计算机系统结构实验报告

Lab06

类 MIPS 多周期处理器的设计与实现

姓　　名：　徐薪

学　　号：　519021910726

日　　期：　2021 年 6 月 09 日

摘 要

本实验要求实现类 MIPS 多周期处理器。该类 MIPS 多周期处理器的 CPU 支持 9 条 MIPS 指令（包括 R 型指令中的 add、sub、and、or、slt；I 型指令中的 lw、sw、beq；J 型指令中的 j）。本实验的设计基础为实验五中已完成的类 MIPS 单周期处理器。最后，本实验通过软件仿真的方式进行结果验证。

目录

1. 实验概述

1.1 实验内容

本实验要求实现类 MIPS 多周期处理器。该类 MIPS 多周期处理器的 CPU 支持 9 条 MIPS 指令（包括 R 型指令中的 add、sub、and、or、slt；I 型指令中的 lw、sw、beq；J 型指令中的 j）。本实验的设计基础为实验五中已完成的类 MIPS 单周期处理器。最后，本实验通过软件仿真的方式进行结果验证。

1.2 实验目的

（1）理解类 MIPS 多周期处理器的工作原理；

（2）实现类 MIPS 多周期处理器的各个功能模块以及顶层模块；

（3）实现类 MIPS 多周期处理器每个阶段的段寄存器；

（4）使用功能仿真对实验结果进行验证。

2. 原理分析

2.1 各模块原理分析

2.1.1 主控制器 Ctr 模块

2.1.1.1 Ctr 模块描述

主控制器单元（Ctr）的输入为 MIPS 指令的 opCode 字段。Ctr 的作用是将操作码译码，并向运算单元控制器（ALUCtr）、数据内存（Data Memory）、寄存器（Register）、数据选择器（MUX）等部件输出正确的控制信号。

Ctr 的工作模式如下所示：

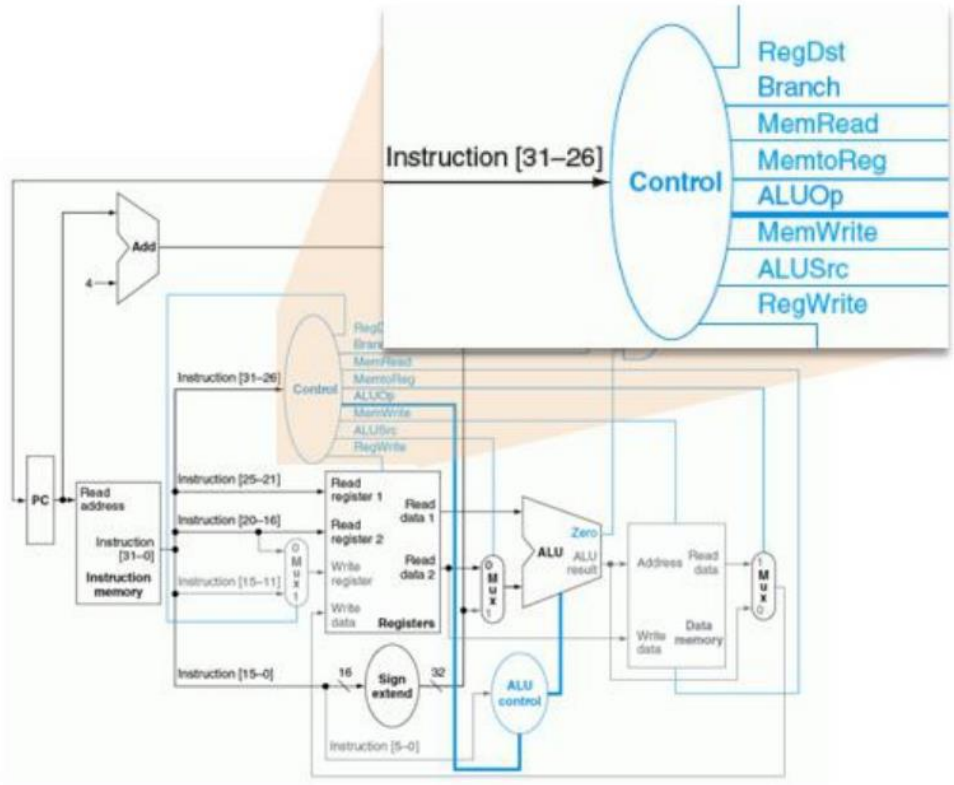


图 1. 主控制模块的 IO 定义

其中，根据输入的指令 instruction[31:26]，可以将 MIPS 指令做如下分类：

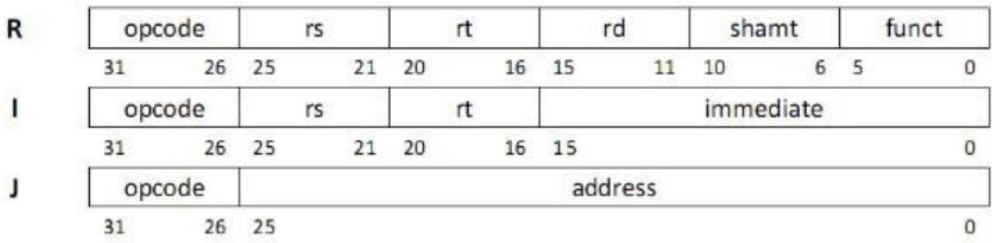


图 2. MIPS 基本指令格式

Ctr 输出的控制信号的含义为：

信号名称	具体含义

RegDst	目标寄存器选择（0：rt，1：rd）
Branch	跳转使能信号（高电平有效）
MemRead	内存读使能（高电平有效）
MemtoReg	内存读取内容写回寄存器信号（高电平有效）
ALUOp	ALU 需执行的操作类型
MemWrite	内存写使能（高电平有效）
ALUSrc	ALU 第二操作数选择信号（0：rt，1：立即数）
RegWrite	寄存器写使能（高电平有效）

表 1. 主控制模块输出的控制信号类型

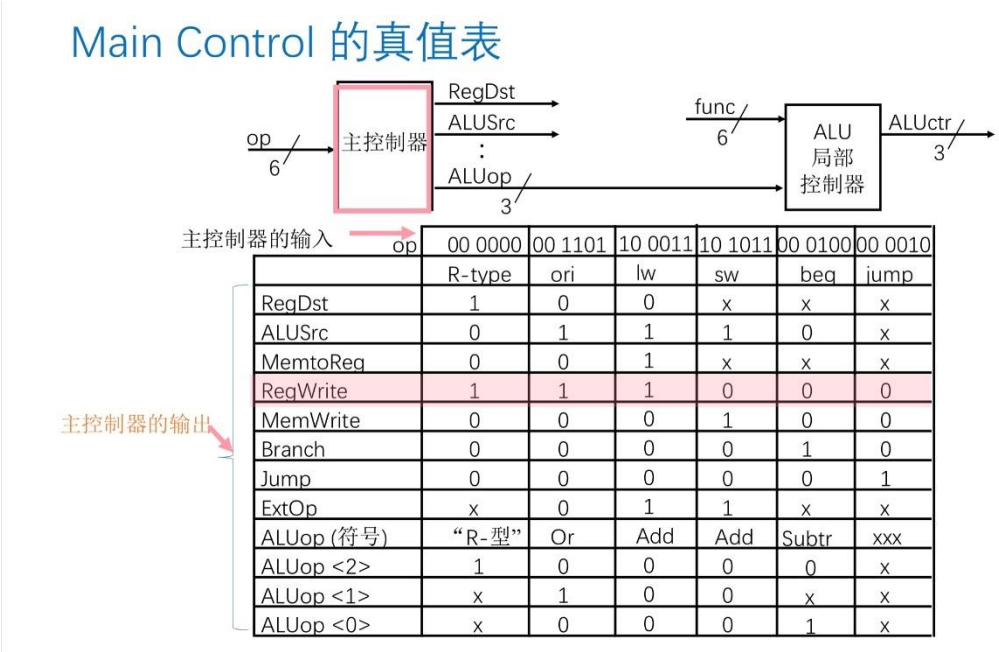
2.1.1.2 Ctr 模块编码与译码

MIPS 指令主要有 R 型、I 型与 J 型之分，可通过 opCode 进行判别。

指令类型	opCode
R 型：add，sub，and，or，slt	000000
I 型：lw	100011
I 型：sw	101011
I 型：beq	000100
J 型：j	000010

表 2. MIPS 指令的 opCode 编码规则

主控制器 Ctr 再将 opCode 转化为控制信号的译码规则如下图所示：



主控制器的输入 op	00 0000 R-type	00 1101 ori	10 0011 lw	10 1011 sw	00 0100 beq	00 0010 jump
RegDst	1	0	0	x	x	x
ALUSrc	0	1	1	1	0	x
MemtoReg	0	0	1	x	x	x
RegWrite	1	1	1	0	0	0
MemWrite	0	0	0	1	0	0
Branch	0	0	0	0	1	0
Jump	0	0	0	0	0	1
ExtOp	x	0	1	1	x	x
ALUop (符号)	“R-型”	Or	Add	Add	Subtr	xxx
ALUop <2>	1	0	0	0	0	x
ALUop <1>	x	1	0	0	x	x
ALUop <0>	x	0	0	0	1	x

主控制器的输出

图 3. 主控制模块的译码规则

2.1.2 运算单元控制器模块 ALUCtr

2.1.2.1 ALUCtr 模块描述

ALU 的控制器模块(ALUCtr）是根据主控制器输出的控制信号 ALUOp 来判断指令类型的。

特别地，对于 R 型指令，由于一个 opCode 可能对应多种运算指令，单凭 opCode 译码而获得 的 ALUOp 尚不足以确定 ALU 操作类型。因此 R 型指令还需根据指令的后 6 位（funct）进一步区分。ALUCtr 需综合这两处输入，以得到正确的控制信号来控制 ALU 执行正确的操作。

其中 ALUCtr 的工作原理如下图所示：

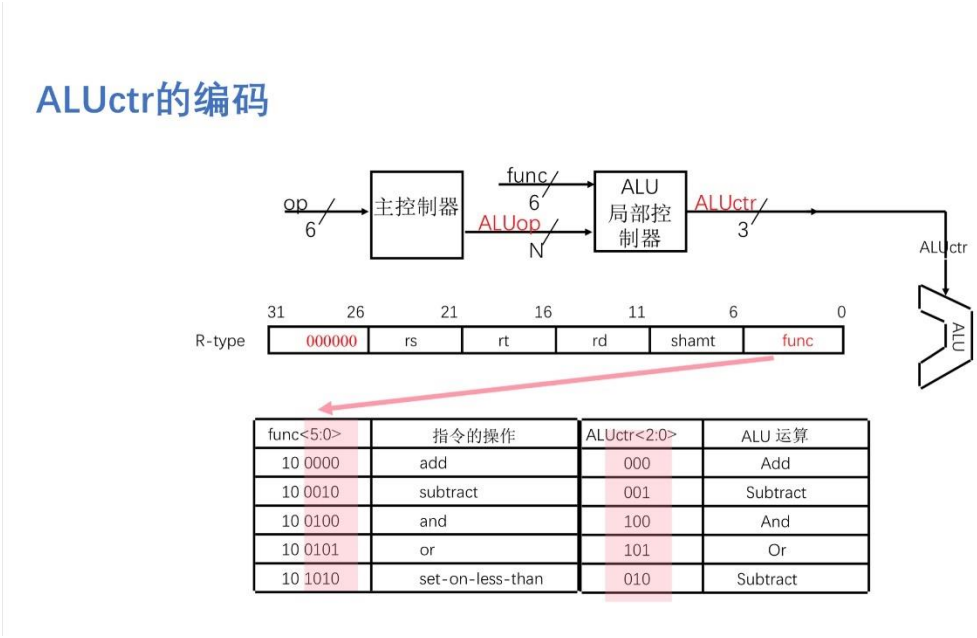


图 4. ALUCtr 的编码

2.1.2.2 ALUCtr 模块编码与译码

ALUCtr 将 ALUOp 以及 funct 翻译为 ALUctrOut 信号：

图 5. ALUCtr 的 译码规则

2.1.3 算术逻辑单元 ALU

2.1.3.1 ALU 模块描述

算术逻辑单元 ALU 根据 ALUCtr 信号将两个输入执行对应的操作，ALURes 为输出结果。若做减法操作，当 ALURes 结果为 0 时，则 Zero 输出置为 1。

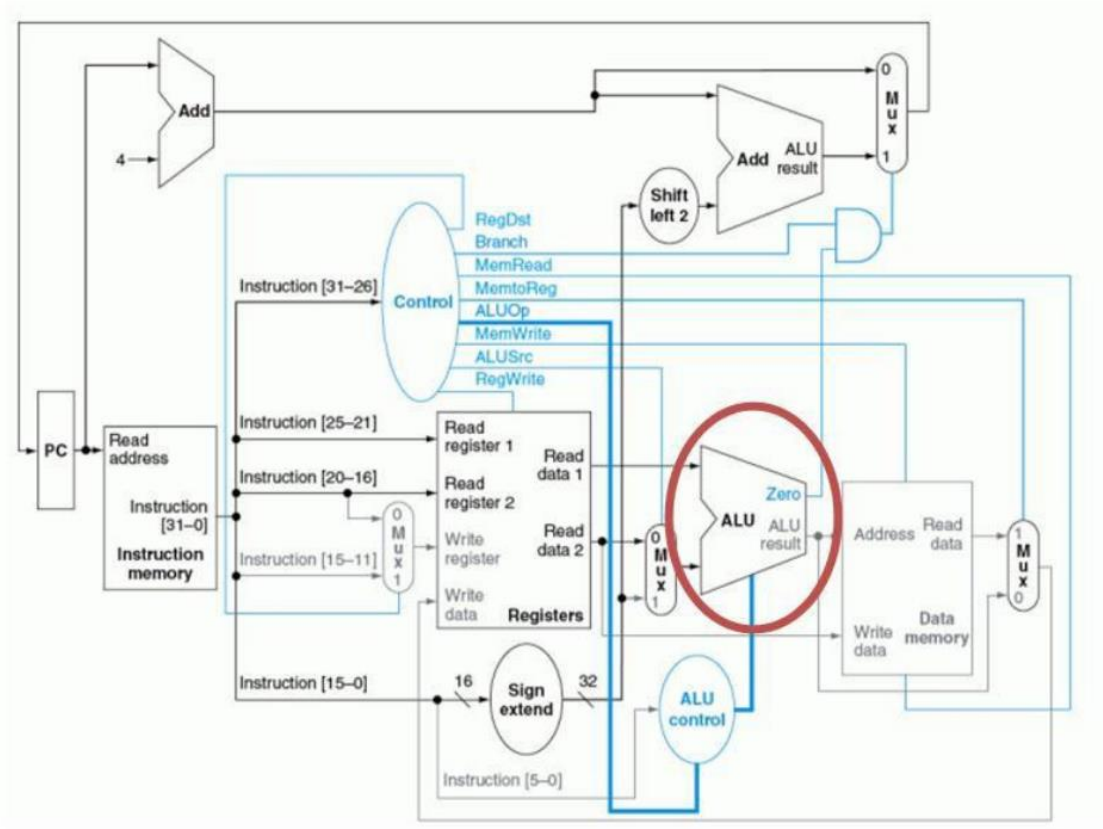


图 6. ALU 模块

2.1.3.2 ALU 模块编码与译码

ALUctrOut 对应不同的 ALU 功能：

ALUctrOut[3:0]	ALU 功能
0000	与运算

0001	或运算
0010	加法运算
0110	减法运算
0111	小于时置位

表 3.ALUctrOut 编码规则

2.1.4 寄存器 Register

寄存器（Register）的功能是暂时存储一些数据以及中间运算结果，其主要接口如图 7 所示。寄存器单元内部一共有 32 个寄存器，所以需要五位的二进制数来确定选中的目标寄存器。由于 MIPS 处理器的存储数据都是 32 位的，所以寄存器输入输出的 data bus 都是 32 位的。除此之外，寄存器还有两个控制信号：时钟 CLK 和写操作 RegWrite。其中，CLK 信号用于控制寄存器的读写，我们要求寄存器在 CLK 发生变化时都能读，在时钟下跳沿写，这样设计能很好地解决多周期 MIPS 处理器不同阶段的寄存器读写冲突问题；RegWrite 信号控制是否进行写入寄存器的操作。

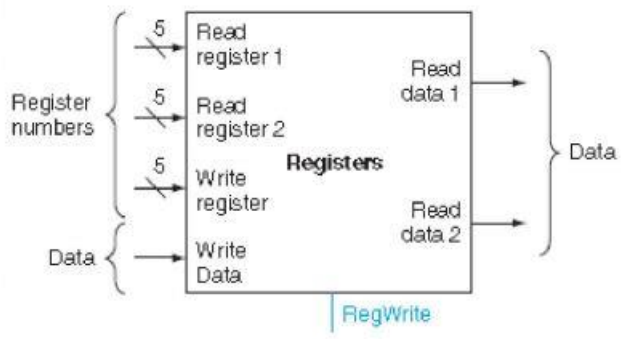


图 7. 寄存器（Register）的图示

2.1.5 存储器模块 Data Memory

存储器（Data Memory）的功能是存储大量内存数据并支持数据的读写，其示意图如图 8 所示。MIPS 处理器能够支持 32 位的数据地址单元，所以 address 接口是 32 位的 input wire。由于 MIPS 支持存储的内存数据的大小是 32 bit，所以 Write data 输入接口和 Read data 输出接口都是 32 位的。除此之外，存储器还有三个控制信号：CLK、MemWrite 以及 MemRead。其中，时钟 CLK 上升沿支持存储器读，下跳沿支持存储器写，这样通过硬件的方式，能够解决 MIPS 多周期处理器在不同阶段同时用到存储器的冒险冲突问题。MemWrite 信号控制数据是否写入存储器，MemRead 信号控制是否从存储器中读取数据。

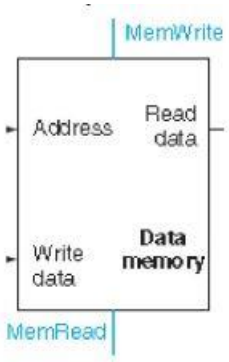


图 8. 存储器（Data Memory）的图示

2.1.6 有符号扩展单元 Signext

有符号扩展单元（Signext）的功能是将指令中的 16 位有符号立即数扩展成为 32 位有符号立即数。由于是带符号扩展，所以只需要将 16 位有符号立即数的符号位扩展到 32 位有符号立即数的高 16 位即可。这个功能可以用数字逻辑运算来进行实现，在最终的代码实现

过程中，我将它设计为了一个 module。

2.1.7 指令存储器 InstMemory

指令存储器 InstMemory 的原理与存储器 Data Memory 相似，只不过 InstMemory 不支持写操作，只能读取输入地址所对应的指令。

2.1.8 数据选择器 Mux

数据选择器 Mux 的原理是根据控制信号，从两个输入里面选择对应的一个进行输出，Mux 的示意图如图 9 所示。数据选择器的实现很简单，一个三目运算符即可：

```
Assign OUT = SEL ? INPUT1 : INPUT2;
```

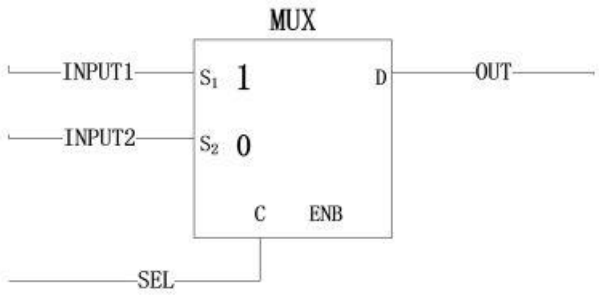


图 9. 数据选择器（Mux）的原理图

2.1.9 程序计数器 PC

程序计数器的功能是输出下一个指令的地址。它有两个控制信号：CLK 和 Reset。其中，CLK 信号控制 PC 何时读取下一条指令的地址，在我的实现代码里，PC 在 CLK 上升沿读取地址；Reset 信号控制何时将 PC 计数器置零，当 Reset 信号为 1 时，PC 的输出置为零。因为有

多个控制信号，我将它设计为一个 module。

2.2 流水线各阶段原理分析

2.2.1 取指阶段（IF）

该阶段主要包括程序计数器 PC、指令寄存器 InstMemory 和一个加法运算单元 ALU。该阶段的作用为从指令寄存器里取出当前需要执行的指令。

2.2.2 译码阶段（ID）

该阶段主要包括寄存器单元 Register、有符号立即数扩展单元 Signext 和主控制模块 Ctr。该阶段的作用为根据指令访问寄存器以及设置控制信号以供后续阶段使用。

2.2.3 执行阶段（EX）

该阶段主要包括运算单元 ALU、运算单元控制器 ALUCtr 和数据选择器 Mux。该阶段的作用为执行指令对应的操作，是一个执行指令的主体部分。

2.2.4 访存阶段（MEM）

该阶段主要包括数据存储器 Data Memory。该阶段的作用为根据指令对内存单元进行读写。

2.2.5 写回阶段（WB）

该阶段主要包括数据选择器 Mux。该阶段的作用为根据指令来决

定是否将数据写入 Register（ID 阶段的寄存器）。

2.3 顶层模块 Top 原理分析

顶层模块 Top 的功能是将之前单独设计的 MIPS 功能模块有机组合在一起，使之能合作完成类 MIPS 多周期处理器的功能。由于在类 MIPS 多周期处理器中，一条指令是分阶段执行的，所以我们需要在每个阶段之间加入段寄存器来储存上一个阶段的运行结果，以便在下一个时钟上升沿来临时，将数据读入下一个阶段。我并没有把段寄存器设计为一个 module，而是直接在 Top 模块里面使用寄存器变量，让它发挥段寄存器的功能。MIPS 多周期处理器的电路设计图如图 10 所示。

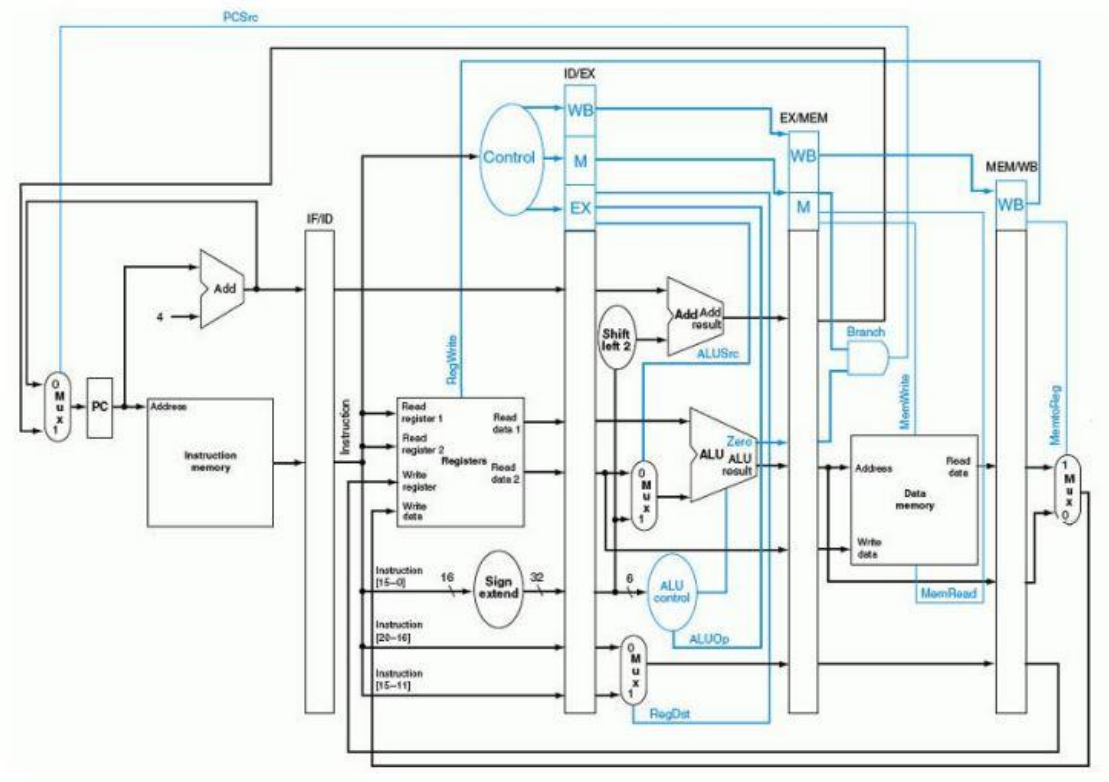


图 10. 类 MIPS 多周期处理器的原理图

3. 功能实现

这里只展示了新写的 Top 模块代码，其余模块（如 Ctr、ALU、ALUCtr 等模块）的代码在实验三、实验四和实验五中已展示，这里不再赘述。

3.1 顶层设计模块 Top 的实现代码

```
module Top_Multi(


    input RESET,
    input CLK


    );


    wire [31:0] IF_INST_W;
    wire [31:0] IF_PC4_W;
    reg [31:0] IF_PC4_R;
    wire [31:0] INST_ADDRESS;
    reg [31:0] IN_ADDRESS;


    InstMemory instMemory(
        .instAddress(INST_ADDRESS[31:0]),
        .inst(IF_INST_W[31:0])
    );


    PC Pc(
        .RESET(RESET),
        .Clk(CLK),
        .inAddress(IN_ADDRESS[31:0]),
        .outAddress(INST_ADDRESS[31:0])
```

```
);

ALU PCadder(
        .input1(INST_ADDRESS[31:0]),
        .input2(4),
        .aluCtr(4'b0010),
        .aluRes(IF_PC4_W[31:0])
);

wire [4:0] WRITE_REG;
wire [31:0] WRITE_DATA;
reg [31:0] ID_REG_READ_DATA1;
reg [31:0] ID_REG_READ_DATA2;
wire [31:0] ID_REG_READ_DATA1_W;
wire [31:0] ID_REG_READ_DATA2_W;
wire REG_WRITE;
reg [31:0] IF_INST_R;
reg [4:0] REG_READ_1;
reg [4:0] REG_READ_2;

Registers register (
        .Reset(RESET),
        .Clk(CLK),
        .readReg1(IF_INST_R[25:21]),
        .readReg2(IF_INST_R[20:16]),
        .writeReg(WRITE_REG),
        .writeData(WRITE_DATA),
        .regWrite(REG_WRITE),
        .readData1(ID_REG_READ_DATA1_W),
```

```
        .readData2(ID_REG_READ_DATA2_W)
    );

    reg [31:0] ID_SIGNEXT_OUT;
    wire [31:0] ID_SIGNEXT_OUT_W;

    signext SignExt(
        .inst(IF_INST_R[15:0]),
        .data(ID_SIGNEXT_OUT_W)
    );

    reg ID_REG_DST;
    reg ID_ALU_SRC;
    reg ID_MEM_TO_REG;
    reg ID_REG_WRITE;
    reg ID_MEM_READ;
    reg ID_MEM_WRITE;
    reg ID_BRANCH;
    reg [1:0] ID_ALU_OP;
    reg ID_JUMP;
    wire ID_REG_DST_W;
    wire ID_ALU_SRC_W;
    wire ID_MEM_TO_REG_W;
    wire ID_REG_WRITE_W;
    wire ID_MEM_READ_W;
    wire ID_MEM_WRITE_W;
    wire ID_BRANCH_W;
    wire [1:0] ID_ALU_OP_W;
    wire ID_JUMP_W;
```

```
Ctr mainCtr(
        .opCode(IF_INST_R[31:26]),
        .regDst(ID_REG_DST_W),
        .aluSrc(ID_ALU_SRC_W),
        .memToReg(ID_MEM_TO_REG_W),
        .regWrite(ID_REG_WRITE_W),
        .memRead(ID_MEM_READ_W),
        .memWrite(ID_MEM_WRITE_W),
        .branch(ID_BRANCH_W),
        .aluOp(ID_ALU_OP_W),
        .jump(ID_JUMP_W)
);

reg [31:0] ID_PC4;
reg [4:0] ID_INST_0;
reg [4:0] ID_INST_1;
reg [4:0] EX_WRITE_REG;
wire [4:0] EX_WRITE_REG_W;

Mux5 muxWriteReg(
        .ctrSignal(ID_REG_DST),
        .input1(ID_INST_1),
        .input2(ID_INST_0),
        .result(EX_WRITE_REG_W)
);

wire [31:0] ALU_INPUT;
```

```
Mux muxAluInput (
        .ctrSignal(ID_ALU_SRC),
        .input1(ID_SIGNEXT_OUT),
        .input2(ID_REG_READ_DATA2),
        .result(ALU_INPUT)
);

wire [3:0] ALUCTR_OUT;

ALUCtr ALUCTR(
        .aluOp(ID_ALU_OP),
        .funct(ID_SIGNEXT_OUT[5:0]),
        .aluCtrOut(ALUCTR_OUT[3:0])
);

reg EX_ZERO;
wire EX_ZERO_W;
reg [31:0] EX_DATA_MEM_ADDRESS;
wire [31:0] EX_DATA_MEM_ADDRESS_W;

ALU Alu(
        .input1(ID_REG_READ_DATA1),
        .input2(ALU_INPUT),
        .aluCtr(ALUCTR_OUT[3:0]),
        .zero(EX_ZERO_W),
        .aluRes(EX_DATA_MEM_ADDRESS_W[31:0])
);

reg [31:0] EX_addALU_OUT;
```

```
wire [31:0] EX_addALU_OUT_W;

ALU addALU(
        .input1(ID_PC4),
        .input2(ID_SIGNEXT_OUT << 2),
        .aluCtr(4'b0010),
        .aluRes(EX_addALU_OUT_W)
);

reg [31:0] EX_REG_READ_DATA2;
reg EX_REG_WRITE;
reg EX_MEM_READ;
reg EX_MEM_WRITE;
reg EX_BRANCH;
reg EX_MEM_TO_REG;
reg EX_JUMP;

wire [31:0] NO_JUMP_ADDRESS;
wire BRANCH = EX_BRANCH & EX_ZERO;

Mux mux_BRANCH(
        .ctrSignal(BRANCH),
        .input1(EX_addALU_OUT),
        .input2(IF_PC4_W),
        .result(NO_JUMP_ADDRESS[31:0])
);

wire [31:0] IN_ADDRESS_W;
```

```
Mux mux_JUMP(
        .ctrSignal(ID_JUMP_W),
        .input1((IF_INST_R[25:0] << 2) + (IF_PC4_R & 32'hF0000000)),
        .input2(NO_JUMP_ADDRESS),
        .result(IN_ADDRESS_W[31:0])
);

reg [31:0] MEM_dataMem_READ_DATA;
wire [31:0] MEM_dataMem_READ_DATA_W;

dataMemory dataMem(
        .Clk(CLK),
        .address(EX_DATA_MEM_ADDRESS[31:0]),
        .writeData(EX_REG_READ_DATA2),
        .memWrite(EX_MEM_WRITE),
        .memRead(EX_MEM_READ),
        .readData(MEM_dataMem_READ_DATA_W[31:0])
);

reg [31:0] MEM_DATA_MEM_ADDRESS;
reg MEM_REG_WRITE;
reg MEM_MEM_TO_REG;
reg MEM_JUMP;
reg [4:0] MEM_WRITE_REG;

assign WRITE_REG = MEM_WRITE_REG;
assign REG_WRITE = MEM_REG_WRITE;

Mux muxWriteData(
```

```
        .ctrSignal(MEM_MEM_TO_REG),
        .input1(MEM_dataMem_READ_DATA),
        .input2(MEM_DATA_MEM_ADDRESS[31:0]),
        .result(WRITE_DATA)
    );

    wire STALL = REG_WRITE & ((MEM_WRITE_REG == REG_READ_1) |
(MEM_WRITE_REG == REG_READ_2));

    always @ (RESET)
        begin
            if(RESET)
                begin
                    IF_PC4_R = 0;
                    ID_REG_READ_DATA1 = 0;
                    ID_REG_READ_DATA2 = 0;
                    IF_INST_R = 0;
                    ID_SIGNEXT_OUT = 0;
                    ID_REG_DST = 0;
                    ID_ALU_SRC = 0;
                    ID_MEM_TO_REG = 0;
                    ID_REG_WRITE = 0;
                    ID_MEM_READ = 0;
                    ID_MEM_WRITE = 0;
                    ID_BRANCH = 0;
                    ID_ALU_OP = 0;
                    ID_JUMP = 0;
                    ID_PC4 = 0;
                    ID_INST_0 = 0;
```

```
                    ID_INST_1 = 0;
                    EX_WRITE_REG = 0;
                    EX_ZERO = 0;
                    EX_DATA_MEM_ADDRESS = 0;
                    EX_addALU_OUT = 0;
                    EX_REG_READ_DATA2 = 0;
                    EX_REG_WRITE = 0;
                    EX_MEM_READ = 0;
                    EX_MEM_WRITE = 0;
                    EX_BRANCH = 0;
                    EX_MEM_TO_REG = 0;
                    EX_JUMP = 0;
                    MEM_dataMem_READ_DATA = 0;
                    MEM_DATA_MEM_ADDRESS = 0;
                    MEM_REG_WRITE = 0;
                    MEM_MEM_TO_REG = 0;
                    MEM_JUMP = 0;
                    MEM_WRITE_REG = 0;
                end
        end

    always @ (posedge CLK)
        begin
            if(STALL || BRANCH)
                begin
                    ID_REG_READ_DATA1 <= 0;
                    ID_REG_READ_DATA2 <= 0;
                    ID_SIGNEXT_OUT <= 0;
                    ID_REG_DST <= 0;
```

```
                ID_ALU_SRC <= 0;
                ID_MEM_TO_REG <= 0;
                ID_REG_WRITE <= 0;
                ID_MEM_READ <= 0;
                ID_MEM_WRITE <= 0;
                ID_BRANCH <= 0;
                ID_ALU_OP <= 0;
                ID_JUMP <= 0;
                ID_PC4 <= 0;
                ID_INST_0 <= 0;
                ID_INST_1 <= 0;
                EX_MEM_READ <= 0;
                EX_MEM_WRITE <= 0;
                IN_ADDRESS <= IN_ADDRESS - 12;
            end
        IN_ADDRESS <= IN_ADDRESS_W;
        IF_INST_R <= IF_INST_W;
        IF_PC4_R <= IF_PC4_W;
        ID_PC4 <= IF_PC4_R[31:0];
        ID_INST_0 <= IF_INST_R[20:16];
        ID_INST_1 <= IF_INST_R[15:11];
        ID_REG_READ_DATA1 <= ID_REG_READ_DATA1_W;
        ID_REG_READ_DATA2 <= ID_REG_READ_DATA2_W;
        ID_SIGNEXT_OUT <= ID_SIGNEXT_OUT_W;
        ID_REG_DST <= ID_REG_DST_W;
        ID_ALU_SRC <= ID_ALU_SRC_W;
        ID_MEM_TO_REG <= ID_MEM_TO_REG_W;
        ID_REG_WRITE <= ID_REG_WRITE_W;
        ID_MEM_READ <= ID_MEM_READ_W;
```

```
                ID_MEM_WRITE <= ID_MEM_WRITE_W;
                ID_BRANCH <= ID_BRANCH_W;
                ID_ALU_OP <= ID_ALU_OP_W;
                ID_JUMP <= ID_JUMP_W;
                REG_READ_1 <= IF_INST_R[25:21];
                REG_READ_2 <= IF_INST_R[20:16];

                EX_REG_READ_DATA2 <= ID_REG_READ_DATA2[31:0];
                EX_REG_WRITE <= ID_REG_WRITE;
                EX_MEM_READ <= ID_MEM_READ;
                EX_MEM_WRITE <= ID_MEM_WRITE;
                EX_BRANCH <= ID_BRANCH;
                EX_MEM_TO_REG <= ID_MEM_TO_REG;
                EX_JUMP <= ID_JUMP;
                EX_ZERO <= EX_ZERO_W;
                EX_DATA_MEM_ADDRESS <= EX_DATA_MEM_ADDRESS_W;
                EX_addALU_OUT <= EX_addALU_OUT_W;

                MEM_DATA_MEM_ADDRESS <= EX_DATA_MEM_ADDRESS[31:0];
                MEM_REG_WRITE <= EX_REG_WRITE;
                MEM_MEM_TO_REG <= EX_MEM_TO_REG;
                MEM_JUMP <= EX_JUMP;
                MEM_WRITE_REG <= EX_WRITE_REG;
                MEM_dataMem_READ_DATA <= MEM_dataMem_READ_DATA_W;
                EX_WRITE_REG <= EX_WRITE_REG_W;

                if((REG_READ_1 == EX_WRITE_REG) && (EX_MEM_TO_REG == 0) &&
(RESET == 0))
                        begin
```

```
                    ID_REG_READ_DATA1 <= EX_DATA_MEM_ADDRESS;
                end
            else if((REG_READ_2 == EX_WRITE_REG) && (EX_MEM_TO_REG == 0)
&& (RESET == 0))
                begin
                    ID_REG_READ_DATA2 <= EX_DATA_MEM_ADDRESS;
                end
            else if((MEM_WRITE_REG == REG_READ_1) && (MEM_MEM_TO_REG
== 0) && (RESET == 0))
                begin
                    ID_REG_READ_DATA1 <= EX_DATA_MEM_ADDRESS;
                end
            else if((MEM_WRITE_REG == REG_READ_2) && (MEM_MEM_TO_REG
== 0) && (RESET == 0))
                begin
                    ID_REG_READ_DATA2 <= EX_DATA_MEM_ADDRESS;
                end
        end
endmodule
```

Top 模块实现了检测竞争并插入停顿（STALL）机制来解决数据冒险、控制冒险和结构冒险，相关代码如下：

```
if(STALL || BRANCH)
    begin
        ID_REG_READ_DATA1 <= 0;
        ID_REG_READ_DATA2 <= 0;
        ID_SIGNEXT_OUT <= 0;
        ID_REG_DST <= 0;
        ID_ALU_SRC <= 0;
        ID_MEM_TO_REG <= 0;
        ID_REG_WRITE <= 0;
        ID_MEM_READ <= 0;
        ID_MEM_WRITE <= 0;
        ID_BRANCH <= 0;
        ID_ALU_OP <= 0;
        ID_JUMP <= 0;
        ID_PC4 <= 0;
        ID_INST_0 <= 0;
        ID_INST_1 <= 0;
        EX_MEM_READ <= 0;
        EX_MEM_WRITE <= 0;
        IN_ADDRESS <= IN_ADDRESS - 12;
    end
```

同时，Top 模块部分实现了 Forwarding 机制来解决数据竞争，在一定程度上上减少了因数据竞争带来的流水线停顿，从而提高流水线处理性能。相关代码如下：

```
if((REG_READ_1 == EX_WRITE_REG) && (EX_MEM_TO_REG == 0) && (RESET == 0))
    begin
        ID_REG_READ_DATA1 <= EX_DATA_MEM_ADDRESS;
    end
else if((REG_READ_2 == EX_WRITE_REG) && (EX_MEM_TO_REG == 0) && (RESET == 0))
    begin
        ID_REG_READ_DATA2 <= EX_DATA_MEM_ADDRESS;
    end
else if((MEM_WRITE_REG == REG_READ_1) && (MEM_MEM_TO_REG == 0) && (RESET == 0))
    begin
        ID_REG_READ_DATA1 <= EX_DATA_MEM_ADDRESS;
    end
else if((MEM_WRITE_REG == REG_READ_2) && (MEM_MEM_TO_REG == 0) && (RESET == 0))
    begin
        ID_REG_READ_DATA2 <= EX_DATA_MEM_ADDRESS;
    end
```

4. 仿真测试

我们首先在内存中载入如下数据：

```
00000000
00000001
00000002
00000003
00000004
00000005
00000006
00000007
00000008
00000009
0000000A
0000000B
0000000C
0000000D
0000000E
0000000F
00000010
00000011
00000012
00000013
00000014
00000015
00000016
00000017
00000018
00000019
0000001A
0000001B
```

```
0000001C
0000001D
0000001E
0000001F
```

并用以下命令读取内存文件：

$readmemh("F:/archlabs/lab06/mem_data.dat",MIPS.dataMem.memFile);

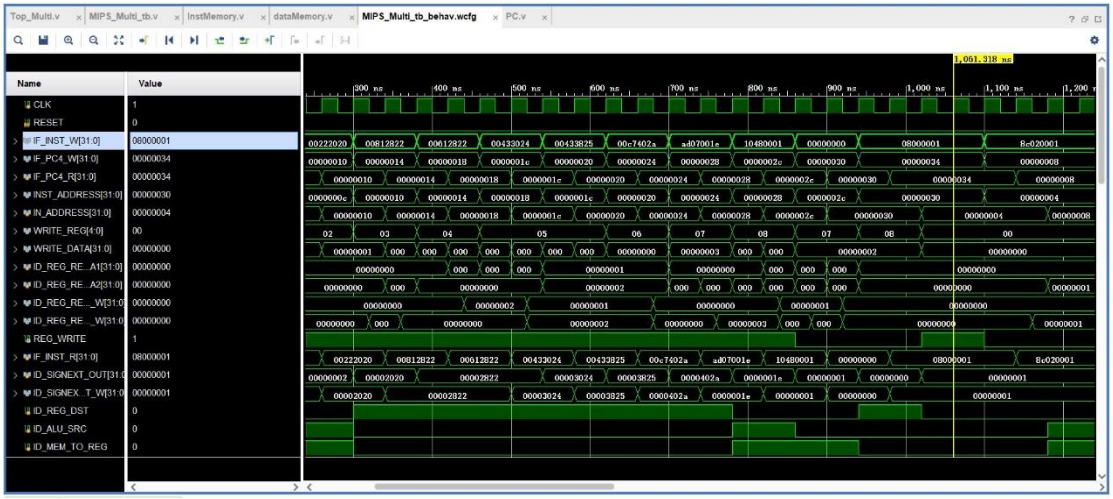
同时，设计包含数据冒险的汇编指令（例如 lw 指令有冲突，sub 指令有冲突）进行测试：

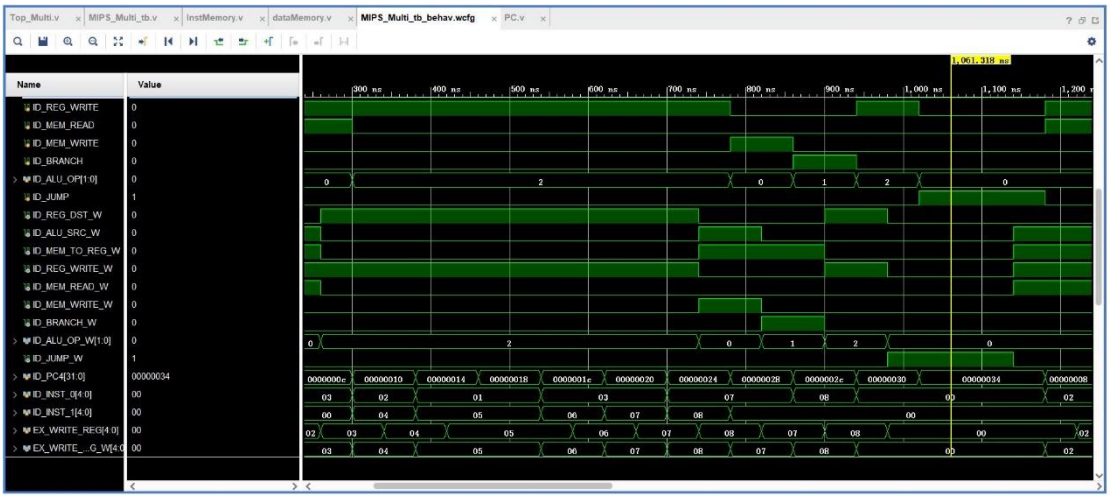
```
10001100000000010000000000000000 // lw $1,0($0) $1 = 0
10001100000000010000000000000001 // lw $2,1($0) $2 = 1
10001100000000011000000000000010 // lw $3,2($0) $3 = 2
00000000001000100010000000100000 // add $4 = $1 + $2 = 1
00000000100000010010100000100010 // sub $5 = $4 - $1 = 1
00000000110000010010100000100010 // sub $5 = $3 - $1 = 2
00000000100001100110000000100100 // and $6 = $2 and $3 = 0
00000000100001100111100000100101 // or $7 = $2 or $3 = 3
00000000110001110100000000101010 // slt $6,$7,$8    $8 = 1
10101101000001110000000000011110 // sw $7,30($8) mem_data[31] = $7 = 3
00010000010010000000000000000001 // beq $2,$8,1    if $2==$8 jump to jump inst
00000000000000000000000000000000 // nop
00001000000000000000000000000001 // jump to inst1
00000000000000000000000000000000 // nop
00000000000000000000000000000000 // nop
00000000000000000000000000000000 // nop
```

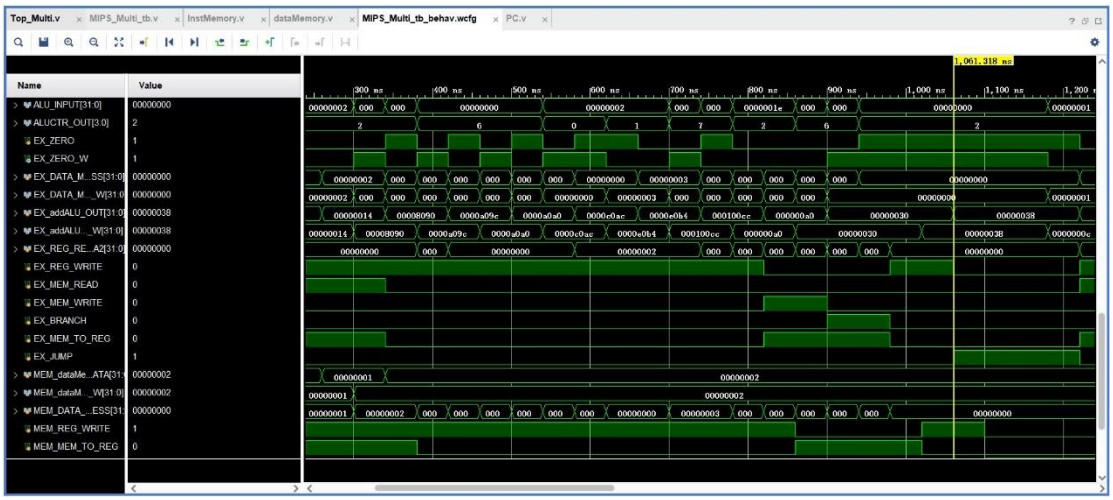
用以下指令读取汇编指令文件：

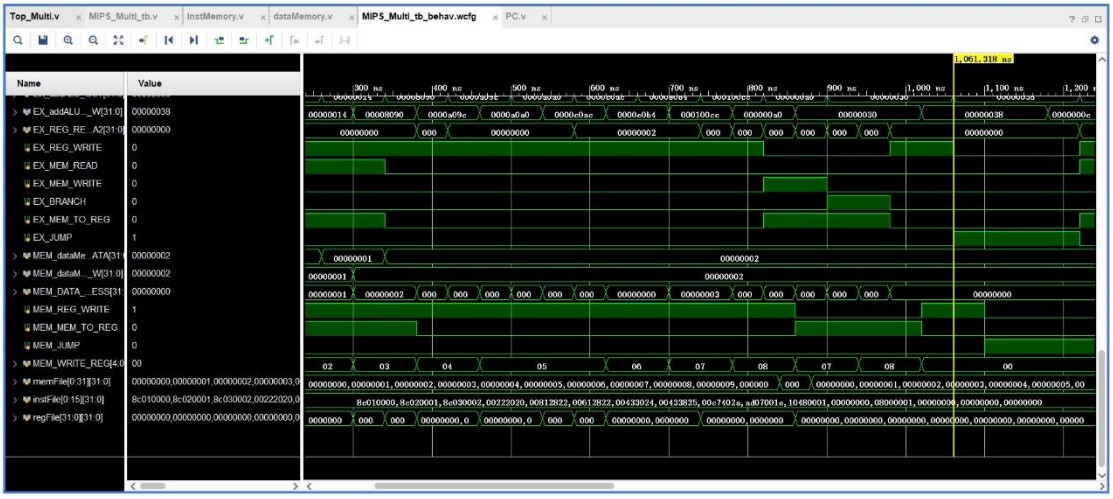
$readmemb("F:/archlabs/lab06/mem_inst.dat",MIPS.instMemory.instFile);

基于以上文件，我们进行仿真测试，测试结果如图 10 所示。









注：程序在黄线所处时刻之后发生跳转

从仿真结果可以看出，我们的类 MIPS 多周期处理器实现成功。

5. 实验总结

5.1 实验评价

本实验实现了一个类 MIPS 多周期处理器。本实验的基础是实验五的类 MIPS 单周期处理器。我们根据执行功能将 MIPS 单周期处理器分为五个阶段，并通过段寄存器将各个阶段联系起来，在 Top 模块中将这些部分组装起来，最终实现类 MIPS 多周期处理器，并得到了正确的仿真结果。

5.2 实验心得

这一次的实验也很难，我也做了很久，感觉和实验五差不多难，遇到的问题也很多。

在一开始设计 Top 模块的时候，我为了图方便，直接将各个功能模块输出到 reg，导致最后在时钟上升时沿无法赋值。另外，在这个

程序中，下一个阶段到来时对段寄存器的赋值应该非阻塞式赋值，我一开始也没有意识到这一点，直接用的阻塞式赋值。

一旦类 MIPS 多周期处理器能跑起来了。STALL 机制的实现就比较容易了。我是通过判断 WB 阶段写入的寄存器是否与 ID 阶段读取的寄存器发生冲突来实现 STALL 机制的。如果发生冲突，则清空相关段寄存器（相当于执行一条 nop 指令），然后再重新读入该指令。

Forwarding 机制比较难，我写的 Forwarding 机制只能在某些情况下能正确完成其功能。我是通过判断需要写入寄存器的数值是否在计算出来后、写入寄存器前就被 ID 阶段读入，如果是的话，就直接将计算结果复制到 ID 的段寄存器。

这次实验让我对 MIPS 多周期处理器的工作流程有了很清晰的认识。之前上系统结构的时候，对这个部分的知识总是感觉一知半解、模模糊糊的，自从开始做这个实验，我硬是把之前没有搞懂的流程给搞明白了，对相关性和冒险也有了非常深刻的认识。这个实验真的让我受益匪浅。