

武汉大学计算机学院

本科生实验报告

MIPS 流水线 CPU 设计与实现

专 业 名 称：计算机科学与技术

课 程 名 称：计算机组成原理实验

指 导 教 师：龚奕利 副教授


学 生 学 号：██████████

学 生 姓 名：██████

二〇一九年五月

郑 重 声 明

本人呈交的实验报告，是在指导老师的指导下，独立进行实验工作所取得的成果，所有数据、图片资料真实可靠。尽我所知，除文中已经注明引用的内容外，本实验报告不包含他人享有著作权的内容。对本实验报告做出贡献的其他个人和集体，均已在文中以明确的方式标明。本实验报告的知识产权归属于培养单位。

本人签名：  _____

日期：2019.5.4

摘 要

计算机组成原理实验的实验目的是了解提高 CPU 性能的方法，掌握流水线 MIPS 微处理器的工作原理，理解数据冒险、控制冒险以及解决流水线冲突的方法，掌握流水线 MIPS 微处理器的测试方法。

实验设计主要遵循 MIPS 流水线 CPU 的设计原理。

实验内容主要包括：设计一个 32 位流水线 MIPS 微处理器。

1) 执行下列 MIPS32 指令。

① 算数运算指令：ADD ADDU SUB SUBU ADDI ADDU LUI

② 逻辑运算指令：AND OR ANDI ORI SLT SLTU NOR

③ 移位指令：SLL SRL SLLV SRLV

④ 条件分支指令：BEQ BNE

⑤ 无条件跳转指令：J JAL JR

⑥ 数据传输指令：LW SW

2) 在用 5 级流水线技术，对数据冒险实现转发或阻塞功能。

3) 用 verilog HDL 语言，modelsim，MARS 软件仿真。

实验结论为通过这次实验，我充分理解了流水线工作原理。

关键词：流水线；CPU；verilog HDL；modelsim；MARS

目录

1	实验目的和意义	1
1.1	实验目的	1
1.2	实验意义	1
2	实验设计	2
2.1	概述	2
2.2	实验环境	2
3.1	Verilog HDL 简介	2
3.2	ModelSim 简介	2
3.3	MARS 简介	2
3.4	Vivado	3
3.5	Nexys 4DDR	3
3	概要设计	4
3.1	总体设计	4
3.2	程序计数器 (PcUnit)	5
3.3	寄存器设计 (GPR)	5
3.4	算术逻辑运算单元 (ALU)	6
3.5	扩展单元 (Extender)	6
3.6	数据存储单元 (DMem)	7
3.7	指令存储器 (IM)	7
3.8	控制器 (Ctrl)	8
3.9	模型机 (MIPS)	8
3.10	IF/ID 级段间寄存器 (IFID)	9
3.11	ID/EX 段间寄存器 (IDEX)	9
3.12	EX/MEM 段间寄存器 (EXMEM)	11
3.13	MEM/WB 段间寄存器 (MEMWB)	12
3.14	冒险检测单元 (Hazard)	13
3.15	多路选择器 (mux)	14
3.16	宏定义文件	14

3.16.1 instruction_def.v	14
3.16.2 ctrl_encode_def.v	14
3.17 mips_tb.v	14
4 详细设计	15
4.1 程序计数器 (PcUnit)	15
4.2 寄存器 (GPR)	16
4.3 算术逻辑运算单元 (ALU)	17
4.4 扩展单元 (Extender)	19
4.5 数据存储器 (DMem)	20
4.6 指令存储器 (IM)	20
4.7 控制器 (Ctrl)	21
4.8 模型机 (MIPS)	29
4.9 IFID 段间寄存器 (IFID)	41
4.10 IDEX 段间寄存器 (IDEX)	42
4.11 EXMEM 段间寄存器 (EXMEM)	46
4.12 MEMWB 段间寄存器 (MEMWB)	49
4.13 多路选择器 (mux)	51
4.14 冒险检测单元 (Hazard)	54
5 测试及结果分析	62
5.1 仿真代码及分析	62
5.2 仿真测试结果	64
结论	65
参考文献	66
教师评语评分	67

1 实验目的和意义

1.1 实验目的

- (1) 了解提高 CPU 性能的方法。
- (2) 掌握流水线 MIPS 微处理器的工作原理。
- (3) 理解数据冒险、控制冒险的概念以及流水线冲突的解决方法。
- (4) 掌握流水线 MIPS 微处理器的测试方法。

1.2 实验意义

- (1) 充分理解流水线 CPU 工作的原理。
- (2) 学会使用 verilog HDL 语言写五级流水线的代码。
- (3) 学会用 modelsim 软件进行仿真实验。

2 实验设计

2.1 概述

流水线式一种可以使多条指令的执行过程相互重叠的实现技巧，目前它是提高处理器处理速度的关键。

通常一条 MIPS 指令包含如下五个处理步骤：

- (1) 从存储器中读取指令
- (2) 指令译码的同时读取寄存器
- (3) 执行操作或计算地址
- (4) 在数据存储器中读取操作数
- (5) 将结果写回寄存器

MIPS 的流水线 CPU 设计将遵循以上几点，并解决冒险等问题。

2.2 实验环境

3.1 Verilog HDL 简介

verilog HDL 是一种硬件描述语言，以文本形式来描述数字系统硬件的结构和行为的语言，用它可以表示逻辑电路图、逻辑表达式，还可以表示数字逻辑系统所完成的逻辑功能。

3.2 ModelSim 简介

Mentor 公司的 ModelSim 是业界最优秀的 HDL 语言仿真软件，它能提供友好的仿真环境，是业界唯一的单内核支持 VHDL 和 Verilog 混合仿真的仿真器。它采用直接优化的编译技术、Tcl/Tk 技术、和单一内核仿真技术，编译仿真速度快，编译的代码与平台无关，便于保护 IP 核，个性化的图形界面和用户接口，为用户加快调错提供强有力的手段，是 FPGA/ASIC 设计的首选仿真软件。

3.3 MARS 简介

MIPS 开发工具,可以将 MIPS 代码编译成机器码。

3.4 Vivado

Vivado 设计套件，是 FPGA 厂商赛灵思公司 2012 年发布的集成设计环境。包括高度集成的设计环境和新一代从系统到 IC 级的工具，这些均建立在共享的可扩展数据模型和通用调试环境基础上。

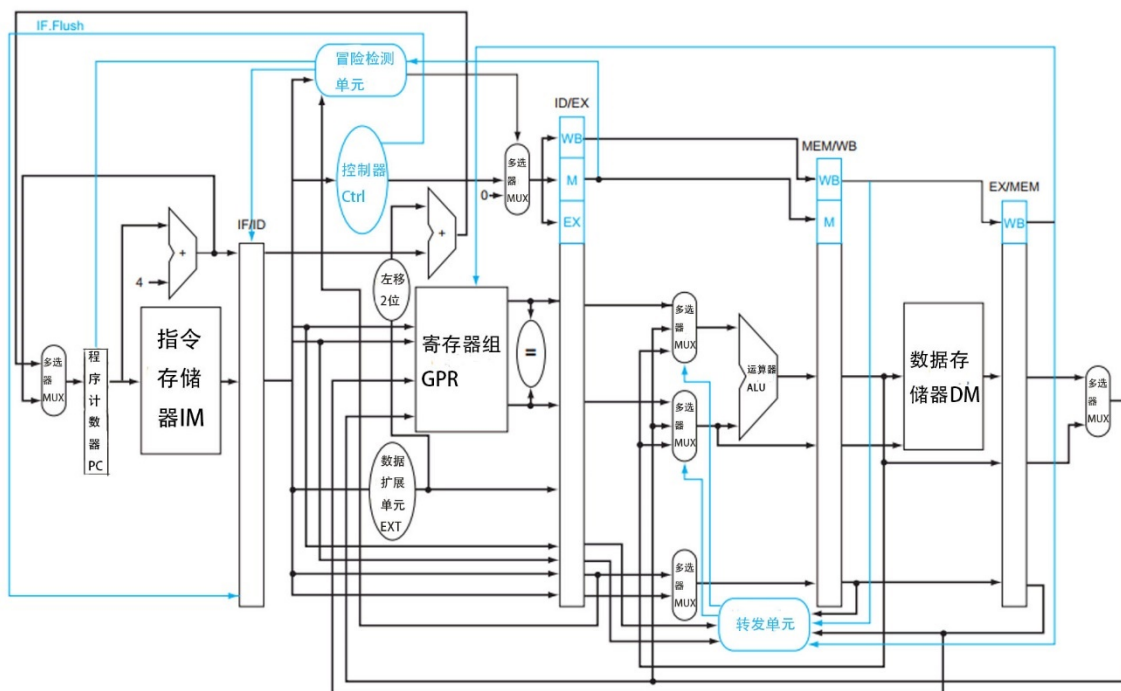
3.5 Nexys 4DDR

Nexys 4 DDR 板是一个完整的，随时可用的数字电路开发平台，基于最新的 Artix-7™现场可编程门阵列(FPGA)从 Xilinx®。凭借其大、高容量 FPGA (Xilinx part number XC7A100T-1CSG324C)、丰富的外部内存以及 USB、以太网和其他端口的集合，Nexys4 DDR 可以承载从介绍性组合电路到功能强大的嵌入式处理器的各种设计。几个内置的外设，包括加速度计、温度传感器、MEMs 数字麦克风、扬声器放大器 and 几个 I/O 设备，使 Nexys4 DDR 可以在不需要任何其他组件的情况下用于广泛的设计。

3 概要设计

3.1 总体设计

流水线 CPU 总体结构如下图所示，其中包括程序计数器（PC）、指令存储器（IM）、寄存器组（GPR）、运算器（ALU）、数据扩展单元（EXT）、数据存储器（DM）和控制器（Ctrl）、冒险检测单元（Hazard）、IF/ID 段间寄存器（IFID）、ID/EX 段间寄存器（IDEX）、EX/MEM 段间寄存器（EXMEM）、MEM/WB 段间寄存器（MEMWB）、多路选择器（mux）。



3.2 程序计数器 (PcUnit)

(1) 功能描述

PcUnit 是程序计数器，找到下一个指令的 PC。

(2) 模块接口

表 2.1 程序计数器模块接口定义

信号名	方向	描述
OldPC [31:0]	I	上一条指令的 PC
stall	I	阻塞信号
PcReSet	I	PC 重置
PcSel	I	左移 2 位信号
Clk	I	时钟信号
Address [31:0]	I	符号扩展结果
Adj [25:0]	I	IFID 级传来的[25:0]的地址
jump	I	j 指令跳转信号
PC	O	指向的下一条的 PC

3.3 寄存器设计 (GPR)

(1) 功能描述

GPR 主要功能是保存寄存器文件，并支持对通用寄存器的访问。

(2) 模块接口

信号名	方向	描述
clk	I	时钟信号
WE	I	寄存器写使能信号 0: 不写, 1: 写
WeSel [4:0]	I	需要写的寄存器的地址
ReSel1 [4:0]	I	需要读的寄存器 1 的地址

ReSel2 [4:0]	I	需要读的寄存器 2 的地址
WData [31:0]	I	需要写的寄存器的数据
DataOut1 [31:0]	O	需要读的寄存器 1 的数据
DataOut2 [31:0]	O	需要读的寄存器 2 的数据

3.4 算术逻辑运算单元（ALU）

（1）功能描述

ALU 主要功能是完成对输入数据的进行加法、减法、与、或、左移、右移、乘法、除法运算以及判断两个操作数是否相等。

（2）模块接口

表 2.3 ALU 模块接口定义

信号名	方向	描述
DataIn1 [31:0]	I	操作数 A
DataIn2 [31:0]	I	操作数 B
ALUCtrl [4:0]	I	需要进行的运算
shame [4:0]	I	位移运算的移动位数
Zero	O	两操作数是否相等
AluResult [31:0]	O	运算结果

3.5 扩展单元（Extender）

（1）功能描述

EXT 主要功能是将 16 位的数据扩展为 32 位数据。

（2）模块接口

信号名	方向	描述
DataIn [15:0]	I	需要进行扩展的数据
EXTOp [1:0]	I	扩展方式的控制信号

		00: 0 扩展 01: 符号扩展 10: 将立即数扩展到高位
ExtOut [31:0]	O	扩展结果

3.6 数据存储器（DMem）

（1）功能描述

DM是数据存储器。

（2）模块接口

信号名	方向	描述
DataAdr [4:0]	I	访问地址
DataIn [31:0]	I	输入到数据寄存器的数据
DMemR	I	读信号
DMemW	I	写信号
clk	I	时钟信号
DataOut [31:0]	O	读出的结果

3.7 指令存储器（IM）

（1）功能描述

IM 是指令存储器。

（2）模块接口

信号名	方向	描述
ImAddress [4:0]	I	访问地址
OpCode [31:0]	O	读出的指令

3.8 控制器 (Ctrl)

(1) 功能描述

Ctrl 是控制信号的单元。

(2) 模块接口

信号名	方向	描述
OpCode [5:0]	I	指令格式中的 OPCODE 域
funct [5:0]	I	指令格式中的 FUNCT 域
jump	O	分支跳转信号
RegDst	O	确定目标寄存器
Branch	O	分支信号
MemR	O	读数据存储器
Mem2R	O	数据存储器到寄存器堆
MemW	O	写数据存储器
RegW	O	寄存器堆写入数据
AluSrc	O	运算器操作数选择
ExtOp [1:0]	O	位扩展/符号扩展选择
AluCtrl [4:0]	O	Alu 运算选择

3.9 模型机 (MIPS)

(1) 功能描述

MIPS 用于连接各模块。

(2) 模块接口

信号名	方向	描述
Clk	I	时钟信号
Reset	I	复位信号

3.10 IF/ID 级段间寄存器（IFID）

（1）功能描述

IFID 用于实现取指与译码阶段之间的寄存器，将取指阶段的结果在下一个时钟周期传递到译码阶段。

（2）模块接口

信号名	方向	描述
clk	I	时钟信号
rst	I	复位信号
flush	I	清空流水线
stall	I	阻塞流水线
pc_in [31:0]	I	取指阶段取得的指令对应的地址
instr_in [31:0]	I	取指阶段取得的指令
pc_out [31:0]	O	译码阶段的指令对应的地址
instr_out [31:0]	O	译码阶段的指令

3.11 ID/EX 段间寄存器（IDEX）

（1）功能描述

IDEX：实现译码阶段与执行阶段之间的寄存器，将译码阶段的结果在下一个时钟周期传递到执行阶段。

（2）模块接口

信号名	方向	描述
clk	I	时钟信号
rst	I	复位信号
flush	I	清空流水线
instr_in [31:0]	I	译码阶段的指令
extData_in [31:0]	I	译码阶段的扩展

data1_in [31:0]	I	源操作数 1
data2_in [31:0]	I	源操作数 2
RegAddrI_in [4:0]	I	译码阶段的 rt
RegAddrR_in [4:0]	I	译码阶段的 rd
RegDst_in	I	译码阶段写入目标寄存器的信号
Aluop_in [4:0]	I	译码阶段的指令要进行的运算的子类型
Alusrc_in [4:0]	I	译码阶段的指令要进行的第二个操作数的选择来源
MemRead_in	I	译码阶段的存储器读信号
MemWrite_in	I	译码阶段的存储器写信号
RegWrite_in	I	译码阶段的寄存器写信号
MemToReg	I	译码阶段的数据存储器到寄存器的信号
instr_out [31:0]	O	执行阶段的指令
extData_out [31:0]	O	执行阶段的扩展结果
data1_out [31:0]	O	执行阶段的操作数 1
data2_out [31:0]	O	执行阶段的操作数 2
RegAddrI_out [4:0]	O	执行阶段的 rt
RegAddrR_out [4:0]	O	执行阶段的 rd
RegRst_out	O	执行阶段写入目标寄存器的信号
Aluop_out [4:0]	O	执行阶段的指令要进行的运算的子类型
Alusrc_out [1:0]	O	执行阶段的指令要进行的第二个操作数的选择来源
MemRead_out	O	执行阶段的存储器读信号
MemWrite_out	O	执行阶段的存储器写信号
RegWrite_out	O	执行阶段的寄存器写信号
MemToReg_out	O	执行阶段的数据存储器到寄存器的信号

3.12 EX/MEM 段间寄存器 (EXMEM)

(1) 功能描述

EXMEM：实现执行阶段与访存阶段之间的寄存器，将执行阶段的结果在下一个时钟周期传递到访存阶段。

(2) 模块接口

信号名	方向	描述
clk	I	时钟信号
rst	I	复位信号
ALUresult_in [31:0]	I	执行阶段 ALU 计算的结果
zero_in	I	执行阶段 zero 的结果
rt_in [31:0]	I	执行阶段的源操作数 2
RegDst_in	I	执行阶段的目标寄存器
RegAddrI_in [4:0]	I	执行阶段的 rt 地址
RegAddrR_in[4:0]	I	执行阶段的 rd 地址
MemRead_in	I	执行阶段的存储器读信号
MemWrite_in	I	执行阶段的存储器写信号
RegWrite_in	I	执行阶段的寄存器写信号
MemToReg_in	I	执行阶段的数据存储器到寄存器的信号
ALUresult_out [31:0]	O	访存阶段 ALU 的结果
zero_out	O	访存阶段 zero 的结果
rt_out [31:0]	O	访存阶段的源操作数 2
RegDst_out	O	访存阶段的目标寄存器
RegAddrI_out [4:0]	O	访存阶段的 rt
RegAddrR_out [4:0]	O	访存阶段的 rd
MemRead_out	O	访存阶段的存储器读信号
MemWrite_out	O	访存阶段的存储器写信号

RegWrite_out	O	访存阶段的寄存器写信号
MemToReg_out	O	访存阶段的数据存储器到寄存器的信号

3.13 MEM/WB 段间寄存器 (MEMWB)

(1) 功能描述

MEMWB: 实现访存阶段与回写阶段之间的寄存器, 将访存阶段的结果在下一个时钟周期传递到回写阶段。

(2) 模块接口

信号名	方向	描述
clk	I	时钟信号
rst	I	复位信号
ALUresult_in [31:0]	I	访存阶段 ALU 的结果
mem_in [31:0]	I	访存阶段的写入数值
RegDst_in	I	访存阶段的目标寄存器
RegAddrI_in [4:0]	I	访存阶段的 rt
RegAddrR_in [4:0]	I	访存阶段的 rd
RegWrite_in	I	访存阶段的写入寄存器信号
MemToReg_in	I	访存阶段的数据存储器到寄存器的信号
ALUresult_out [31:0]	O	写回阶段 ALU 的结果
mem_out [31:0]	O	写回阶段的写入数值
RegDst_out	O	写回阶段的目标寄存器
RegAddrI_out [4:0]	O	写回阶段的 rt
RegAddrR_out [4:0]	O	写回阶段的 rd
RegWrite_out	O	写回阶段的写入寄存器信号
MemToReg_out	O	写回阶段的数据存储器到寄存器的信号

3.14 冒险检测单元 (Hazard)

(1) 功能描述

Hazard 用于检测数据冒险，结构冒险实现数据冒险转发功能，bne,beq 在 ID 级的跳转、j 在 ID 级的跳转，lw 指令在 ID 级阻塞流水线。

(2) 模块接口

信号名	方向	描述
EXMEM_RegWrite	I	执行阶段的寄存器写信号
EXMEM_rd [4:0]	I	EX 到 MEM 级的 rd
MEMWB_rd [4:0]	I	MEM 到 WB 的 rd
MEMWB_RegWrite	I	访存阶段的寄存器写信号
IDEX_rs [4:0]	I	ID 到 EX 级的 rs
IDEX_rt [4:0]	I	ID 到 EX 级的 rt
IDEX_MemRead	I	译码阶段的存储器读信号
IFID_rs [4:0]	I	IF 到 ID 级的 rs
IFID_rt [4:0]	I	IF 到 ID 级的 rt
pcsel	I	左移 2 位信号
jump	I	j 跳转信号
branch	I	分支指令跳转信号
EXMEM_MemRead	I	执行阶段的读存储器信号
EXMEM_rt [4:0]	I	EX 到 MEM 的 rt
IDEX_rd [4:0]	I	ID 到 EX 的 rd
IDEX_RegWrite	I	译码阶段的寄存器写信号
ForwardA [1:0]	O	rd-rs 旁路信号
ForwardB [1:0]	O	rd-rt 旁路信号
stallID	O	阻塞 ID 级
stallPC	O	阻塞 PC

flushIFID	O	清空 IFID 级流水线
flushIDEX	O	清空 IDEX 级流水线
compareSrc1	O	分支指令 ID 级跳转 rd-rs 判断
compareSrc2	O	分支指令 ID 级跳转 rd-rt 判断

3.15 多路选择器 (mux)

(1) 功能描述

mux 主要功能是多路选择器。mux.v 文件包含二选一、四选一多路选择器。

(2) 模块接口

信号名	方向	描述
d0,d1,d2	I	供选择数据
s	I	片选信号
y	O	片选后的数据

3.16 宏定义文件

3.16.1 instruction_def.v

定义指令的 OPCODE/FUNCT 域

3.16.2 ctrl_encode_def.v

相关控制信号的宏。

3.17 mips_tb.v

激励文件。初始化时钟周期等信号。

4 详细设计

4.1 程序计数器 (PcUnit)

PcUnit 记录下一条指令的地址

复位时, PC 回到起点, $PC = 32'h0000_3000$;

顺序时, $PC=PC+4$;

分支指令时, $PC=PC+4$,再左移 2 位

Jump 时, $PC=PC$ 高四位拼接指令存储器[25:0]拼接 2 个 0。

module PcUnit(PC,OldPC,stall,PcReSet,PcSel,Clk,Adress,Adj,j);

```
input    [31:0] OldPC;
```

```
input    stall;
```

```
input    PcReSet;
```

```
input    PcSel;
```

```
input    Clk;
```

```
input    [31:0] Adress;
```

```
input    [25:0] Adj;
```

```
input    j;
```

```
output reg[31:0] PC;
```

```
integer i;
```

```
reg [31:0] temp;
```

```
always@(posedge Clk or posedge PcReSet)
```

```
begin
```

```
    if(PcReSet == 1)
```

```
        PC <= 32'h0000_3000;
```

```
    if(stall==0)
```

```
        begin
```

```
            PC=PC+4;
```

```
            if(PcSel == 1)
```

```

        begin
            for(i=0;i<30;i=i+1)
                temp[31-i] = Adress[29-i];
            temp[0] = 0;
            temp[1] = 0;

            PC = OldPC+temp;
        end
    if(j==1)
        begin
            PC={OldPC[31:28],Adj[25:0],2'b00};
        end
    end
end
endmodule

```

4.2 寄存器（GPR）

写使能信号有效时，将数据写入。

```

module GPR(DataOut1,DataOut2,clk,WData,WE,WeSel,ReSel1,ReSel2);
    input clk;
    input WE;
    input [4:0] WeSel,ReSel1,ReSel2;
    input [31:0] WData;

    output [31:0] DataOut1,DataOut2;

    reg [31:0] Gpr[31:0];

```

```

always@(negedge clk)
begin
    if(WE == 1)
        Gpr[WeSel] <= WData;

        $display("R[00-07]=%8X, %8X, %8X, %8X, %8X, %8X, %8X, %8X", 0,
Gpr[1], Gpr[2], Gpr[3], Gpr[4], Gpr[5], Gpr[6], Gpr[7]);

        $display("R[08-15]=%8X, %8X, %8X, %8X, %8X, %8X, %8X, %8X",
Gpr[8], Gpr[9], Gpr[10], Gpr[11], Gpr[12], Gpr[13], Gpr[14], Gpr[15]);

        $display("R[16-23]=%8X, %8X, %8X, %8X, %8X, %8X, %8X, %8X",
Gpr[16], Gpr[17], Gpr[18], Gpr[19], Gpr[20], Gpr[21], Gpr[22], Gpr[23]);

        $display("R[24-31]=%8X, %8X, %8X, %8X, %8X, %8X, %8X, %8X",
Gpr[24], Gpr[25], Gpr[26], Gpr[27], Gpr[28], Gpr[29], Gpr[30], Gpr[31]);

        $display("R[%4X]=%8X", WeSel, Gpr[WeSel]);
    end

    assign DataOut1 = (ReSel1==0)?0:Gpr[ReSel1];
    assign DataOut2 = (ReSel2==0)?0:Gpr[ReSel2];

endmodule

```

4.3 算术逻辑运算单元（ALU）

根据 ALUctrl 决定应该进行哪种运算。包括: addu, add, sub, subu, or, and, slt, sll, srl, beq, bne, lui

```

`include "ctrl_encode_def.v"

module Alu(AluResult, Zero, DataIn1, DataIn2, AluCtrl, shame);

```

```

    input  [31:0]      DataIn1;    //运算数据 1
    input  [31:0]      DataIn2;    //运算数据 2

```

```

input  [4:0]    AluCtrl;    //运算器控制信号
input  [4:0]    shame;

output reg[31:0]    AluResult;    //运算器输出结果
output reg          Zero;        //结果是否为零

initial                                //初始化数据
begin
    Zero = 0;
    AluResult = 0;
end

always@(DataIn1 or DataIn2 or AluCtrl)
begin
    case ( AluCtrl )
        `ALUOp_ADDU: AluResult = DataIn1 + DataIn2;
        `ALUOp_SUBU: AluResult = DataIn1 - DataIn2;
        `ALUOp_AND: AluResult = DataIn1 & DataIn2;
        `ALUOp_OR: AluResult = DataIn1 | DataIn2;
        `ALUOp_ADD: AluResult = DataIn1 + DataIn2;
        `ALUOp_SUB: AluResult = DataIn1 +1+~DataIn2;
        `ALUOp_SLT: AluResult = (DataIn1 +1+~DataIn2)>>31;
        `ALUOp_SLL: AluResult = DataIn2 << shame;
        `ALUOp_SRL: AluResult = DataIn2 >> shame;
        `ALUOp_EQL: AluResult = (DataIn1 == DataIn2) ? 1 : 0;
        `ALUOp_BNE: AluResult = (DataIn1 != DataIn2)? 32'b1 : 32'b0;
        `ALUOp_LUI: AluResult = {DataIn2[15: 0], 16'd0};
        default:    ;
    endcase

    if(AluResult == 0)

```



```

        Zero = 1;
    else
        Zero = 0;
    end

endmodule

```

4.4 扩展单元 (Extender)

扩展有三种：

0 扩展；符号扩展，将立即数扩展到高位

```
`include "ctrl_encode_def.v"
```

```
module Extender(ExtOut,DataIn,ExtOp);
```

```
    input [15:0] DataIn;
```

```
    input [1:0] ExtOp;
```

```
    output reg[31:0] ExtOut;
```

```
    integer i;                //逻辑计数
```

```
    always@(DataIn or ExtOp)
```

```
    begin
```

```
        case (ExtOp)
```

```
            `EXT_ZERO:    ExtOut = {16'd0, DataIn};
```

```
            `EXT_SIGNED:  ExtOut = {{16{DataIn[15]}}, DataIn};
```

```
            `EXT_HIGHPOS: ExtOut = {DataIn, 16'd0};
```

```
            default: ;
```

```
        endcase
```

```
    end
```

```
endmodule
```

4.5 数据存储器 (DMem)

将数据传到数据存储器中

```
module DMem(DataOut,DataAdr,DataIn,DMemW,DMemR,clk);
```

```
    input [4:0] DataAdr;
```

```
    input [31:0] DataIn;
```

```
    input      DMemR;
```

```
    input      DMemW;
```

```
    input      clk;
```

```
    output[31:0] DataOut;
```

```
    reg [31:0]  DMem[1023:0];
```

```
    always@(posedge clk)
```

```
    begin
```

```
        if(DMemW)
```

```
            DMem[DataAdr] <= DataIn;
```

```
    end
```

```
    assign DataOut = DMem[DataAdr];
```

```
endmodule
```

4.6 指令存储器 (IM)

存储指令

```
module IM(OpCode,ImAdress);
```

```
    input [4:0] ImAdress;
```

```

output [31:0]  OpCode;

reg[31:0] Opcode;

reg [31:0]  IMem[1024:0];

always@(ImAdress)
begin
    Opcode = IMem[ImAdress];
end

assign OpCode = Opcode;
endmodule

```

4.7 控制器 (Ctrl)

控制信号：

信号名称	0	1
jump	无	无分支指令时跳转
RegDst	写入寄存器的目标 编号来自 rt 字段 (20:16)	写入寄存器的目标 编号来自 rd 字段 (15:11)
Branch	无	分支指令时跳转
MemR	无	输入地址对应的数 据内存的内容放置 到读出数据的输出
Mem2R	ALU 提供寄存器写 数据的输入值	数据内存提供寄存 器写数据的输入值
RegW	无	将写入数据的输入 写入至寄存器输入 对应的寄存器

AluSrc	第二个 ALU 操作数 来自第二个寄存器 堆的输出	第二个 ALU 操作数 为已符号化扩展的 指令的低 16 位
--------	---------------------------------	--------------------------------------

ExtOp[1:0]见 Extender

AluCtrl[4:0]见 ALU

```
`include "ctrl_encode_def.v"
```

```
`include "instruction_def.v"
```

```
module
```

```
Ctrl(jump,RegDst,Branch,MemR,Mem2R,MemW,RegW,Alusrc,ExtOp,Aluctrl,O  
pCode,funct);
```

```
    input [5:0]      OpCode;
```

```
    input [5:0]      funct;
```

```
    output reg jump;
```

```
    output reg RegDst;
```

```
    output reg Branch;
```

```
    output reg MemR;
```

```
    output reg Mem2R;
```

```
    output reg MemW;
```

```
    output reg RegW;
```

```
    output reg Alusrc;
```

```
    output reg [1:0] ExtOp;
```

```
    output reg [4:0] Aluctrl;
```

```
    always@(OpCode or funct)
```

```
    begin
```

```

case(OpCode)
  `INSTR_RTYPE_OP: //R type 000000
  begin
    assign jump=0;
    assign Branch=0;
    assign Mem2R=0;
    assign MemW=0;
    assign MemR=0;
    assign Alusrc=0;
    assign ExtOp=`EXT_ZERO;
    assign RegDst=1;
    case(funcnt)

      `INSTR_ADDU_FUNCT: //ADDU
      begin
        assign RegW=1;
        assign Aluctrl=`ALUOp_ADDU;
      end

      `INSTR_SUBU_FUNCT: //SUBU
      begin
        assign RegW=1;
        assign Aluctrl=`ALUOp_SUBU;
      end

      `INSTR_ADD_FUNCT: //ADD
      begin
        assign RegW=1;
        assign Aluctrl=`ALUOp_ADD;

```

end

`INSTR_SUB_FUNCT: //SUB

begin

assign RegW=1;

assign Aluctrl=`ALUOp_SUB;

end

`INSTR_OR_FUNCT: //OR

begin

assign RegW=1;

assign Aluctrl=`ALUOp_OR;

end

`INSTR_AND_FUNCT: //AND

begin

assign RegW=1;

assign Aluctrl=`ALUOp_AND;

end

`INSTR_SLT_FUNCT: //SLT

begin

assign RegW=1;

assign Aluctrl=`ALUOp_SLT;

end

`INSTR_SLL_FUNCT: //SLL

begin

assign RegW=1;

assign Aluctrl=`ALUOp_SLL;

```

        end

        `INSTR_SRL_FUNCT: //SRL
        begin
            assign RegW=1;
            assign Aluctrl=`ALUOp_SRL;
        end

        default;;
    endcase
end

```

```

`INSTR_ORI_OP: //ORI
begin
    assign jump=0;
    assign Branch=0;
    assign Mem2R=0;
    assign MemW=0;
    assign MemR=0;
    assign Alusrc=1;
    assign ExtOp=`EXT_ZERO;
    assign RegDst=0;
    assign RegW=1;
    assign Aluctrl=`ALUOp_OR;
end

```

```

`INSTR_ADDI_OP: //ADDI
begin
    assign jump=0;
    assign Branch=0;

```

```

    assign Mem2R=0;
    assign MemW=0;
    assign MemR=0;
    assign Alusrc=1;
    assign ExtOp=`EXT_ZERO;
    assign RegDst=0;
    assign RegW=1;
    assign Aluctrl=`ALUOp_ADD;
end

```

```

`INSTR_LUI_OP: //LUI
begin
    assign jump=0;
    assign Branch=0;
    assign Mem2R=0;
    assign MemW=0;
    assign MemR=0;
    assign Alusrc=1;
    assign ExtOp=`EXT_SIGNED;
    assign RegDst=0;
    assign RegW=1;
    assign Aluctrl=`ALUOp_LUI;
end

```

```

`INSTR_SW_OP: //SW
begin
    assign jump=0;
    assign Branch=0;
    assign Mem2R=0;
    assign MemW=1;

```



```

    assign MemR=0;
    assign Alusrc=1;
    assign ExtOp=`EXT_SIGNED;
    assign RegDst=0;
    assign RegW=0;
    assign AluCtrl=`ALUOp_ADD;
end

```

```

`INSTR_LW_OP: //LW
begin
    assign jump=0;
    assign Branch=0;
    assign Mem2R=1;
    assign MemW=0;
    assign MemR=1;
    assign Alusrc=1;
    assign ExtOp=`EXT_SIGNED;
    assign RegDst=0;
    assign RegW=1;
    assign AluCtrl=`ALUOp_ADD;
end

```

```

`INSTR_BEQ_OP: //BEQ
begin
    assign jump=0;
    assign Branch=1;
    assign Mem2R=0;
    assign MemW=0;
    assign MemR=0;
    assign Alusrc=0;

```

```

    assign ExtOp=`EXT_SIGNED;
    assign RegDst=0;
    assign RegW=0;
    assign Aluctrl=`ALUOp_EQL;
end

```

```

`INSTR_BNE_OP: //BNE
begin
    assign jump=0;
    assign Branch=1;
    assign Mem2R=0;
    assign MemW=0;
    assign MemR=0;
    assign Alusrc=0;
    assign ExtOp=`EXT_SIGNED;
    assign RegDst=0;
    assign RegW=0;
    assign Aluctrl=`ALUOp_BNE;
end

```

```

`INSTR_J_OP: //J
begin
    assign jump=1;
    assign Branch=0;
    assign Mem2R=0;
    assign MemW=0;
    assign MemR=0;
    assign Alusrc=0;
    assign ExtOp=`EXT_SIGNED;
    assign RegDst=0;

```

```

        assign RegW=0;
        assign Aluctrl=`ALUOp_SUB;
    end

    endcase

end

endmodule

```

4.8 模型机（MIPS）

连接各模块

```

`include "instruction_def.v"
module Mips(Clk,Reset);

    input Clk;
    input Reset;

//PC
    wire [31:0] pcOut;

//IM
    wire [4:0] imAdr;
    wire [31:0] opCode;

//GPR
    wire [4:0] gprWeSel,gprReSel1,gprReSel2;
    wire [31:0] gprDataIn;

    wire [31:0] gprDataOut1,gprDataOut2;
    wire Equal;

```

```
//Extender
```

```
wire [15:0] extDataIn;  
wire [31:0] extDataOut;
```

```
//DMem
```

```
wire [4:0]  dmDataAdr;  
wire [31:0] dmDataOut;
```

```
//Ctrl
```

```
wire [5:0]      op;  
wire [5:0]      funct;  
wire           jump;           //指令跳转  
wire           RegDst;  
wire           Branch;        //分支  
wire           MemR;          //读存储器  
wire           Mem2R;          //数据存储器到寄存器堆  
wire           MemW;          //写数据存储器  
wire           RegW;          //寄存器堆写入数据  
wire           Alusrc;        //运算器操作数选择  
wire [1:0]      ExtOp;         //位扩展/符号扩展选择  
wire [4:0]      Aluctrl;       //Alu 运算选择
```

```
//Alu
```

```
wire [31:0] aluDataIn2;  
wire [31:0] aluDataOut;  
wire       zero;
```

//IFID

```
wire [31:0] pc_in_IFID;  
wire [31:0] instr_in_IFID;  
wire [31:0] pc_out_IFID;  
wire [31:0] instr_out_IFID;
```

//IDEX

```
wire [31:0] instr_in_IDEX;  
wire [31:0] extData_in_IDEX;  
wire [31:0] data1_in_IDEX;  
wire [31:0] data2_in_IDEX;  
wire [4:0] RegAddrI_in_IDEX;  
wire [4:0] RegAddrR_in_IDEX;  
wire RegDst_in_IDEX;  
wire [4:0] ALUOp_in_IDEX;  
wire [1:0] ALUsrc_in_IDEX;  
wire Branch_in_IDEX;  
wire MemRead_in_IDEX;  
wire MemWrite_in_IDEX;  
wire RegWrite_in_IDEX;  
wire MemToReg_in_IDEX;  
wire [31:0] instr_out_IDEX;  
wire [31:0] extData_out_IDEX;  
wire [31:0] data1_out_IDEX;  
wire [31:0] data2_out_IDEX;  
wire [4:0] RegAddrI_out_IDEX;  
wire [4:0] RegAddrR_out_IDEX;  
wire RegDst_out_IDEX;  
wire [4:0] ALUOp_out_IDEX;  
wire [1:0] ALUsrc_out_IDEX;
```

```

wire Branch_out_IDEX;
wire MemRead_out_IDEX;
wire MemWrite_out_IDEX;
wire RegWrite_out_IDEX;
wire MemToReg_out_IDEX;

//EXMEM

wire [31:0] ALUresult_in_EXMEM;
wire zero_in_EXMEM;
wire [31:0] rt_in_EXMEM;
wire RegDst_in_EXMEM;
wire [4:0] RegAddrI_in_EXMEM;
wire [4:0] RegAddrR_in_EXMEM;
wire Branch_in_EXMEM;
wire MemRead_in_EXMEM;
wire MemWrite_in_EXMEM;
wire RegWrite_in_EXMEM;
wire MemToReg_in_EXMEM;
wire [31:0] ALUresult_out_EXMEM;
wire zero_out_EXMEM;
wire [31:0] rt_out_EXMEM;
wire RegDst_out_EXMEM;
wire [4:0] RegAddrI_out_EXMEM;
wire [4:0] RegAddrR_out_EXMEM;
wire Branch_out_EXMEM;
wire MemRead_out_EXMEM;
wire MemWrite_out_EXMEM;
wire RegWrite_out_EXMEM;
wire MemToReg_out_EXMEM;

```

```
//MEMWB
```

```
wire [31:0] ALUresult_in_MEMWB;  
wire [31:0] mem_in_MEMWB;  
wire RegDst_in_MEMWB;  
wire [4:0] RegAddrI_in_MEMWB;  
wire [4:0] RegAddrR_in_MEMWB;  
wire RegWrite_in_MEMWB;  
wire MemToReg_in_MEMWB;  
wire [31:0] ALUresult_out_MEMWB;  
wire [31:0] mem_out_MEMWB;  
wire RegDst_out_MEMWB_MEMWB;  
wire [4:0] RegAddrI_out_MEMWB;  
wire [4:0] RegAddrR_out_MEMWB;  
wire RegWrite_out_MEMWB;  
wire MemToReg_out_MEMWB;
```

```
//forwarding
```

```
wire [31:0] ForwardData1;  
wire [31:0] ForwardData2;  
  
wire [4:0] IDEX_rd_in;  
wire [4:0] EXMEM_rd_in;  
wire [4:0] MEMWB_rd_in;  
wire [1:0] ForwardA_out;  
wire [1:0] ForwardB_out;  
wire stallID;  
wire stallPC;  
wire flushIFID;  
wire flushIDEX;  
wire compareSrc1;
```

```

wire compareSrc2;

wire [31:0] compareData1;

wire [31:0] compareData2;

//PC 块实例化

assign                                pcSel                                =
((Branch&&((op=='INSTR_BEQ_OP)?Equal:!Equal))==1)?1:0;

PcUnit

U_pcUnit(.PC(pcOut),.OldPC(pc_out_IFID),.stall(stallPC),.PcReSet(Reset),.PcSel
l(pcSel),.Clk(Clk),.Adress(extDataOut),.Adj(instr_out_IFID[25:0]),.j(jump));

assign imAdr = pcOut[6:2];

//指令寄存器实例化

IM U_IM(.OpCode(opCode),.ImAdress(imAdr));


//IFID

assign pc_in_IFID=pcOut+4;

assign instr_in_IFID=opCode;

IFID

U_IFID(.clk(Clk),.rst(Reset),.flush(flushIFID),.pc_in(pc_in_IFID),.instr_in(instr_i
n_IFID),

.pc_out(pc_out_IFID),.instr_out(instr_out_IFID),.stall(stallID));


//寄存器堆实例化

assign gprReSel1 = instr_out_IFID[25:21];

```



```

    assign gprReSel2 = instr_out_IFID[20:16];

    GPR
    U_gpr(.DataOut1(gprDataOut1),.DataOut2(gprDataOut2),.clk(Clk),.WData(gprDataIn)

        ,.WE(RegWrite_out_MEMWB),.WeSel(gprWeSel),.ReSel1(gprReSel1),.ReSel2(gprReSel2));

    mux2compare1
    U_mux2compare1(.d0(gprDataOut1),.d1(ALUresult_out_EXMEM),.s(compareSrc1),.y(compareData1));

    mux2compare2
    U_mux2compare2(.d0(gprDataOut2),.d1(ALUresult_out_EXMEM),.s(compareSrc2),.y(compareData2));

    assign Equal=    (compareData1==compareData2);

//控制器实例化

    assign op = instr_out_IFID[31:26];
    assign funct = instr_out_IFID[5:0];

    Ctrl
    U_Ctrl(.jump(jump),.RegDst(RegDst),.Branch(Branch),.MemR(MemR),.Mem2R(Mem2R)

        ,.MemW(MemW),.RegW(RegW),.Alusrc(Alusrc),.ExtOp(ExtOp),.Aluctrl(Aluctrl)

        ,.OpCode(op),.funct(funct));

//扩展器实例化

    assign extDataIn = instr_out_IFID[15:0];

    Extender U_extend(.ExtOut(extDataOut),.DataIn(extDataIn),.ExtOp(ExtOp));

```

```
//IDEX
```

```
assign pc_in_IDEX=pc_out_IFID;
assign instr_in_IDEX=instr_out_IFID;
assign extData_in_IDEX=extDataOut;
assign data1_in_IDEX=gprDataOut1;
assign data2_in_IDEX=gprDataOut2;
assign RegAddrI_in_IDEX=instr_out_IFID[20:16];
assign RegAddrR_in_IDEX=instr_out_IFID[15:11];
assign RegDst_in_IDEX=RegDst;
assign ALUOp_in_IDEX=Aluctrl;
assign ALUsrc_in_IDEX=Alusrc;
assign MemRead_in_IDEX=MemR;
assign MemWrite_in_IDEX=MemW;
assign RegWrite_in_IDEX=RegW;
assign MemToReg_in_IDEX=Mem2R;
```

```
IDEX
```

```
U_IDEX(.clk(Clk),.rst(Reset),.instr_in(instr_in_IDEX),.flush(flushIDEX),.extData_in(extData_in_IDEX),.data1_in(data1_in_IDEX),
        .data2_in(data2_in_IDEX),.RegAddrI_in(RegAddrI_in_IDEX),.RegAddrR_in(RegAddrR_in_IDEX),
        .RegDst_in(RegDst_in_IDEX),.ALUOp_in(ALUOp_in_IDEX),.ALUsrc_in(ALUsrc_in_IDEX),
        .MemRead_in(MemRead_in_IDEX),.MemWrite_in(MemWrite_in_IDEX),.RegWrite_in(RegWrite_in_IDEX),
        .MemToReg_in(MemToReg_in_IDEX), .instr_out(instr_out_IDEX),.extData_out(extData_out_IDEX),
        .data1_out(data1_out_IDEX),.data2_out(data2_out_IDEX),.RegAddrI_out(RegAddrI_out_IDEX),.RegAddrR_out(RegAddrR_out_IDEX),
        .RegDst_out(RegDst_out_IDEX),.ALUOp_out(ALUOp_out_IDEX),.ALUs
```

```

rc_out(ALUsrc_out_IDEX),
    .MemRead_out(MemRead_out_IDEX),.MemWrite_out(MemWrite_out_I
DEX),
    .RegWrite_out(RegWrite_out_IDEX),.MemToReg_out(MemToReg_out_I
DEX));

```

```

mux4alu1
U_mux4alu1(.d0(data1_out_IDEX),.d1(gprDataIn),.d2(ALUresult_out_EXMEM)
,.s(ForwardA_out),.y(ForwardData1));
mux4alu2
U_mux4alu2(.d0(data2_out_IDEX),.d1(gprDataIn),.d2(ALUresult_out_EXMEM)
,.s(ForwardB_out),.y(ForwardData2));

```

```

assign
aluDataIn2=(ALUsrc_out_IDEX==1)?extData_out_IDEX:ForwardData2;
//ALU 实例化
Alu
U_Alu(.AluResult(aluDataOut),.Zero(zero),.DataIn1(ForwardData1),.DataIn2(alu
DataIn2),
    .AluCtrl(ALUOp_out_IDEX),.shame(instr_out_IDEX[10:6]));

```

```

//EXMEM
assign ALUresult_in_EXMEM=aluDataOut;

assign rt_in_EXMEM=ForwardData2;
assign RegDst_in_EXMEM=RegDst_out_IDEX;

```

```

assign RegAddrI_in_EXMEM=RegAddrI_out_IDEX;
assign RegAddrR_in_EXMEM=RegAddrR_out_IDEX;
assign MemRead_in_EXMEM=MemRead_out_IDEX;
assign MemWrite_in_EXMEM=MemWrite_out_IDEX;
assign RegWrite_in_EXMEM=RegWrite_out_IDEX;
assign MemToReg_in_EXMEM=MemToReg_out_IDEX;
assign ALUresult_in_EXMEM=aluDataOut;

```

EXMEM

```

U_EXMEM(.clk(Clk),.rst(Reset) ,.ALUresult_in(ALUresult_in_EXMEM),.zero_in(
zero_in_EXMEM),
    .rt_in(rt_in_EXMEM),.RegDst_in(RegDst_in_EXMEM),.RegAddrI_in(Reg
AddrI_in_EXMEM),.RegAddrR_in(RegAddrR_in_EXMEM),
    .MemRead_in(MemRead_in_EXMEM),.MemWrite_in(MemWrite_in_EXM
EM),.RegWrite_in(RegWrite_in_EXMEM),
    .MemToReg_in(MemToReg_in_EXMEM),.ALUresult_out(ALUresult_out_
EXMEM),.zero_out(zero_out_EXMEM),.rt_out(rt_out_EXMEM)
    ,.RegDst_out(RegDst_out_EXMEM),.RegAddrI_out(RegAddrI_out_EXME
M),.RegAddrR_out(RegAddrR_out_EXMEM),
    .MemRead_out(MemRead_out_EXMEM),.MemWrite_out(MemWrite_out_
EXMEM),
    .RegWrite_out(RegWrite_out_EXMEM),.MemToReg_out(MemToReg_out_
EXMEM));

```

//DM 实例化

```

assign dmDataAdr = ALUresult_out_EXMEM[4:0];

```

DMem

```

U_Dmem(.DataOut(dmDataOut),.DataAdr(dmDataAdr),.DataIn(rt_out_EXMEM)
,.DMemW(MemWrite_out_EXMEM),DMemR(MemRead_out_EXMEM),clk(C
lk));

```

```

//MEMWB

```

```

    assign ALUresult_in_MEMWB=ALUresult_out_EXMEM;
    assign mem_in_MEMWB=dmDataOut;
    assign RegDst_in_MEMWB=RegDst_out_EXMEM;
    assign RegAddrI_in_MEMWB=RegAddrI_out_EXMEM;
    assign RegAddrR_in_MEMWB=RegAddrR_out_EXMEM;
    assign RegWrite_in_MEMWB=RegWrite_out_EXMEM;
    assign MemToReg_in_MEMWB=MemToReg_out_EXMEM;

```

```

MEMWB

```

```

U_MEMWB(.clk(Clk),.rst(Reset), .ALUresult_in(ALUresult_in_MEMWB) ,me
m_in(mem_in_MEMWB),RegDst_in(RegDst_in_MEMWB),

```

```

    .RegAddrI_in(RegAddrI_in_MEMWB),.RegAddrR_in(RegAddrR_in_MEMW
B),.RegWrite_in(RegWrite_in_MEMWB),.MemToReg_in(MemToReg_in_MEM
WB),

```

```

    .ALUresult_out(ALUresult_out_MEMWB),.mem_out(mem_out_MEMWB),.R
egDst_out(RegDst_out_MEMWB),.RegAddrI_out(RegAddrI_out_MEMWB),

```

```

    .RegAddrR_out(RegAddrR_out_MEMWB),.RegWrite_out(RegWrite_out_ME
MWB),.MemToReg_out(MemToReg_out_MEMWB));

```

```

    assign                                gprDataIn                                =
(MemToReg_out_MEMWB==1)?mem_out_MEMWB:ALUresult_out_MEMWB
;

```

```

    assign                                gprWeSel                                =
(RegDst_out_MEMWB==1)?RegAddrR_out_MEMWB:RegAddrI_out_MEMW
B;

```

```

//Hazard

    assign
    IDEX_rd_in=(RegDst_out_IDEX==0)?RegAddrI_out_IDEX:RegAddrR_out_ID
    EX;

    assign
    EXMEM_rd_in=(RegDst_out_EXMEM==0)?RegAddrI_out_EXMEM:RegAddr
    R_out_EXMEM;

    assign
    MEMWB_rd_in=(RegDst_out_MEMWB==0)?RegAddrI_out_MEMWB:RegAdd
    rR_out_MEMWB;

    Hazard
    U_DepAndForw(.EXMEM_RegWrite(RegWrite_out_EXMEM),.EXMEM_rd(E
    XMEM_rd_in),.IDEX_rs(instr_out_IDEX[25:21]),
        .IDEX_rt(instr_out_IDEX[20:16]),.MEMWB_RegWrite(RegWrite_out_MEM
    WB),.MEMWB_rd(MEMWB_rd_in),.ForwardA(ForwardA_out),
        .ForwardB(ForwardB_out),.IDEX_MemRead(MemRead_out_IDEX),.IFID_rs(
    instr_out_IFID[25:21]),
        .IFID_rt(instr_out_IFID[20:16]),.stallID(stallID),.stallPC(stallPC),.flushIFID(fl
    ushIFID),.flushIDEX(flushIDEX),
        .pcsel(pcSel),.jump(jump),.branch(Branch),.EXMEM_MemRead(MemRead_o
    ut_EXMEM),
        .compareSrc1(compareSrc1),.compareSrc2(compareSrc2),.IDEX_rd(IDEX_rd_
    in),.IDEX_RegWrite(RegWrite_out_IDEX),
        .EXMEM_rt(rt_out_EXMEM[20:16]));

endmodule

```

4.9 IFID 段间寄存器 (IFID)

如果没有 flush 或者 stall, 将 IF 级地址和指令传递给 ID 级

```
module IFID (clk, rst, flush, stall, pc_in ,instr_in, pc_out, instr_out);
```

```
    input      clk;
    input      rst;
    input      flush;
    input      stall;
    input  [31:0] pc_in;
    input  [31:0] instr_in;
    output [31:0] pc_out;
    output [31:0] instr_out;
```

```
    reg [31:0] instr;
```

```
    reg [31:0] pc;
```

```
    always @(posedge clk or posedge rst) begin
```

```
        if ( rst )
```

```
            begin
```

```
                instr <= 0;
```

```
                pc <= 0;
```

```
            end
```

```
        else if(stall==1)
```

```
            begin
```

```
                instr <= instr;
```

```
                pc <= pc;
```

```
            end
```

```
        else if(flush==1)
```

```

        instr <= 0;
    else
        begin
            instr <= instr_in;
            pc <= pc_in;
        end

    end

    assign pc_out=pc;
    assign instr_out=instr;

endmodule

```

4.10 IDEX 段间寄存器 (IDEX)

将 ID 级结果传递给 EX 级

```

module IDEX
(
    clk,rst,flush,instr_in,extData_in,data1_in,data2_in,RegAddrI_in,RegAddrR_in,RegDst_in,ALUOp_in,ALUSrc_in,MemRead_in,MemWrite_in,RegWrite_in,MemToReg_in,instr_out,extData_out,data1_out,data2_out,RegAddrI_out,RegAddrR_out,RegDst_out,ALUOp_out,ALUSrc_out,MemRead_out,MemWrite_out,RegWrite_out,MemToReg_out);

```

```

    input      clk;
    input      rst;
    input      flush;
    input [31:0] instr_in;
    input [31:0] extData_in;
    input [31:0] data1_in;
    input [31:0] data2_in;
    input [4:0] RegAddrI_in;

```



```

input  [4:0] RegAddrR_in;
input  RegDst_in;
input  [4:0] ALUOp_in;
input  [1:0] ALUsrc_in;
input  MemRead_in;
input  MemWrite_in;
input  RegWrite_in;
input  MemToReg_in;
output [31:0] instr_out;
input  [31:0] extData_out;
output [31:0] data1_out;
output [31:0] data2_out;
output [4:0] RegAddrI_out;
output [4:0] RegAddrR_out;
output RegDst_out;
output [4:0] ALUOp_out;
output [1:0] ALUsrc_out;
output MemRead_out;
output MemWrite_out;
output RegWrite_out;
output MemToReg_out;

reg [31:0] instr;
reg [31:0] extData;
reg [31:0] data1;
reg [31:0] data2;
reg [4:0] RegAddrI;
reg [4:0] RegAddrR;
reg RegDst;
reg [4:0] ALUOp;

```

```

reg [1:0] ALUsrc;
reg MemRead;
reg MemWrite;
reg RegWrite;
reg MemToReg;

always @(posedge clk or posedge rst) begin
    if ( rst )
        begin
            instr <= 0;
            extData <= 0;
            data1 <= 0;
            data2 <= 0;
            RegAddrI <= 0;
            RegAddrR <= 0;
            RegDst <= 0;
            ALUop <= 0;
            ALUsrc <= 0;
            MemRead <= 0;
            MemWrite <= 0;
            RegWrite <= 0;
            MemToReg <= 0;
        end
    else
        begin
            if(flush==0)
                begin
                    instr <= instr_in;
                    MemRead <= MemRead_in;
                    MemWrite <= MemWrite_in;

```

```

        RegWrite <= RegWrite_in;
        MemToReg <= MemToReg_in;
    end
else
    begin
        instr <= 0;
        MemRead <= 0;
        MemWrite <= 0;
        RegWrite <= 0;
        MemToReg <= 0;
    end

    extData <= extData_in;
    data1 <= data1_in;
    data2 <= data2_in;
    RegAddrI <= RegAddrI_in;
    RegAddrR <= RegAddrR_in;
    RegDst <= RegDst_in;
    ALUop <= ALUop_in;
    ALUsrc <= ALUsrc_in;
end

end

assign instr_out = instr;
assign extData_out = extData;
assign data1_out = data1;
assign data2_out = data2;
assign RegAddrI_out = RegAddrI;
assign RegAddrR_out = RegAddrR;
assign RegDst_out = RegDst;
assign ALUop_out = ALUop;

```

```

    assign ALUsrc_out = ALUsrc;
    assign MemRead_out = MemRead;
    assign MemWrite_out = MemWrite;
    assign RegWrite_out = RegWrite;
    assign MemToReg_out = MemToReg;

endmodule

```

4.11 EXMEM 段间寄存器 (EXMEM)

将 EX 级结果传递给 MEM 级

```

module EXMEM
(
    clk,rst ,ALUresult_in,zero_in,rt_in,RegDst_in,RegAddrI_in,RegAddrR_in,MemRead_in,
    MemWrite_in,RegWrite_in,MemToReg_in,ALUresult_out,zero_out,rt_out,RegDst_out,
    RegAddrI_out,RegAddrR_out,MemRead_out,MemWrite_out,RegWrite_out,MemToReg_out);

    input          clk;
    input          rst;
    input  [31:0] ALUresult_in;
    input  zero_in;
    input  [31:0] rt_in;
    input  RegDst_in;
    input  [4:0] RegAddrI_in;
    input  [4:0] RegAddrR_in;
    input  MemRead_in;
    input  MemWrite_in;
    input  RegWrite_in;
    input  MemToReg_in;

```

```

output [31:0] ALUresult_out;
output zero_out;
output [31:0] rt_out;
output RegDst_out;
output [4:0] RegAddrI_out;
output [4:0] RegAddrR_out;
output MemRead_out;
output MemWrite_out;
output RegWrite_out;
output MemToReg_out;

```

```

reg [31:0] ALUresult;
reg zero;
reg [31:0] rt;
reg RegDst;
reg [4:0] RegAddrI;
reg [4:0] RegAddrR;
reg MemRead;
reg MemWrite;
reg RegWrite;
reg MemToReg;

```

```

always @(posedge clk or posedge rst) begin
    if ( rst )
        begin
            ALUresult <= 0;
            zero <= 0;
            rt<=0;
            RegDst<=0;
            RegAddrI <= 0;

```

```

        RegAddrR <= 0;
        MemRead <= 0;
        MemWrite <= 0;
        RegWrite <= 0;
        MemToReg <= 0;
    end
else
    begin
        ALUresult <= ALUresult_in;
        zero <= zero_in;
        rt<=rt_in;
        RegDst<=RegDst_in;
        RegAddrI <= RegAddrI_in;
        RegAddrR <= RegAddrR_in;
        MemRead <= MemRead_in;
        MemWrite <= MemWrite_in;
        RegWrite <= RegWrite_in;
        MemToReg <= MemToReg_in;
    end
end // end always
assign ALUresult_out = ALUresult;
assign zero_out = zero;
assign rt_out = rt;
assign RegDst_out=RegDst;
assign RegAddrI_out = RegAddrI;
assign RegAddrR_out = RegAddrR;
assign MemRead_out = MemRead;
assign MemWrite_out = MemWrite;
assign RegWrite_out = RegWrite;
assign MemToReg_out = MemToReg;

```

```
endmodule
```

4.12 MEMWB 段间寄存器 (MEMWB)

将 MEM 结果传递给 WB 级

```
module MEMWB (clk,  
rst,ALUresult_in ,mem_in,RegDst_in,RegAddrI_in,RegAddrR_in,RegWrite_in,MemTo  
Reg_in,ALUresult_out,mem_out,RegDst_out,RegAddrI_out,RegAddrR_out,RegWrite_  
out,MemToReg_out);
```

```
    input      clk;  
    input      rst;  
    input  [31:0] ALUresult_in;  
    input  [31:0] mem_in;  
    input   RegDst_in;  
    input  [4:0]  RegAddrI_in;  
    input  [4:0]  RegAddrR_in;  
    input   RegWrite_in;  
    input   MemToReg_in;  
    output [31:0] ALUresult_out;  
    output [31:0] mem_out;  
    output   RegDst_out;  
    output [4:0]  RegAddrI_out;  
    output [4:0]  RegAddrR_out;  
    output RegWrite_out;  
    output MemToReg_out;  
  
    reg [31:0] ALUresult;
```

```

reg [31:0] mem;
reg RegDst;
reg [4:0]  RegAddrI;
reg [4:0]  RegAddrR;
reg RegWrite;
reg MemToReg;

always @(posedge clk or posedge rst) begin
    if ( rst )
        begin
            ALUresult<=0;
            mem<=0;
            RegDst<=0;
            RegAddrI<=0;
            RegAddrR<=0;
            RegWrite<=0;
            MemToReg<=0;
        end
    else
        begin
            ALUresult<=ALUresult_in;
            mem<=mem_in;
            RegDst<=RegDst_in;
            RegAddrI<=RegAddrI_in;
            RegAddrR<=RegAddrR_in;
            RegWrite<=RegWrite_in;
            MemToReg<=MemToReg_in;
        end
    end
end
assign ALUresult_out=ALUresult;

```



```

    assign mem_out=mem;
    assign RegDst_out=RegDst;
    assign RegAddrI_out=RegAddrI;
    assign RegAddrR_out=RegAddrR;
    assign RegWrite_out=RegWrite;
    assign MemToReg_out=MemToReg;

endmodule

```

4.13 多路选择器 (mux)

包括四路选择器和二路选择器

```

module mux4alu1 (d0, d1, d2, s, y);

```

```

    input  [31:0] d0, d1, d2;

```

```

    input  [1:0] s;

```

```

    output [31:0] y;

```

```

    reg [31:0] y_r;

```

```

    always @( * ) begin

```

```

        case ( s )

```

```

            2'b00: y_r = d0;

```

```

            2'b01: y_r = d1;

```

```

            2'b10: y_r = d2;

```

```

            default: ;

```

```

        endcase

```

```

    end

```

```

    assign y = y_r;

```

```
endmodule
```

```
module mux4alu2 (d0, d1, d2,s, y);
```

```
    input  [31:0] d0, d1, d2;
```

```
    input  [1:0] s;
```

```
    output [31:0] y;
```

```
    reg [31:0] y_r;
```

```
    always @( * ) begin
```

```
        case ( s )
```

```
            2'b00: y_r = d0;
```

```
            2'b01: y_r = d1;
```

```
            2'b10: y_r = d2;
```

```
            default: ;
```

```
        endcase
```

```
    end
```

```
    assign y = y_r;
```

```
endmodule
```

```
module mux2compare1 (d0, d1, s, y);
```

```
    input  [31:0] d0, d1;
```

```
    input  s;
```

```
    output [31:0] y;
```

```

    reg [31:0] y_r;

    always @( * ) begin
        case ( s )
            0: y_r = d0;
            1: y_r = d1;
            default: ;
        endcase
    end

    assign y = y_r;

endmodule

module mux2compare2 (d0, d1,s, y);

    input  [31:0] d0, d1;
    input  s;
    output [31:0] y;

    reg [31:0] y_r;

    always @( * ) begin
        case ( s )
            0: y_r = d0;
            1: y_r = d1;
            default: ;
        endcase
    end
end

```

```

        assign y = y_r;

    endmodule

```

4.14 冒险检测单元 (Hazard)

流水线冒险:

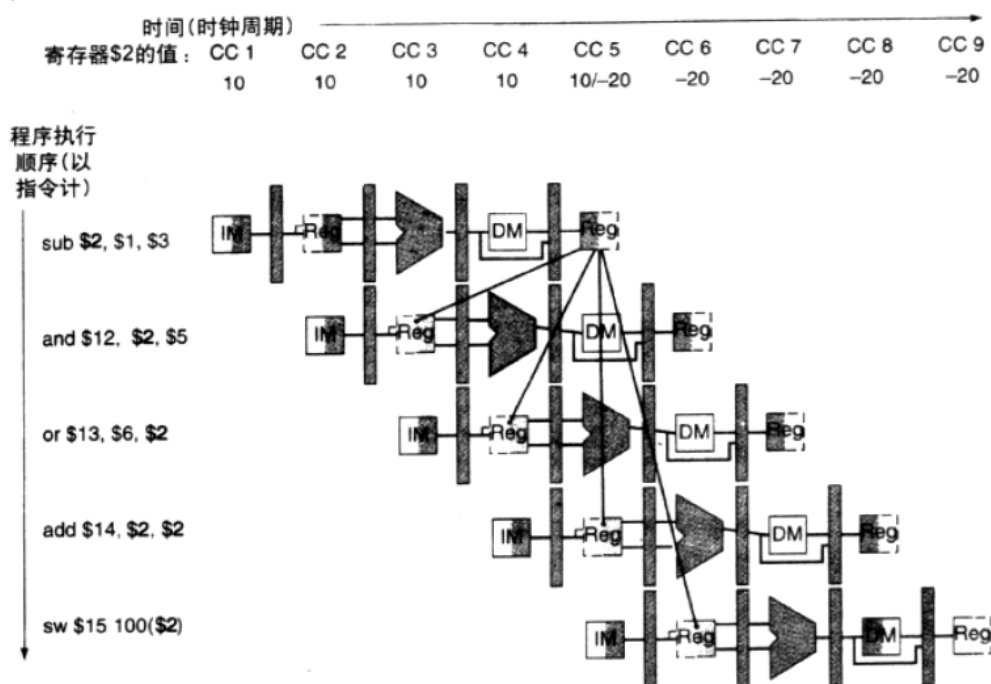
在流水线CPU中, 多条指令通知执行, 由于各种各样的原因, 在下一个时钟周期中下一条指令不能执行, 这种情况称为冒险。冒险分为三类:

(1) 结构冒险: 硬件不支持多条指令在同一个时钟周期内执行。MIPS指令集专为流水线设计, 因此在MIPS CPU中不存在此类冒险。

(2) 数据冒险: 在一个操作必须等待另一操作完成后才能进行时, 流水线必须停顿, 这种情况称为数据冒险。数据冒险分为两类:

数据相关与转发: 流水线内部其中任何一条指令要用到任何其他指令的计算结果时, 将导致数据冒险。通常可以用数据转发(数据定向)来解决此类冒险。

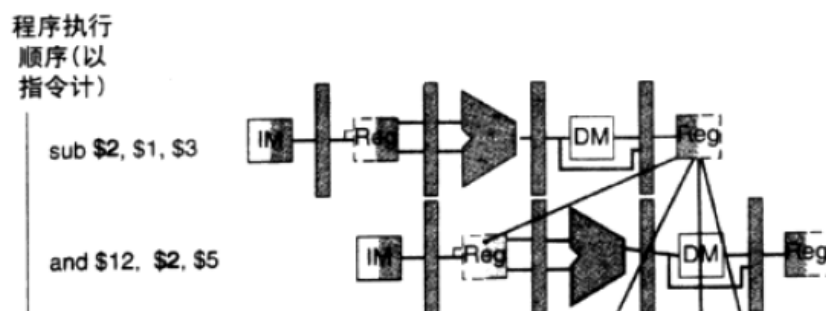
如下图:



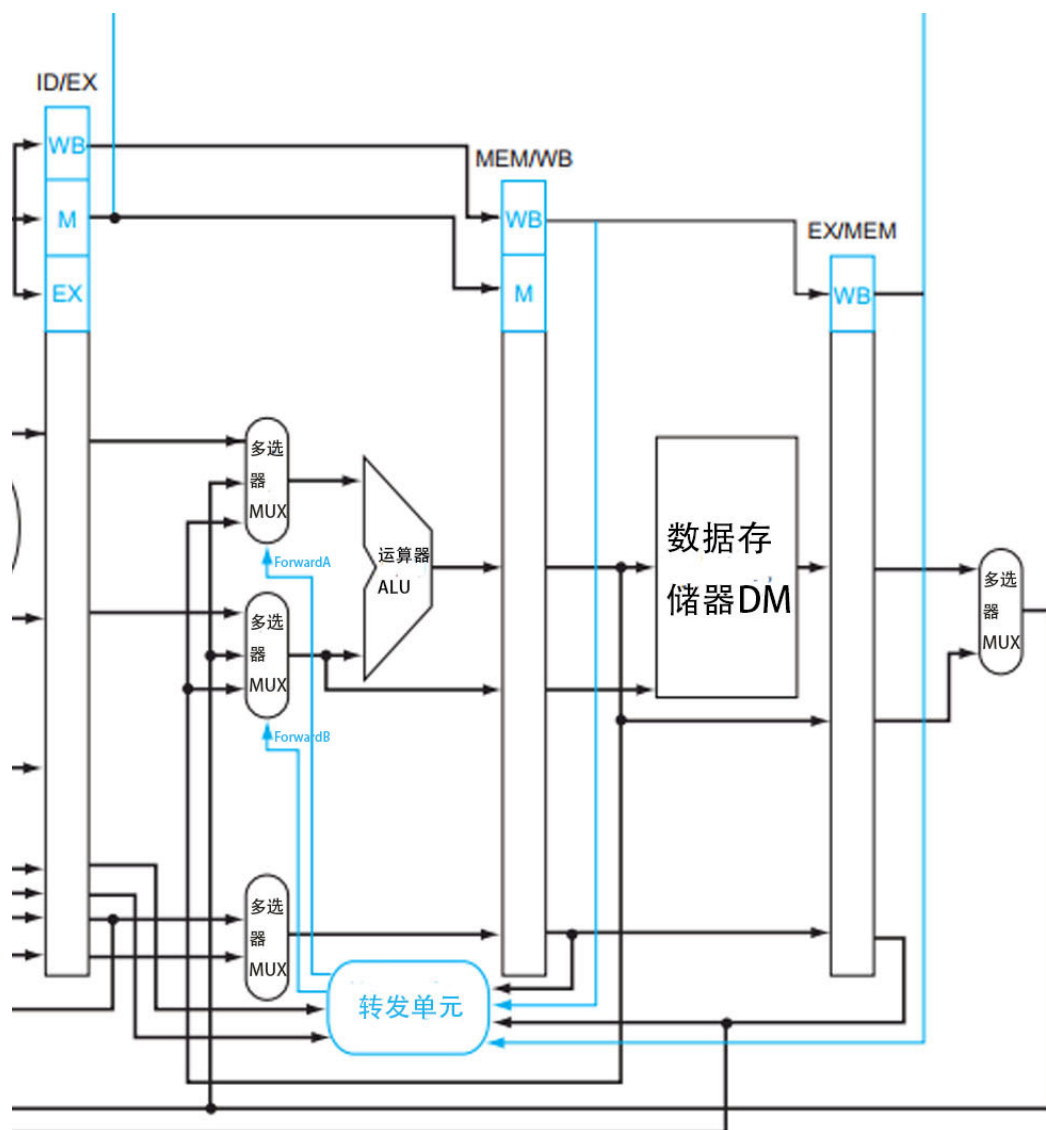
可见, 后4条指令都依赖于第一条指令得到寄存器\$2的结果, 但sub指令要在第五周期才写回寄存器\$2, 但在第三、四、五个时钟周期\$2分别要被and、or和add三个指令用到, 所以这三个指令得到的是错误的未更新的数据, 会引起错误的结果; 而第六个时钟周期\$2要被sw指令用到, 此时得到的才是正确的已更新的数据。这种数据之间的互相关联引起的冒险就是数据相关。可以看出, 当一条依赖关系的方向与时间轴的方向相反时, 就会产生数据冒险。

(1) EX级转发

如下图，sub指令在第五周期写回寄存器\$2，而and指令在第四周期就对sub指令的结果\$2提出申请，显然将得到错误的未更新的数据。



sub指令的结果其实在EX级结尾，即第三周期末就产生了；而and指令在第四时钟周期向sub指令结果发出请求，请求时间晚于结果产生时间，所以只需要sub指令结果产生之后直接将其转发给and指令就可以避免一阶数据相关。转发机制见下图：



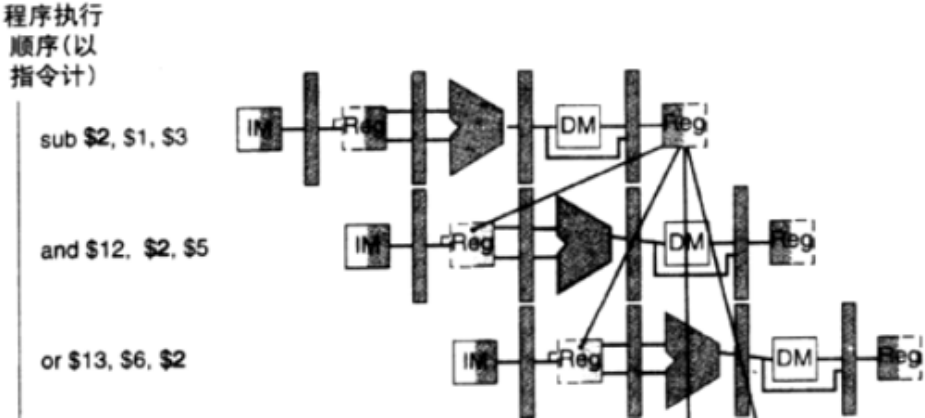
转发条件ForwardA、ForwardB作为数据选择器的地址信号，转发条件不成立时，ALU操作数从ID/EX流水线寄存器中读取；转发条件成立时，ALU操作数取自数据旁

路。

EX级转发条件是：

- a.EX/MEM.RegisterRd=ID/EX.RegisterRs那么ForwardA=2;
- b.EX/MEM.RegisterRd=ID/EX.RegisterRt那么ForwardB=2。

(2) MEM级转发



sub指令在第5时钟周期写回寄存器，而or指令也在第5时钟周期对sub指令的结果提出了请求，很显然or指令读取的数据是未被更新的错误内容。比如sub和or指令间就是利用MEM转发解决问题。

MEM级转发条件是：

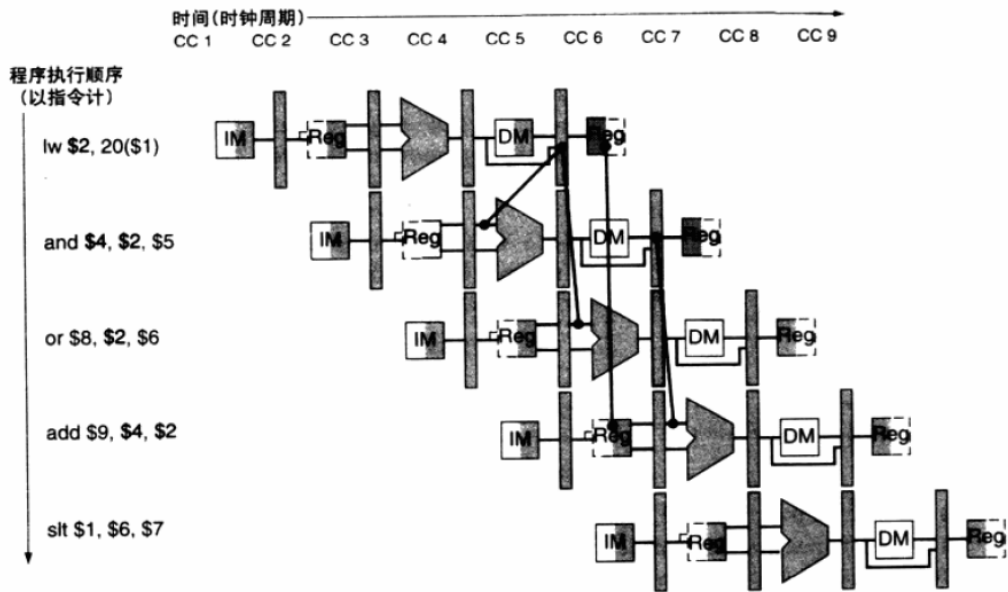
- a.MEM/WB.RegisterRd=ID/EX.RegisterRs，那么ForwardA=1;
- b.MEM/WB.RegisterRd=ID/EX.RegisterRt，那么ForwardB=1。

多路选择器中转发单元的选择：

多路选择器信号线	源	解释
ForwardA=00	ID/EX	第一个ALU操作数来自寄存器堆
ForwardA=10	EX/MEM	第一个ALU操作数由上一个ALU运算结果转发获得
ForwardA=01	MEM/WB	第一个ALU操作数从数据内存或者前面的ALU结果中转发获得
ForwardB=00	ID/EX	第二个ALU操作数来自寄存器堆
ForwardB=10	EX/MEM	第二个ALU操作数由上一个ALU运算结果转发获得
ForwardB=01	MEM/WB	第二个ALU操作数从数据内存或者前面的ALU结果中转发获得

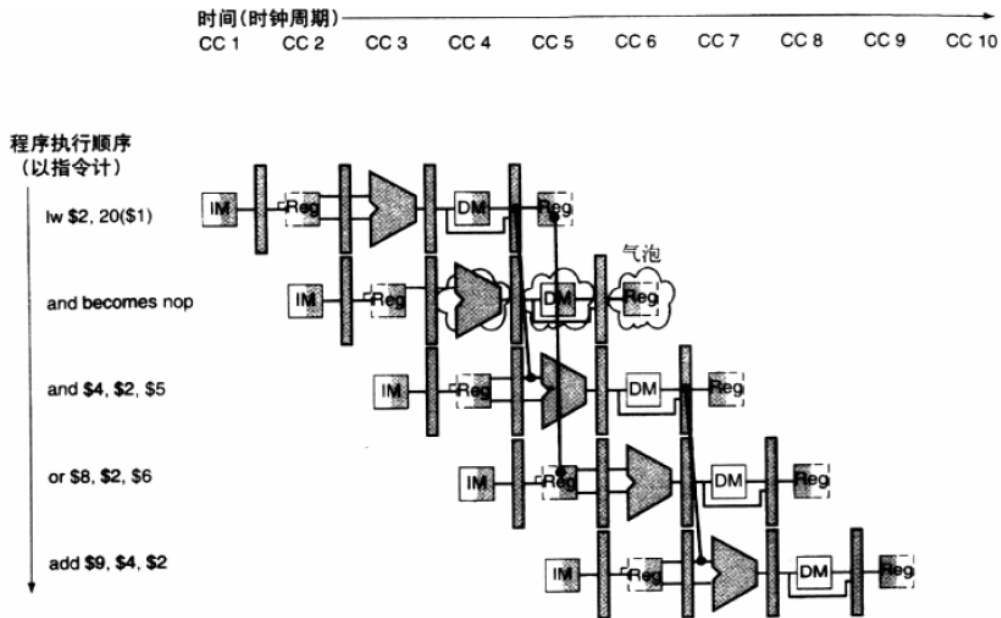
数据冒险：此类冒险发生在当定向的目标阶段在时序上早于定向的源阶段时，数

据转发无效。通常是引入流水线阻塞，即气泡（bubble）来解决。



注意到lw指令只能在第四时钟周期从内存中读出数据，因此它和紧随其后的and指令之间的依赖关系与时序方向是相反的，这种冒险是无法通过转发来实现的。

冒险的解决：为解决数据冒险，我们引入流水线阻塞。当冒险检测单元检测到冒险条件成立时，在lw指令和下一条指令之间插入阻塞，即流水线气泡（bubble），使后一条指令延迟一个时钟周期执行，这样就将该冒险转化为MEM转发。

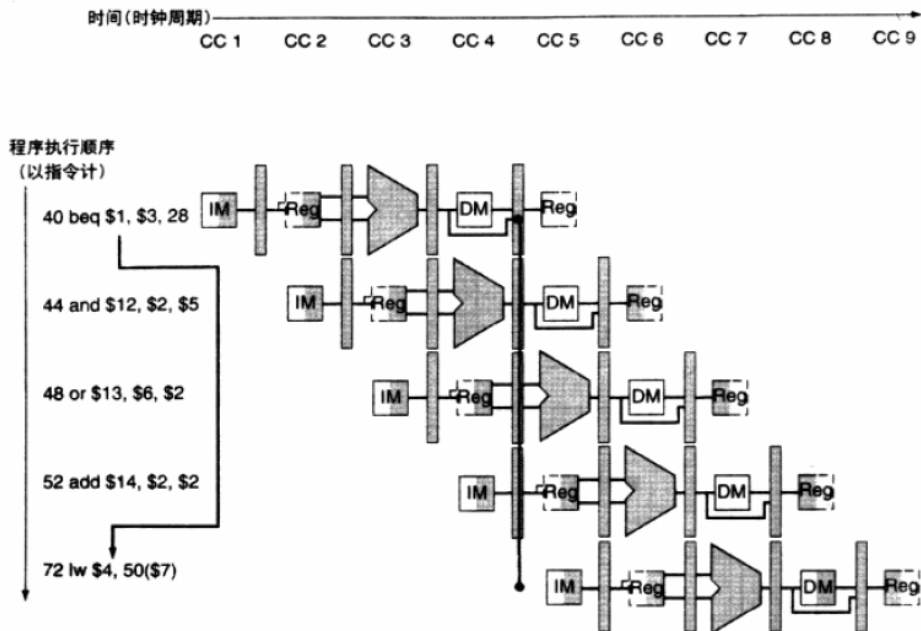


需要注意的是，如果处于ID级的指令被阻塞，那么处于IF级的指令也必须阻塞，否则，处于ID级的指令就会丢失。防止这两条指令继续执行的方法是：保持PC寄存器和IF/ID流水线寄存器不变，同时插入一个流水线气泡。

。具体实现方法如下：在ID级检测到冒险条件时，Hazard输出两个信号：stallID与stallPC。stallID信号将ID/EX流水线寄存器中的EX、MEM和WB级控制信号全部清

零。这些信号传递到流水线后面的各级，由于控制信号均为零，所以不会对任何寄存器和存储器进行写操作，高电平有效。stallPC信号禁止PC寄存器和IF/ID流水线寄存器接收新数据，低电平有效。

(3) 控制冒险：CPU需要根据分支指令的结果做出决策，而此时其他指令可能还在执行中，这时会出现控制冒险，也称为分支冒险。



流水线每个时钟周期都得取指令才能维持运行，但分支指令必须等到 MEM 级才能确定是否执行分支。这种为了确定预取正确的指令而导致的延迟叫做控制冒险或分支冒险。

如果我们能在流水线中提前分支指令的执行过程，那么就能减少需要清除的指令数。这是一种提高分支效率的方法，降低了执行分支的代价。因此我们采用提前分支指令的方法解决分支冒险。

提前分支指令需要提前完成两个操作：

① 计算分支的目的地址：

由于已经有了 PC 值和 IF/ID 流水线寄存器中的指令值，所以可以很方便地将 EX 级的分支地址计算电路移到 ID 级。我们针对所有指令都执行分支地址的计算过程，但只有在需要它的时候才会用到。

② 判断分支指令的跳转条件：

加入一个控制信号 flushIFID，做为 IF/ID 流水线寄存器的清零信号。当分支冒险成立 flushIFID=1，否则 flushIFID=0。

bne, beq 指令需要上一级的结果时，需要多选器信号将上一级数据传到分支指令，即 EXMEM_rd==IFID_rs, compareSrc1=1，或者 EXMEM_rd==IFID_rt, compareSrc2=1。

考虑到本系统还要实现的无条件跳转指令：J 和 JR，在执行这两个指令时也必须要对

IF/ID 流水线寄存器进行清空，因此， 当 jump=1 时， flushIFID=1。

module

Hazard(EXMEM_RegWrite,EXMEM_rd,IDEX_rs,IDEX_rt,MEMWB_RegWrite
,MEMWB_rd,ForwardA,ForwardB,IDEX_MemRead,IFID_rs,IFID_rt,stallID,stallPC,flushIFID,flushIDEX,pcsel,jump,branch,compareSrc1,compareSrc2,IDEX_rd
,IDEX_RegWrite,EXMEM_MemRead,EXMEM_rt);

input EXMEM_RegWrite;

input [4:0] EXMEM_rd;

input [4:0] MEMWB_rd;

input MEMWB_RegWrite;

input [4:0] IDEX_rs;

input [4:0] IDEX_rt;

input IDEX_MemRead;

input [4:0] IFID_rs;

input [4:0] IFID_rt;

input ptsel;

input jump;

input branch;

input EXMEM_MemRead;

input [4:0]EXMEM_rt;

input [4:0] IDEX_rd;

input IDEX_RegWrite;

output reg [1:0] ForwardA;

output reg [1:0] ForwardB;

output reg stallID;

output reg stallPC;

output reg flushIFID;

```

output reg flushIDEX;

output reg compareSrc1;
output reg compareSrc2;


always@(*)
begin
//EX rd-rs
    if(EXMEM_RegWrite==1&&EXMEM_rd!=0&&EXMEM_rd==IDEX_rs)
        assign ForwardA=2;
//MEM rd-rs
    else
if(MEMWB_RegWrite==1&&MEMWB_rd!=0&&MEMWB_rd==IDEX_rs)
        assign ForwardA=1;
    else
        assign ForwardA=0;
//EX rd-rt

if(EXMEM_RegWrite==1&&EXMEM_rd!=0&&EXMEM_rd==IDEX_rt)
        assign ForwardB=2;
//MEM rd-rt
    else
if(MEMWB_RegWrite==1&&MEMWB_rd!=0&&MEMWB_rd==IDEX_rt)
        assign ForwardB=1;
    else
        assign ForwardB=0;
//beq.bne

if(branch==1&&EXMEM_RegWrite==1&&EXMEM_rd!=0&&EXMEM_rd==I
FID_rs)
        assign compareSrc1=1;

```

```

else
    assign compareSrc1=0;

if(branch==1&&EXMEM_RegWrite==1&&EXMEM_rd!=0&&EXMEM_rd==I
FID_rt)
    assign compareSrc2=1;
else
    assign compareSrc2=0;

//ID load stall
if((IDEX_MemRead==1&&(IDEX_rt==IFID_rs||IDEX_rt==IFID_rt))||
    (branch==1&&EXMEM_MemRead==1&&
(EXMEM_rt==IFID_rs||EXMEM_rt==IFID_rt)) ||

(branch==1&&IDEX_RegWrite==1&&IDEX_rd!=0&&(IDEX_rd==IFID_rs||ID
EX_rd==IFID_rt)) )
    begin
        assign stallID=1;
        assign stallPC=1;
        assign flushIDEX=1;
    end
else
    begin
        assign stallID=0;
        assign stallPC=0;
        assign flushIDEX=0;
    end

//jump flush

```

```

        if(jump==1||pcsel==1)
            assign flushIFID=1;
        else
            assign flushIFID=0;
        end
    endmodule

```

5 测试及结果分析

5.1 仿真代码及分析

详细见: **mipstest_pipelinedloop.asm**文件

```

# add, sub, and, or, slt, addi, lw, sw, beq, j have been done in mipstest.asm
# sll, lui, bne, jal, ori, addu, subu, jr, srl have been done in mipstest_extended.asm
# sltu, nor are added in this test. (21 intructions in total)

```

#	Assembly	Description	Address	Machine
# Test if there is data hazard				
main:	addi \$2, \$0, 5	# initialize \$2 = 5	00	20020005
	ori \$3, \$0, 12	# initialize \$3 = 12	04	3403000c
	subu \$1, \$3, \$2	# \$1 = 12 - 5 = 7	08	00620823
	srl \$7, \$1, 1	# \$7 = 7 >> 1 = 3	0c	00013842
call_a: j	a	# jump to a	10	08000015
	or \$4, \$7, \$2	# \$4 = (3 or 5) = 7	14	00e22025
	and \$5, \$3, \$4	# \$5 = (12 and 7) = 4	18	00642824
	add \$5, \$5, \$4	# \$5 = 4 + 7 = 11	1c	00a42820
	beq \$5, \$7, end	# should not be taken	20	10a70018
	sltu \$4, \$3, \$4	# \$4 = (12 < 7) = 0	24	0064202b
#Test if there is control hazard				
	beq \$4, \$0, around	# should be taken	28	10800003
	addi \$5, \$0, 0	# should not happen	2c	20050000
	addi \$5, \$0, 0	# should not happen	30	20050000
	addi \$5, \$0, 0	# should not happen	34	20050000

```

around: slt  $4, $7, $2      # $4 = 3 < 5 = 1      38      00e2202a
        addu $7, $4, $5      # $7 = 1 + 11 = 12    3c      00853821
        sub  $7, $7, $2      # $7 = 12 - 5 = 7     40      00e23822
        sw   $7, 68($3)      # [80] = 7           44      ac670044
        lw   $2, 80($0)      # $2 = [80] = 7       48      8c020050
        j    end             # should be taken      4c      08000021
        addi $2, $0, 1       # should not happen    50      20020001
a:      sll  $7, $7, 2        # $7 = 3 << 2 = 12    54      00073880
call_b: jal  b               # jump to b            58      0c000019
        addi $31,$0,20       # $31 <= 20           5c      201f0014
        jr   $31             # return to call_a     60      03e00008
b:      lui  $1, 0xFFAA      # $1 <= 0xFFAA0000    64      3c01ffaa
        slt  $1, $7, $1      # $1 <= 0           68      00e1082a
        bne  $1, $0, end     # should not be taken 6c      14200005
        sub  $7, $7, $2      # $7 = 12 - 5 = 7     70      00e23822
        srl  $7, $7, 1       # $7 = 7 >> 1 = 3     74      00073842
        nor  $1, $7, $1      # $1 = 0xFFFFFFF0    78      00e10827
        sltu $1, $1, $7      # $1 <= 0           7c      0027082b
        jr   $31             # return to call_b     80      03e00008

# Test if there is load use hazard

end:    sw   $3, 84($0)      # [84] = 12           84      ac030054
        lw   $7, 72($3)      # $7 = [84] = 12      88      8c670048
        sw   $7, 68($3)      # [80] = 12           8c      ac670044
        lw   $6, 68($3)      # $6 = [80] = 12      88      8c660044
        add  $6, $6, $5      # $6 = 12 + 11 = 23   90      00c53020
loop:   j    loop           # dead loop           94      08000026

# $0  = 0  # $1  = 0  # $2  = 7  # $3  = c
# $4  = 1  # $5  = b  # $6  = 17h # $7  = c
# $31 = 14h

```

5.2 仿真测试结果

由算法可知：如果成功，它应该将值 12 写入地址 80 和 84，并且寄存器\$7 和寄存器\$3 应该为 12

部分寄存器值应该为：

```
# $0  = 0  # $1  = 0  # $2  = 7    # $3  = c
# $4  = 1  # $5  = b  # $6  = 17h  # $7  = c
# $31 = 14h
```

```
# instr: 08000c26
# R[00-07]=00000000, 00000000, 00000007, 0000000c, 00000001, 0000000b, 00000017, 0000000c
# R[08-15]=00000000, 00000000, 00000000, 00000000, 00000000, 00000000, 00000000, 00000000
# R[16-23]=00000000, 00000000, 00000000, 00000000, 00000000, 00000000, 00000000, 00000000
# R[24-31]=00000000, 00000000, 00000000, 00000000, 00000000, 00000000, 00000000, 00000014
# R[32-33]=00000000, 00000000
# pc: 0000309c
# instr: xxxxxxxx
# pc: 00003098
# instr: 08000c26
```

结论

本实验的各子模块结构清晰、原理明确，编写代码时，思维比较清晰，但让我为难的问题也不少，比如：

（1）代码中用到的变量名较多，很容易弄混乱。在时钟周期赋值时，变量名必须是reg类型。

（2）在有多种情况选择时，原先我用的是if-else循环嵌套，但感觉代码较为复杂，改为case使得代码显得更加简洁明了，但case语句不能嵌套，且需要default排除异常情况。

（3）在MIPS连接中，经常一个大小写不对就能发生错误，而且这种错误非常难找，所以在写代码时需要非常注意。

（4）在ctrl单元里，信号的判断需要十分熟悉所有指令的执行过程，才能保证不能出错。

（5）在Hazard单元中，要弄清楚控制冒险单元的原理。

通过本次试验，巩固了我在计算机组成原理课上学习的流水线MIPS CPU的知识，让我了解到设计流水线MIPS CPU的方法，并从中得到乐趣自己编程学习的趣味。

参考文献

- [1]雷思磊, 自己动手写 CPU[M], 电子工业出版社, 第一版, 2014.
- [2]李亚民, 计算机原理与设计[M], 清华大学出版社, 第一版, 2011.

教师评语评分

评语： _____

评分： _____

评阅人：

年 月 日