浙江大学

本科实验报告

课程名称：计算机体系结构

姓　　名：蔡武威

学　　院：计算机科学与技术学院

　　系：计算机科学与技术系

专　　业：计算机科学与技术

学　　号：3130101796

指导教师：姜晓红

2016年　　6 月　　6 日

浙江大学实验报告

课程名称：计算机体系结构 实验类型：综合

实验项目名称：Lab3：Implementing “forwarding paths” and “predict_not-taken”

学生姓名：蔡武威 专业：计算机科学与技术 学号：3130101796

同组学生姓名：None 指导老师：姜晓红

实验地点：曹二期 301 实验日期：2016 年 6 月 4 日

一、 实验目的和要求

1. Experiment Purpose
   - Understand the principles of Pipelined CPU Bypass Unit.
   - Master the method of Pipelined pipeline Forwarding Detection and Pipeline Forwards.
   - Master the Condition In Which Pipeline Forwards.
   - Master the Condition In Which Bypass Unit doesn’t Work and the Pipeline stalls
   - Master methods of program verification of Pipelined CPU with forwarding
2. Experiment Task and Requirements
   - Design the Bypass Unit of Datapath of 5-stages Pipelined CPU
   - Modify the CPU Controller
     - Conditions in Which Pipeline Forwards
     - Conditions in Which Pipeline Stalls
   - Verify the Pipeline CPU with program and observe the execution of program

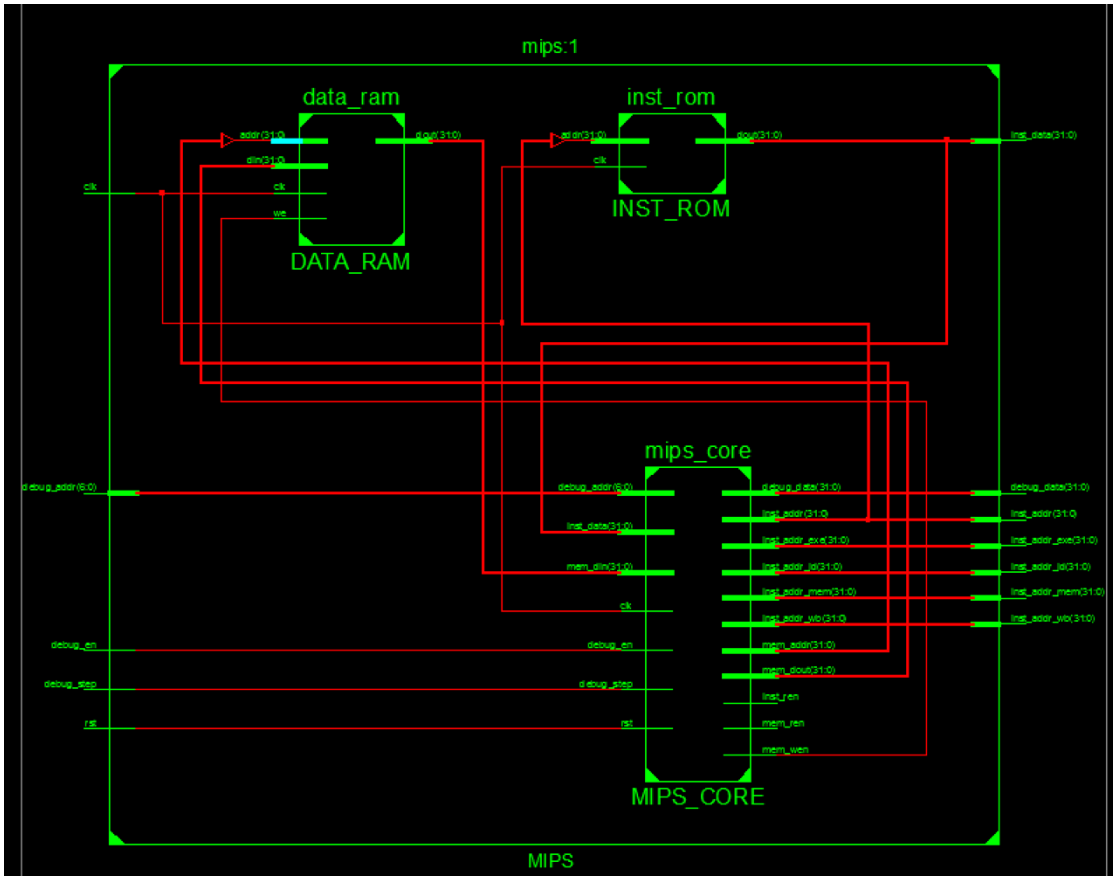
二、 实验内容和原理

- The Verilog code can be divided into the following modules:

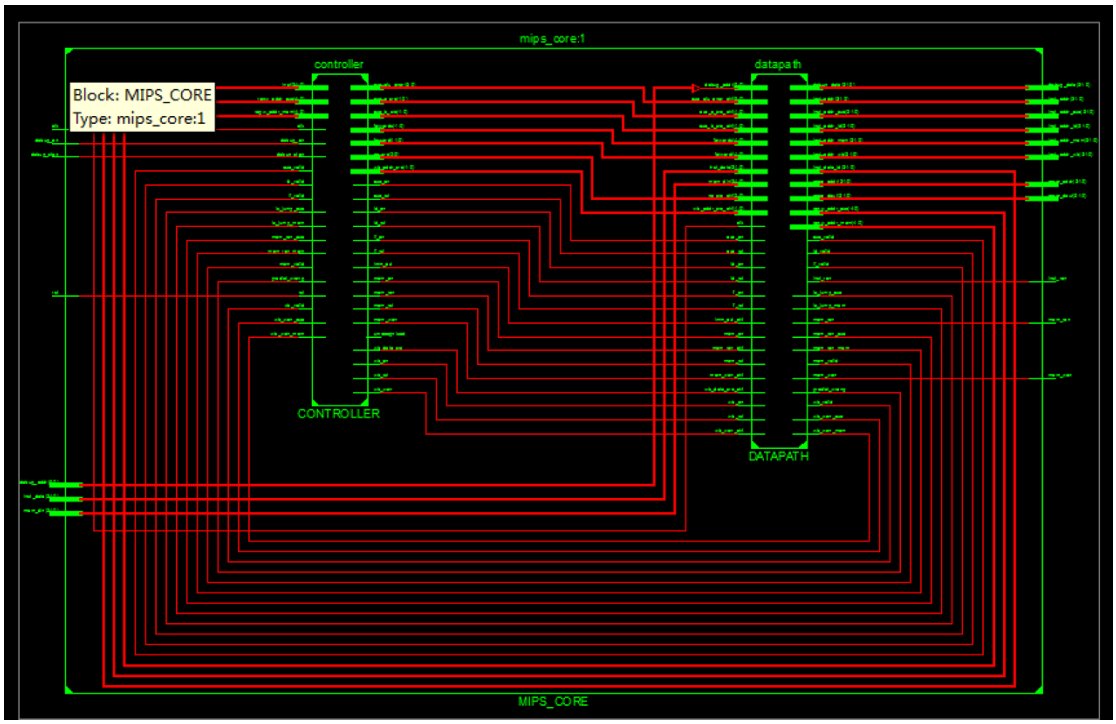
1. mips_top: The top module of the whole project.
2. clk_gen: To generate different clocks.
3. anti_jitter: To avoid the jitter of the button.
4. display: To display some information, including instruction address of which is IF\ID\EXE\MEM\WB stage, register values and some other important information.
5. mips: The main module to implement a MIPS CPU, which has also some sub-module, such as mips_core, inst_rom, data_ram. Also, the mips_core module has sub-module named controller and datapath. The sub-module “controller” is a sub-module to generate control signals to make pipeline work and the sub-module “datapath” is the datapath of pipeline which also has sub-modules named “regfile” and “alu”.

- The circuit graph of important module is shown as following:

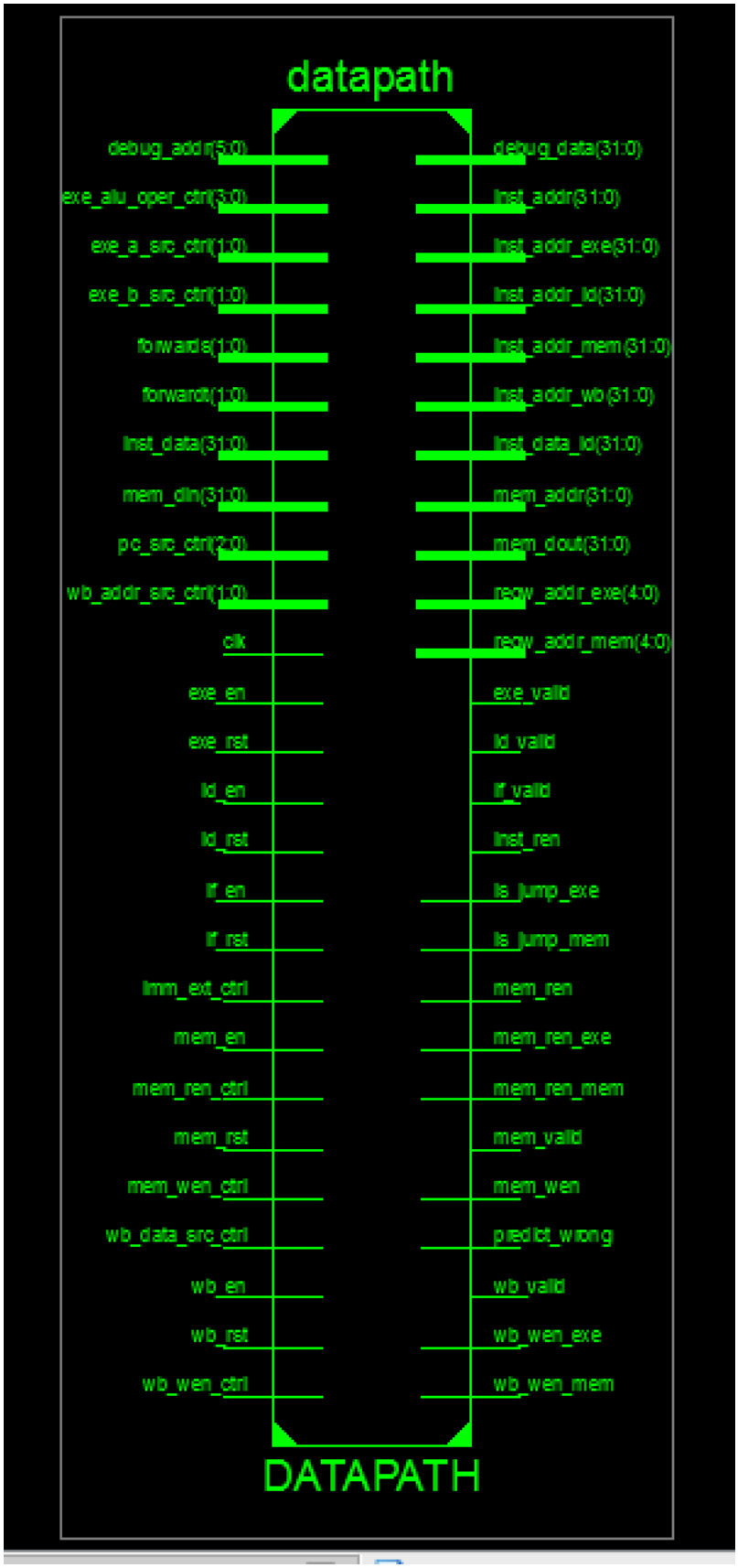
1. mips:



2. mips_core


mips_core:1
controller
datapath
Block: MIPS_CORE
Type: mips_core:1
CONTROLLER
DATAPATH
MIPS_CORE

3. datapath



- Control signals:

Here we define the following control signals:

1. For PC source(which is the next PC):

   PC_NEXT: no jump and branch, the next PC is just PC+4

   PC_JUMP: used when the instruction is j or jal

   PC_JR: used when the instruction is jr

   PC_BEQ: used when the instruction is beq

   PC_BNE: used when the instruction is bne
2. For ALU A source(the input a of ALU):

   A_RS: come from rs of the instruction

   A_SA: come from sa of the instruction

   A_LINK: used when the instruction is jal

   A_BRANCH: used when the instruction is beq or bne
3. For ALU B source(the input b of ALU):

   B_RT: come from rt of the instruction

   B_IMM: come from imm of the instruction

   B_LINK: used when the instruction is jal

   B_BRANCH: used when the instruction is beq or bne
4. For ALU operation: ALU_ADD, ALU_SUB, ALU_SLT, ALU_LUI, ALU_AND, ALU_OR, ALU_SR
5. For WB(write back) address source

   WB_ADDR_RD: to rd of the instruction

   WB_ADDR_RT: to rt of the instruction

   WB_ADDR_LINK: used when the instruction is jal
6. For WB data source

   WB_DATA_ALU: from ALU result

   WB_ADDR_MEM: from memory
7. Imm_ext: Imm_ext=1 when the immediate of the instruction should be sign-extension
8. Mem_ren: memory read enable
9. Mem_wen: memory write enable

10. WB_wen:write back enable

11. Rs_used: whether rs is used in this instruction

12. Rt_used: whether rt is used in this instruction

The following is the table which describes the control signal of each instruction:

Instruction	add	sub	and	or	slt	srl
PC source	PC_NEXT	PC_NEXT	PC_NEXT	PC_NEXT	PC_NEXT	PC_NEXT
ALU A source	A_RS	A_RS	A_RS	A_RS	A_RS	A_RS
ALU B source	B_RT	B_RT	B_RT	B_RT	B_RT	B_RT
ALU operation	ALU_ADD	ALU_SUB	ALU_AND	ALU_OR	ALU_SLT	ALU_SR
WB address source	WB_ADDR_RD	WB_ADDR_RD	WB_ADDR_RD	WB_ADDR_RD	WB_ADDR_RD	WB_ADDR_RD
WB data source	WB_DATA_ALU	WB_DATA_ALU	WB_DATA_ALU	WB_DATA_ALU	WB_DATA_ALU	WB_DATA_ALU
Imm_ext	0	0	0	0	0	0
Mem_ren	0	0	0	0	0	0
Mem_wen	0	0	0	0	0	0
WB_wen	1	1	1	1	1	1
Rs_used	1	1	1	1	1	0
Rt_used	1	1	1	1	1	1

Instruct	jr	j	jal	beq	bne	addi

ion						
PC source	PC_JR	PC_J	PC_J	PC_BEQ	PC_BNE	PC_NEXT
ALU A source	/	/	A_LINK	A_BRANCH	A_ BRANCH	A_RS
ALU B source	/	/	A_LINK	B_ BRANCH	B_ BRANCH	B_IMM
ALU operation	/	/	ALU_ADD	ALU_SUB	ALU_SUB	ALU_ADD
WB address source	/	/	WB_ADDR_ LINK	WB_ADDR_ RD	WB_ADDR_ RD	WB_ADDR_ RT
WB data source	/	/	WB_DATA_ ALU	WB_DATA_ ALU	WB_DATA_ ALU	WB_DATA_ ALU
Imm_ext	0	0	0	1	1	1
Mem_ren	0	0	0	0	0	0
Mem_wen	0	0	0	0	0	0
WB_wen	0	0	1	0	0	1
Rs_used	1	0	0	1	1	1
Rt_used	0	0	0	1	1	0

Instruct ion	andi	ori	slti	Lw	sw	lui
PC source	PC_NEXT	PC_NEXT	PC_NEXT	PC_NEXT	PC_NEXT	PC_NEXT
ALU A source	A_RS	A_RS	A_RS	A_RS	A_RS	A_RS
ALU B	B_IMM	B_IMM	B_IMM	B_IMM	B_IMM	B_IMM

source						
ALU operation	ALU_AND	ALU_OR	ALU_SLT	ALU_ADD	ALU_ADD	ALU_LUI
WB address source	WB_ADDR_RT	WB_ADDR_RT	WB_ADDR_RT	WB_ADDR_RT	/	WB_ADDR_RT
WB data source	WB_DATA_ALU	WB_DATA_ALU	WB_DATA_ALU	WB_DATA_MEM	/	WB_DATA_ALU
Imm_ext	0	0	1	1	1	1
Mem_ren	0	0	0	1	0	0
Mem_wen	0	0	0	0	1	0
WB_wen	1	1	1	1	0	1
Rs_used	1	1	1	1	1	0
Rt_used	0	0	0	0	1	1

Here are signals to control forward:

1. Forwards: If forwards = 0, forwards doesn't work; if forwards = 1, the output of alu in EXE stage is forwarded to the data of rs; if forwards = 2, the output of alu in MEM stage is forwarded to the data of rs; if forwards = 3, the data got from memory is forwarded to the data of rs.
2. Forwardt: If forwardt = 0, forwardt doesn't work; if forwardt = 1, the output of alu in EXE stage is forwarded to the data of rt; if forwardt = 2, the output of alu in MEM stage is forwarded to the data of rt; if forwardt = 3, the data got from memory is forwarded to the data of rt.
3. Reg_stall: if reg_stall = 1, the forward is not taken and a stall is implemented to deal with data hazard.

- Key code segment

1. Pipeline control part in controller module

```
// pipeline control
    reg reg_stall;
```

```
    reg branch_stall;
    wire [4:0] addr_rs, addr_rt;

    assign
        addr_rs = inst[25:21],
        addr_rt = inst[20:16];

    always @(*) begin
        reg_stall = 0;
        forwards = 0;
        forwardt = 0;
        if (rs_used && addr_rs != 0) begin
            if (regw_addr_exe == addr_rs && wb_wen_exe)
begin
                if (mem_ren_exe == 1) begin
                    reg_stall = 1;
                end
                else
                    forwards = 1;
            end
            else   if   (regw_addr_mem   ==   addr_rs   &&
wb_wen_mem) begin
                if (mem_ren_mem == 0)
                    forwards = 2;
                else
                    forwards = 3;
            end
        end
        if (rt_used && addr_rt != 0) begin
            if (regw_addr_exe == addr_rt && wb_wen_exe)
begin
                if (mem_ren_exe == 1) begin
                    if (is_save == 1)
                        forwardt = 3;
                    else
                        reg_stall = 1;
                end
                else
                    forwardt = 1;
            end
            else   if   (regw_addr_mem   ==   addr_rt   &&
wb_wen_mem) begin
                if (mem_ren_mem == 0)
                    forwardt = 2;
```

```
            else
                forwardt = 3;
        end
    end

    always @(*) begin
        branch_stall = 0;
        if (pc_src == PC_JUMP || pc_src == PC_JR ||
is_jump_exe || is_jump_mem)
            branch_stall = 1;
    end

    `ifdef DEBUG
    reg debug_step_prev;

    always @(posedge clk) begin
        debug_step_prev <= debug_step;
    end
    `endif

    always @(*) begin
        if_rst = 0;
        if_en = 1;
        id_rst = 0;
        id_en = 1;
        exe_rst = 0;
        exe_en = 1;
        mem_rst = 0;
        mem_en = 1;
        wb_rst = 0;
        wb_en = 1;
        if (rst) begin
            if_rst = 1;
            id_rst = 1;
            exe_rst = 1;
            mem_rst = 1;
            wb_rst = 1;
        end
        `ifdef DEBUG
        // suspend and step execution
        else if ((debug_en) && ~(~debug_step_prev &&
debug_step)) begin
            if_en = 0;
```

```
            id_en = 0;
            exe_en = 0;
            mem_en = 0;
            wb_en = 0;
        end
        `endif
        // this stall indicate that ID is waiting for
previous instruction, should insert NOPs between ID and
EXE.
        else if (reg_stall) begin
            if_en = 0;
            id_en = 0;
            exe_rst = 1;
        end
        //  this  stall  indicate  that  a  jump/branch
instruction is running, so that 3 NOP should be inserted
between IF and ID
        else if (branch_stall) begin
            id_rst = 1;
        end
        else if (predict_wrong) begin
            id_rst = 1;
            exe_rst = 1;
            mem_rst = 1;
        end
    end
```

2. Code segment to control forward in datapath module

```
          if (forwards == 3)
              data_rs_exe <= mem_din;
           else if (forwards == 2)
              data_rs_exe <= alu_out_mem;
           else if (forwards == 1)
              data_rs_exe <= alu_out_exe;
           else
              data_rs_exe <= data_rs;
           if (forwardt == 3)
              data_rt_exe <= mem_din;
           else if (forwardt == 2)
              data_rt_exe <= alu_out_mem;
           else if (forwardt == 1)
              data_rt_exe <= alu_out_exe;
           else
              data_rt_exe <= data_rt;
```

3. Code segment to implement predict-not-taken in datapath module

```
       case (pc_src_mem)
          PC_JUMP:         branch_target_mem         <=
{inst_addr_exe[31:28],inst_data_mem[25:0],2'b00};
          PC_JR: branch_target_mem <= data_rs_mem;
          PC_BEQ:          branch_target_mem         <=
rs_rt_equal_mem? alu_out_mem : inst_addr_next;
          PC_BNE:          branch_target_mem         <=
rs_rt_equal_mem? inst_addr_next : alu_out_mem;
          default:         branch_target_mem         <=
inst_addr_next_mem;  // will never used
       endcase
```

三、 实验过程和数据记录及结果分析

1. Input data sets and output results

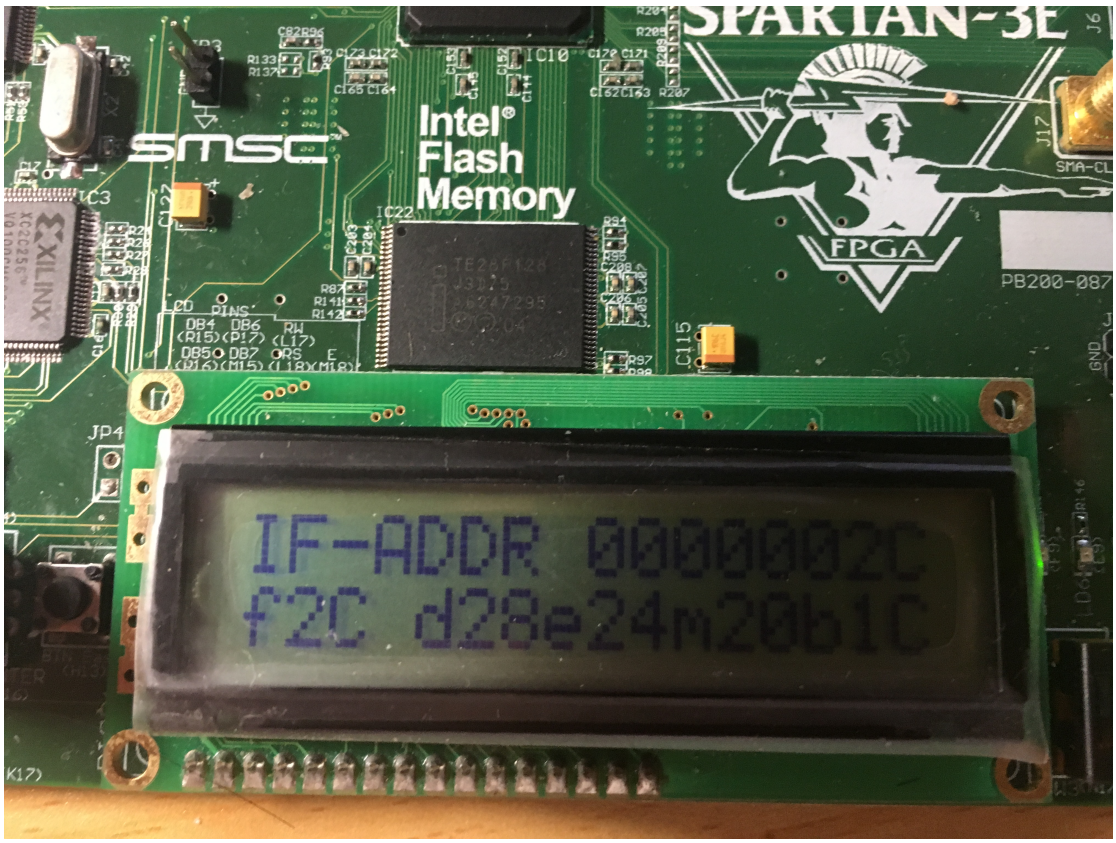
Line number	Assembly code	Machine code	Result
0	Lw r1, 20(r0)	8c010014	R1=4
1	Lw r2, 24(r0)	8c020018	R2=1
2	Add r3,r2,r1	00221820	R3=5; stall , forwarding
3	Sub r4,r3,r1	00612022	R4=1
4	And r5,r3,r1	00232824	R5=4
5	Or r6,r3,r1	00223025	R6=5
6	Addi r6,r3,4	20660004	R6=9
7	Add r7,r0,r1	00203820	R7=4
8	Lw r8,0(r7)	8ce80000	R8=8
9	Sw r8,0(r7)	Ace80008	Forwarding
10	Lw r9,8(r7)	8ce90008	R9=8
11	Sw r7,0(r9)	Ad270000	Stall, forwarding
12	Lw r10,0(r9)	8d2a0000	R10=4
13	Add r10, r1,r1	00215020	R10=8
14	Add r11, r2,r2	00425820	R11=2
15	Add r10,r1,r2	00415020	R10=5

16	Beq r10,r11,8	10220008	Not taken, stall
17	Lw r1,8(r7)	8ce10008	R1=8
18	Lw r2,24(r0)	8c020018	R2=1
19	Add r3,r2,r1	00221820	R3=9, stall,forwarding
20	Sub r4,r3,r1	00612022	R4=1
21	Addi r20,r4,1	20940001	R20=2,forwarding
22	Ori r20, r4, 1	34940001	R20=1
23	Lw r2,0(r7)	8ce20000	R2=8
24	Bne r2,r20,4	14540004	Taken, stall, forwarding
25	Lw r1,20(r0)	8c010014	Not execute
26	Lw r2,24(r0)	8c020018	Not execute
27	Add r3,r2,r1	00221820	Not execute
28	Sub r4,r3,r1	00412022	Not execute
29	J 0	08000000	PC =0

2. Screen shots and pictures of the running result on Spartan3E

- The first line of the screen display some important information, including register values and instruction data or address of instruction in IF\ID\EX\MEM\WB stages; the second line of the screen display the instruction address in IF\ID\EX\MEM\WB stages. The following image shows the stage that the third instruction is ID stage, which has a LW-ALU hazard, and there is a stall implemented before forwarding.

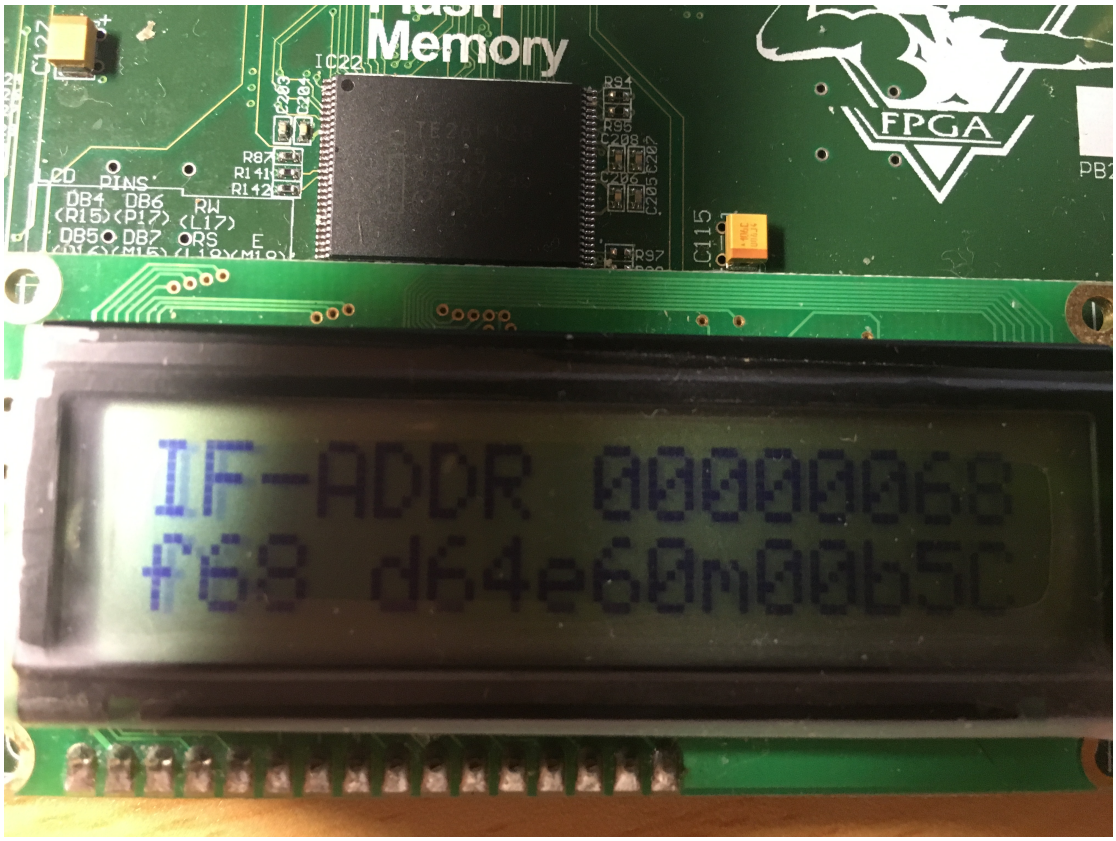
- The following image shows the stage that the instruction “Lw r8,0(r7)” after “add r7,r0,r1” , which is “ALU-LW” hazard, is using forwarding and there is not a stall. What’s more, it also show that there is a LW-SW hazard between instructions at address 20 and 24, which are “lw, r8,0(r7)” and “sw r8,0(r7)” accordingly, but stall is not needed and a forwarding can solve it.



- The following image shows the stage that there is a LW-SW hazard between instruction at address 28 and 2c, which are “lw r9, 8(r7) ” and “sw r7,0(r9)” accordingly. Unlike the last stage, these LW-SW hazard needs a stall and a forwarding to deal with.

- The instruction at address 40 is “beq r10, r11, 8” and the following image shows how predict-not-taken work when the branch is actually not taken.

- The instruction at address 60 is “bne r2,r20,4” and the following images show how predict-not-taken work when the branch is actually taken.



3. Steps to do this experiment:

- Review and extend the key code in lab2.

- Use new display module to display data to satisfy the new requirement.
- Design the testing code of MIPS.
- Compile and download the bit file to the board.
- Execute the testing code and debug.

四、 讨论与心得

In this lab, I keenly feel that it's much more difficult than any other lab about Verilog I ever do. As a result, it cost me much time to implement the code and even more time to debug. Thanks to it, however, it's obvious that I master the acknowledge about forwarding and predict-not-taken much better than before, especially that I know better about when forwarding works and when forwarding doesn't work.