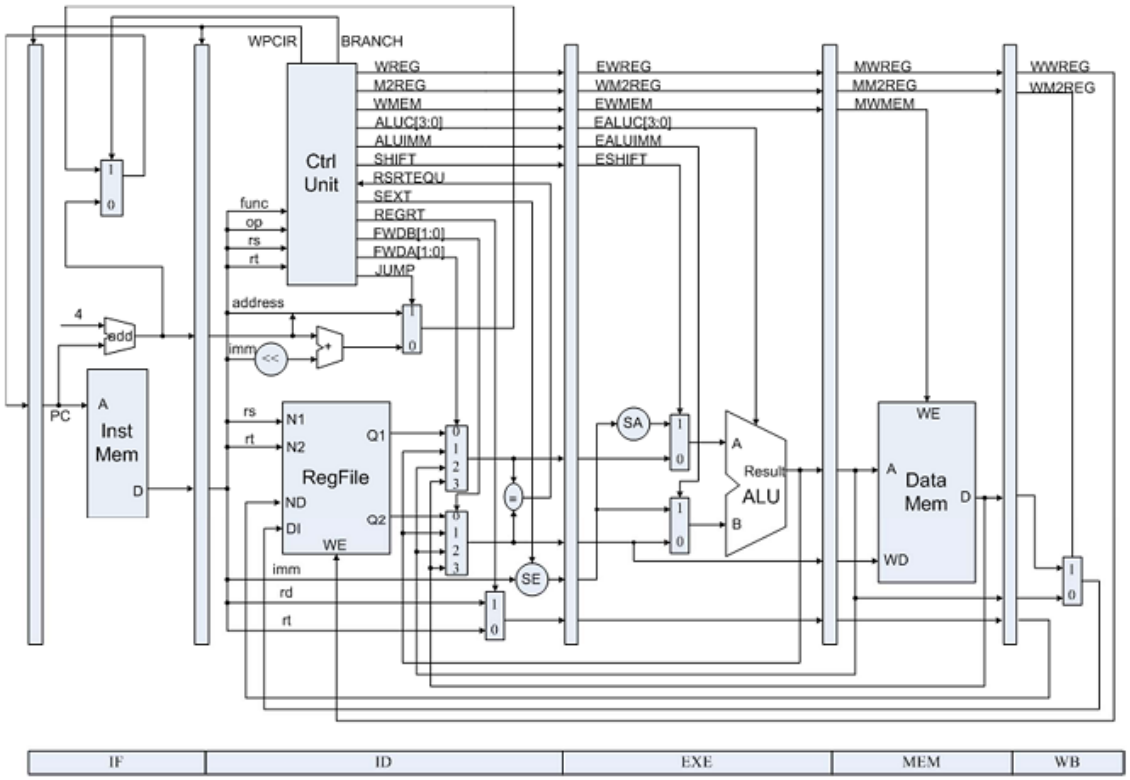
二、 实验内容和原理

1. Control Hazard：
    1) Control hazards arise from the pipelining of branches and other instructions that change the PC. When the branch condition and the branch PC are not available in time to fetch an instruction on the next clock, a control hazard occurs.
    2) Control hazards can cause a greater performance loss for MIPS pipeline than do data hazards.
2. Methods of resolving Control hazards

1) Freeze or flush the pipeline: Holding or deleting any instruction after branch until the branch destination is known.
2) Predict-not-taken: Treat every branch as not taken.
3) Predict-taken: Treat every branch as taken.
    a) Evidence: 60% of branch result is taken
    b) Bring forward calculation of branch condition from MEM Stage to EX Stage, stall reduce from 3-cycle to 2-cycle. Then bring forward from EX to ID, stall reduce from 2-cycle to 1-cycle.
    c) 1-cycle stall may be used for 1 delay slot.
4) Delayed branch: Insert a stall and wait for a cycle to figure out what the right branch address is.

In our code, we use the delayed branch method.

3. Datapath resolving Control Hazards



三、 实验过程和数据记录

The modified code of the ctrl_unit module:

```
module ctrl_unit(clk, rst, if_instr, instr, rsrtequ, cu_branch, cu_jump,
jmp_stall, cu_wreg, cu_m2reg, cu_wmem, cu_aluc, cu_shift, cu_aluimm,
cu_sext, cu_regrt, cu_wpcir, cu_fwda, cu_fwdb);
    //rs-equal-to-rt signal, used as a condition of  branch
    input rsrtequ;
    output cu_jump;       //jump control signal
    output jmp_stall;     //stall signal caused by jump
    wire branch;
    wire AfromMemALU;
    wire BfromMemALU;
    wire AfromMemLW;
    wire BfromMemLW;
    assign AfromEx = (ex_op==`OP_ALUOp) &&(ex_rd==rs) &&(ex_rd!=0);
    assign BfromEx = (ex_op==`OP_ALUOp) &&(ex_rd==rt) &&(ex_rd!=0);
    assign AfromMemALU = (mem_op==`OP_ALUOp) &&(ex_rd!=rs) &&(mem_rd==rs)
&&(mem_rd!=0);
    assign BfromMemALU = (mem_op==`OP_ALUOp) &&(ex_rd!=rt) &&(mem_rd==rt)
&&(mem_rd!=0);
    assign AfromMemLW = (mem_op==`OP_LW) &&(ex_rd!=rs) &&(mem_rd==rs)
&&(mem rd!=0);
    assign BfromMemLW = (mem_op==`OP_LW) &&(ex_rd!=rt) &&(mem_rd==rt)
&&(mem_rd!=0);
    assign AfromExLW = (if_rs==rt) & (if_rs!=0) & (opcode==`OP_LW);
    assign BfromExLW = (if_rt==rt) & (if_rt!=0) & (opcode==`OP_LW);
    //normal stall signal
    assign stall =  AfromExLW || BfromExLW ;
    //stall signal caused by jump
    assign jmp_stall = (opcode == `OP_JMP)? 1: 0;
    assign cu_wpcir = stall;
```

```
    //forwarding signals
    assign cu_fwda[1:0]=(AfromMemLW==1)? 2'b11: ((AfromMemALU==1)? 2'b10:
((AfromEx == 1)? 2'b01: 2'b00));
    assign cu_fwdb[1:0]=(BfromMemLW==1)? 2'b11: ((BfromMemALU==1)? 2'b10:
((BfromEx == 1)? 2'b01: 2'b00));
    //if instruction type == BEQ then 1 else 0
    assign branch = (opcode == `OP_BEQ)? 1: 0;
    //if instruction type == JMP then 1 else 0
    assign cu_jump = (opcode == `OP_JMP)? 1: 0;
    //branch control signal
    assign cu_branch = (branch & rsrtequ) | cu_jump;
    //rd registers depends on the instruction type
    assign rd[4:0] = (opcode==`OP_LW)? instr[20:16]: instr[15:11];
    assign   if_rd[4:0]   =   (if_opcode==`OP_LW)?   if_instr[20:16]:
if_instr[15:11];
    assign    ex_rd[4:0]    =    (ex_op==`OP_LW)?    ex_instr[20:16]:
ex_instr[15:11];
    assign   mem_rd[4:0]   =   (mem_op==`OP_LW)?   mem_instr[20:16]:
mem_instr[15:11];
    assign    wb_rd[4:0]    =    (wb_op==`OP_LW)?    wb_instr[20:16]:
wb_instr[15:11];
```

The modified code of the id_stage module:

```
module id_stage (clk, rst, if_inst, if_pc4, wb_destR, ex_aluR, mem_aluR,
mem_mdata, wb_dest, wb_wreg, cu_wreg, cu_m2reg, cu_wmem, cu_aluc,
cu_shift, cu_aluimm, cu_branch, id_pc4, nid_pc, jmp_stall, id_a_in,
id_b_in, id_imm, cu_regrt, rt, rd, cu_wpcir,id_fwda,id_fwdb, IF_ins_type,
IF_ins_number, ID_ins_type, ID_ins_number, which_reg, reg_content);
    input [31:0] ex_aluR;
    input [31:0] mem_aluR;
    input [31:0] mem_mdata;
    output [31:0] nid_pc;
    output jmp_stall;
    output [31:0] id_a_in;  //ALU A-port input multiplexer in ID stage
    output [31:0] id_b_in;  //ALU B-port input multiplexer in ID stage
    wire [31:0] id_a_in;
    wire [31:0] id_b_in;
    //rs-equal-to-rt signal, used as a condition of branch
    wire rsrtequ;
    //branch or jump target which connects to one port of the PC multiplexer
    wire [31:0] nid_pc;
    wire [31:0] id_branch_pc;   //branch target
    wire [31:0] id_jump_pc;     //jump target
    wire cu_jump;               //jump control signal
    wire jmp_stall;             //stall signal caused by jump
    //add multiplexors for forwading
    assign id_a_in = (id_fwda==2'b11)? mem_mdata: ((id_fwda==2'b10)?
mem_aluR: ((id_fwda==2'b01)? ex_aluR: id_inA));
    assign id_b_in = (id_fwdb==2'b11)? mem_mdata: ((id_fwdb==2'b10)?
mem_aluR: ((id_fwdb==2'b01)? ex_aluR: id_inB));
    assign rsrtequ = (id_a_in - id_b_in) ? 0: 1;
    assign id_branch_pc = pc4 + id_imm;
    assign id_jump_pc = {pc4[31:26], reg_inst[25:0]};
    assign nid_pc =(cu_jump)? id_jump_pc: id_branch_pc;
    always @ (posedge clk or posedge rst)
        if (rst==1)
        begin
            pc4 <= 32'hffffffff;
            ID_ins_type <= 4'b0000;
            ID_ins_number <= 4'b0000;
```

```
        end
        else
        begin
        //when without stalls, renew the following signals
            if(cu_wpcir == 0 && jmp_stall == 0)
            begin
                reg_inst <= if_inst;
                pc4 <= if_pc4;
                ID_ins_type <= IF_ins_type;
                ID_ins_number <= IF_ins_number;
            end
            else              //when there exit normal or jump stalls
            begin
                reg_inst <= 0;
                ID_ins_type <= `INST_TYPE_NONE;
            end
        end
    //pass the rsrtequ signal as an input of ctrl_unit module
    //get the cu_branch, cu_jump, jmp_stall signals from ctrl_unit module
    ctrl_unit  x_ctrl_unit(clk,  rst,  if_inst[31:0],  reg_inst[31:0],
rsrtequ, cu_branch, cu_jump, jmp_stall, cu_wreg, cu_m2reg, cu_wmem,
cu_aluc, cu_shift, cu_aluimm, cu_sext, cu_regrt, cu_wpcir, id_fwda,
id_fwdb);
```

The modified code of the if_stage module:

```
module if_stage (mem_clk, clk, rst, npc, nid_pc, ctrl_branch,
    id_wpcir,
    jmp_stall,
    if_pc, if_pc4, if_inst, IF_ins_type, IF_ins_number, ID_ins_type,
ID_ins_number);
    input id_wpcir;
    input jmp_stall;
    wire id_wpcir;
    wire jmp_stall;
    assign if_pc4 =pc+1;
    assign if_pc = (ctrl_branch==1)? nid_pc: if_pc4;
    always @ (posedge clk) begin
    //renew pc and set run when without normal stalls
        if(id_wpcir == 0) begin
            pc[31:0] <= npc[31:0];
            run <= 1'b1;
        end
        //when there're normal stalls, pc will not be changed
        //note that when it's a jump stall, pc must also be changed
    end
    always @(if_inst) begin
        if (run == 1'b0) begin   //for initial 0
            ID_ins_type[3:0] <= 4'b0000;
            ID_ins_number[3:0] <= 4'b0000;
        end
        else
        begin
            //when there're no stalls, renew ID_ins_number and ID_ins_type
            if(id_wpcir == 0) begin
                ID_ins_number[3:0] <= pc[3:0];
                //omit some codes here
            end
            else if(jmp_stall == 1) begin   //with jump stall
                ID_ins_number[3:0] <= pc[3:0];
                ID_ins_type <= `INST_TYPE_NONE;
            end
```

```
            else begin                          //with normal stall
                ID_ins_type <= `INST_TYPE_NONE;
            end
        end
    end
```

The modified code of the ex_stage module:

```
module ex_stage (clk, id_imm, id_a_in, id_b_in, id_wreg, id_m2reg,
id_wmem, id_aluc, id_aluimm, id_shift, id_branch, id_pc4, id_regrt, id_rt,
id_rd,   ex_wreg,   ex_m2reg,  ex_wmem,  ex_aluR,  ex_inB,  ex_destR,
ex_branch, ex_pc, ex_zero, ID_ins_type, ID_ins_number, EXE_ins_type,
EXE_ins_number );
    input [31:0] id_a_in;
    input [31:0] id_b_in;
    assign a_in = eshift? sa: edata_a;            //ALU A-port input
    assign b_in = ealuimm? odata_imm: edata_b;    //ALU B-port input
    //latch id a in, id b in as edata a,edata b and forward control
signals no longer require to latch
    Reg_ID_EXE x_Reg_ID_EXE(clk, id_wreg, id_m2reg, id_wmem, id_aluc,
id_shift, id_aluimm, id_a_in, id_b_in, id_imm, id_branch, id_pc4,
id_regrt, id_rt, id_rd, ex_wreg, ex_m2reg, ex_wmem, ealuc, eshift,
ealuimm, edata_a, edata_b, odata_imm, ex_branch, epc4,e_regrt,e_rt,e_rd,
ID_ins_type, ID_ins_number, EXE_ins_type, EXE_ins_number);
```

The modified code of the top module:

```
    wire [31:0] nid_pc;
    wire jmp_stall;
    wire [31:0] id_a_in;
    wire [31:0] id_b_in;

    if_stage x_if_stage(clk0, BTN3, rst, pc,
        nid_pc,       //added signal
        id_branch, id_wpcir,
        jmp_stall,    //added signal
        if_npc, if_pc4, if_inst, IF_ins_type, IF_ins_number, ID_ins_type,
ID_ins_number);

    id_stage x_id_stage(BTN3, rst, if_inst, if_pc4, wb_destR,
        //added signals
        ex_aluR, mem_aluR, mem_mdata,
        wb_dest,wb_wreg,
        id_wreg,  id_m2reg,  id_wmem,  id_aluc,  id_shift,  id_aluimm,
id_branch, id_pc4,
        //added signals
        nid_pc, jmp_stall,
        id_a_in, id_b_in, id_imm, id_regrt,id_rt,id_rd,
        //stall and forwarding signals
        id_wpcir, id_fwda, id_fwdb,
        ID_ins_type,   ID_ins_number,   EX_ins_type,   EX_ins_number,
{1'b0,which_reg}, reg_content);

    ex_stage x_ex_stage(BTN3, id_imm,
        //added signals
        id_a_in, id_b_in,
        id_wreg,  id_m2reg,  id_wmem,  id_aluc,  id_aluimm,id_shift,
id_branch, id_pc4, id_regrt, id_rt, id_rd, ex_wreg, ex_m2reg, ex_wmem,
ex_aluR, ex_inB, ex_destR, ex_branch, ex_pc, ex_zero, EX_ins_type,
EX_ins_number, MEM_ins_type, MEM_ins_number);

    mem_stage x_mem_stage(BTN3, ex_destR, ex_inB, ex_aluR, ex_wreg,
ex_m2reg, ex_wmem, ex_branch, ex_pc, ex_zero, mem_wreg, mem_m2reg,
```

```
mem_mdata, mem_aluR, mem_destR, mem_branch, mem_pc, MEM_ins_type,
MEM_ins_number, WB_ins_type, WB_ins_number);

    wb_stage x_wb_stage(BTN3, mem_destR, mem_aluR, mem_mdata, mem_wreg,
mem_m2reg,      wb_wreg,      wb_dest,      wb_destR,      WB_ins_type,
WB_ins_number,OUT_ins_type, OUT_ins_number);
```

四、 实验结果分析

Program for verification is as follows:

MIPS code	Bin data	Address	Inst. Type
lw r1, $20(r0)	0x8c01_0014	0	6
lw r2, $21(r0)	0x8c02_0015	1	6
add r3, r1, r2	0x0022_1820	2	1
add r2,r0,r0	0x0000_1020	3	1
sub r4, r1, r3	0x0023_2022	4	2
and r5, r3, r4	0x0064_2824	5	3
or r7, r5, r7	0x00a7_3825	6	4
beq r5,r7,-6	0x10a7_fffa	7	8
add r7,r0,r1	0x0001_3820	8	1
jmp 0	0x0800_0000	9	9
add r7, r0, r0	0x0000_3820	A	1

Data Mem[20]=1

Data Mem[21]=4

The result of execution is shown as follows:

Clock Count	Instruction Code	Instruction	“stage name” /number/type	Comment
00	01234567		f01d01e01m01w01	The first instruction “lw r1, $20(r0)” enters the IF stage.
01	8c010014	lw r1, $20(r0)	f1fd06e00m00w00	The first instruction “lw r1, $20(r0)” enters the ID stage. And a new instruction “lw r2,

				$21(r0)" enters the IF stage.
02	8c020015	lw r2, $21(r0)	f2fd16e06m00w00	The first instruction "lw r1, $20(r0)" enters the EX stage. And a new instruction " add r3, r1, r2" enters the IF stage.
03	00221820	add r3, r1, r2	f3fd2fe16m06w00	The first instruction "lw r1, $20(r0)" enters the MEM stage. And the instruction " add r3, r1, r2" enters the ID stage and find that the RAW data dependency with the first instruction "lw r1, $20(r0)" on r1 is resolved by forwarding. However, there still exits RAW data dependency with the instruction "lw r2, $21(r0)" on r2. And the data hazard can't be resolved by forwarding, so a stall is inserted.
04	00221820	add r3, r1, r2	f3fd21e1fm16w06	The first instruction

				“lw r1, $20(r0)” enters the WB stage. Now the instruction “add r3, r1, r2” can actually enter the ID stage.
05	00001020	add r2,r0,r0	f4fd31e21m1fw16	The first instruction “lw r1, $20(r0)” completes the WB stage, so the content of r1 becomes 1. In addition, the instruction “lw r2, $21(r0)” completes the MEM stage, so the content of r2 is forwarding to “add r3, r1, r2”.
06	00232022	sub r4, r1, r3	f5fd42e31m21w1f	The instruction “lw r2, $21(r0)” completes the WB stage, so the content of r2 becomes 4. In addition, the instruction “add r3, r1, r2” completes the EX stage, so the content of r3 is forwarding to “sub r4, r1, r3”.
07	00642824	and r5, r3, r4	f6fd53e42m31w21	The inserting stall completes the WB

				stage. In addition, the instruction “add r3, r1, r2” completes the MEM stage, so the content of r3 is forwarding to “and r5, r3, r4”. In addition, the content of r4 can be forwarding to “and r5, r3, r4” when the instruction “sub r4, r1, r3” completes the EX stage. Because the forward bypass unit is moved to ID Stage, this forwarding operation will be performed right after “sub r4, r1, r3” completes the EX stage and “and r5, r3, r4” finishes its ID stage. When “and r5, r3, r4” enters the EX stage, the content of r4 is already forwarded.
08	00a73825	or r7, r5, r7	f7fd64e53m42w31	The instruction “add r3, r1, r2” completes

				the WB stage, so the content of r3 becomes 5. In addition, the content of r5 is forwarding to “nor r6, r4, r5” right after it finishes the ID stage and the instruction “and r5, r3, r4” completes the EX stage.
09	10a7fffa	beq r5,r7,-6	f8fd78e64m53w42	The instruction “add r2, r0, r0” completes the WB stage, so the content of r2 becomes 0. In addition, the content of r7 is forwarding to “beq r5, r7, -6” right after it finishes the ID stage and the instruction “or r7, r5, r7” completes the EX stage. Similarly, the content of r5 is forwarding to “beq r5, r7, -6” right after it finishes the ID stage since the instruction

				“and r5, r3, r4” has already completed the EX stage.
0a	00013820	add r7,r0,r1	f2fd81e78m64w53	The instruction “sub r4, r1, r3” completes the WB stage, so the content of r4 becomes 0x ffff fffc. In addition, the instruction “beq r5, r7, -6” completes the ID stage, so the branch target (instruction whose PC equals to 2 because r5 is equal to r7) enters the IF stage. The instruction in the ID stage “add r7,r0,r1” is the branch delay slot instruction.
0b	00221820	add r3, r1, r2	f3fd21e81m78w64	The instruction “and r5, r3, r4” completes the WB stage, so the content of r5 becomes 4.
0c	00001020	add r2,r0,r0	f4fd31e21m81w78	The instruction “or r7, r5, r7” completes the WB stage, so the content of r7 becomes 4.

0d	00232022	sub r4, r1, r3	f5fd42e31m21w81	The instruction “beq r5, r7, -6” completes the WB stage.
0e	00642824	and r5, r3, r4	f6fd52e42m31w21	The branch delay slot instruction “add r7, r0, r1” completes the WB stage, so the content of r7 is modified to 1.
0f	00a73825	or r7, r5, r7	f7fd64e53m42w31	The instruction “add r3, r1, r2” completes the WB stage. Note that the content of r2 has been modified to 0, so the content of r3 becomes 1.
10	10a7fffa	beq r5,r7,-6	f8fd78e64m53w42	The instruction “add r2,r0,r0” completes the WB stage, so the content of r2 becomes 0.
11	00013820	add r7,r0,r1	f9fd81e78m64w53	The instruction “sub r4, r1, r3” completes the WB stage, so the content of r4 becomes 0. In addition, the instruction “beq r5, r7, -6” completes the ID stage and branch is not taken.

12	08000000	jmp 0	fafd99e81m78w64	The instruction “and r5, r3, r4” completes the WB stage, so the content of r5 becomes 0.
13	00003820	add r7, r0, r0	f0fda1e99m81w78	The instruction “or r7, r5, r7” completes the WB stage, so the content of r7 becomes 1. In addition, the instruction “jmp 0” completes the ID stage, so the jump target enters the IF stage. There’s a stall after the jump instruction.
14	8c010014	lw r1, $20(r0)	f1fd06e9fm99w81	The instruction “beq r5, r7, -6” completes the WB stage.
15	8c020015	lw r2, $21(r0)	f2fd16e06m9fw99	The instruction “add r7, r0, r1” completes the WB stage.
16	00221820	add r3, r1, r2	f3fd2fe16m06w9f	The instruction “jmp 0” completes the WB stage.
17	00221820	lw r1, $20(r0)	f3fd21e1fm16w06	The stall after the instruction “jmp 0”

				completes the WB stage.

The result indicates that the code implements the required functions correctly.

五、 讨论与心得

A major puzzle we've met is the confusing problem of jump instructions.

We combine the stall from jump instruction with the former stall signal "id_wpcir". And our previous result of "jmp 0" is f0fda1e99m81w78, which means the instruction in IF stage is the correct jump target, but next it becomes fbf, which means the instruction in IF stage is rewritten with a wrong instruction.

Therefore, we make an analysis of the flow of how the PC changes:

Firstly, the jump target is assigned to "if_pc" in IF stage module:

```
assign if_pc4 = pc+1;
assign if_pc = (ctrl_branch==1)? nid_pc: if_pc4;
```

Then this signal connects to the "if_npc" signal in top module:

```
assign pc [31:0] = if_npc[31:0];
```

Then "pc" is used as an input signal "npc" of IF stage module. In the following codes, "pc" will not be renewed to "npc" if there're no stalls:

```
always @ (posedge clk) begin
    if(id_wpcir == 0) begin //renew pc and set run when without stalls
        pc[31:0] <= npc[31:0];
        run <= 1'b1;
    end
    //when there're stalls, pc will not be changed
end
```

Here comes the reason of the previous ridiculous result: although the "pc" signal in top module ("npc" of IF stage module) is assigned to the jump target, the "pc" signal in IF stage module still remains the previous state. Then "if_pc4" signal will be refreshed according to it, and "if_pc" will be assigned to a wrong "if_pc4" value. What's worse, ID_ins_number and ID_ins_type are decided in IF stage module, so the related signals will be incorrect when they pass into the module of next stage.

The solution is to give special treatment to stall signals that are caused by jump instruction. When the jump stall occurs, the "pc" signal in IF stage module must also be renewed. Methods to check the type of stalls are not unique. Our approach is to use a separated signal "jmp_stall" to mark this condition. And we finally solved this puzzle.