武汉大学计算机学院
本科生课程设计报告

MIPS 流水线 CPU 设计

专 业 名 称　：计算机科学与技术

课 程 名 称　：计算机组成与设计

指 导 教 师　：蔡朝晖

学 生 学 号　：2018302120209

学 生 姓 名　：罗宇轩

二〇二〇年五月

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：罗宇轩　　日期：2018.5.2

摘　要

MIPS 的 CPU 设计实验的实验目的是分别实现单周期和流水线的 CPU 设计，要求 CPU 能完成常见的 35 条 MIPS 汇编指令。

实验设计主要遵循先理解学习 verilog 语言的语法以及实验环境 modelsim 的使用，再复习上学期计算机组成原理的相关知识，然后进行代码的编写和在 modelsim 中进行调试。

实验内容主要包括：首先进行单周期 CPU 的部件设计，然后完成单周期 CPU 的顶层设计。完成上述设计后即可以在 modelsim 中进行调试以验证代码是否正确；接下来在单周期 CPU 的部件的基础上进行复用和扩展，使部件能够适合流水线 CPU 的设计，然后进行流水线 CPU 的顶层设计。同样，完成上述设计后在 modelsim 的环境下进行调试和验证。需特别说明的是：因为今年疫情的影响，无法在课程其间返回学校，将代码下载到 FPGA 开发板上进行测试，故本实验报告中的内容将仅包括 CPU 在 modelsim 中仿真的部分。

实验结论为验证该实验设计的 CPU 在测试程序集上的运行结果是否和 MARS 中运行的结果相一致。

关键词：MIPS； CPU 设计；单周期；流水线

目 录

1 实验目的和意义

1.1 实验目的

本实验通过课堂上学习老师对该实验的要求与讲解，课下认真阅读课本和相关参考书籍，在弄清楚 CPU 设计的原理后进行代码的编写和调试，以分别实现能支持常见 MIPS 汇编指令的单周期和流水线 CPU

1.2 实现目的

本实验通过实现能够支持常见 MIPS 汇编指令的单周期和流水线 CPU，来提升对计算机组成原理这门课程中关于 CPU 部分的理解和相关实践能力，并加强在未来编写程序的时候从底层硬件的角度来优化软件设计的思维。

2 实验环境介绍

2.1 Verilog HDL

Verilog HDL 是一种硬件描述语言，用于从算法级、门级到开关级的多种抽象设计层次的数字系统建模。被建模的数字系统对象的复杂性可以介于简单的门和完整的电子数字系统之间。数字系统能够按层次描述，并可在相同描述中显式地进行时序建模。

Verilog HDL 语言不仅定义了语法，而且对每个语法结构都定义了清晰的模拟、仿真语义。因此，用这种语言编写的模型能够使用 Verilog 仿真器进行验证。语言从 C 编程语言中继承了多种操作符和结构。Verilog HDL 提供了扩展的建模能力，其中许多扩展最初很难理解。但是，Verilog HDL 语言的核心子集非常易于学习和使用，这对大多数建模应用来说已经足够。当然，完整的硬件描述语言足以对从最复杂的芯片到完整的电子系统进行描述。

2.2 MARS

MARS is a lightweight interactive development environment (IDE) for programming in MIPS assembly language, intended for educational-level use with Patterson and Hennessy’s Computer Organization and Design.（MARS 是一个轻量级的交互式开发环境（IDE），用于使用 MIPS 汇编语言进行编程，旨在与 Patterson 和 Hennessy 的计算机组织和设计一起用于教育级别。）

本实验中使用的 MARS 版本为 MARS 4_5。

2.3 ModelSim

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。

本实验中使用的 modelsim 版本为 10.4

2.4 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。这也是一个基于 AMBA AXI4 互联规范、IP-XACT IP 封装元数据、工具命令语言(TCL)、Synopsys 系统约束(SDC) 以及其它有助于根据客户需求量身定制设计流程并符合业界标准的开放式环境。赛灵思构建的的 Vivado 工具把各类可编程技术结合在一起，能够扩展多达 1 亿个等效 ASIC 门的设计。

如前文所提到的，受疫情影响本次实验无法在 FPGA 开发板上进行调试，故实验中没有用到 vivado 相关环境。

2.5 Nexys 4DDR

Nexys4DDR 开发板，是基于最新技术 Spartan-6 FPGA 的数字系统开发平台。

本次实验同样没有用到该开发环境，具体理由同上

2.6 非实验环境信息

在代码编写的过程中选用的代码编辑器为 vscode，而没有使用 modelsim 自带的代码编辑器。

本实验已在 github 上开源，具体项目地址为：https://github.com/Lotherxuan/MIPS_CPUdesign

3 单周期概要设计

3.1 总体设计

该 CPU 采取单周期的设计，支持的基本指令如下：add/sub/and/or/slt/sltu/addu/subu/addi/ori/lw/sw/beq/j/jal。支持的扩展指令如下：sll/nor/lui/slti/bne/andi/srl/sllv/srlv/jr/jalr/xor/sra/srav/lb/lh/lbu/lhu/sb/sh

3.2 ENCODE（预定义单元）

3.2.1 功能描述

通过 verilog 中的预编译语句，将指令的操作码，功能码，以及 CPU 设计中的控制信号从二进制的数字唯一映射成字符标识的预定义宏以提高代码的可读性。对应的文件为 ENCODE.v

3.2.2 模块接口

该模块并不是具体的实体部件设计，故没有模块接口表。

3.3 PC（程序计数器）

3.3.1 功能描述

PC 主要负责接受 NPC 部件所更新的程序地址，以产生下一条指令的地址。对应的文件为 PC.v，在顶层设计中的实体名为 pc。

3.3.2 模块接口

信号名	方向	描述
clk	输入	时钟信号
rst	输入	重置信号
NPC[31:0]	输入	由 NPC 部件产生的 pc 更新的值
PC[31:0]	输出	更新后的 pc

3.4 NPC（程序计数器更新单元）

3.4.1 功能描述

NPC 主要负责接受当前 PC 的值以及如 16 位和 26 位立即数等和 PC 更新相关的值，来进行 PC 的更新。除了实现 PC+4 之外，还有根据 PC 更新的控制信号以

实现分支和跳转的功能。对应的文件名为 NPC.v，在顶层设计中的实体名为 npc。

3.4.2 模块接口

信号名	方向	描述
PC[31:0]	输入	需要更新的 pc 的值
NPCop[2:0]	输入	控制 pc 进行+4 或分支或跳转的控制信号
Zero	输入	由 ALU 产生的零信号
IMM	输入	分支指令中的 16 位立即数
Imm16	输入	跳转指令中的 26 位立即数
rs	输入	从寄存器中读取出的跳转的目的地址
NPC	输出	更新后的 pc 的值

3.5 IM（程序存储器）

3.5.1 功能描述

存储机器指令，并根据 pc 的值读取出当前执行的指令。对应的文件名为 IM.v，在顶层设计中的实体名为 im。

3.5.2 模块接口

信号名	方向	描述
Addr[31:0]	输入	当前 pc 的值
Instr[31:0]	输出	由 pc 的值读取出来的指令

3.6 RF（寄存器组）

3.6.1 功能描述

存储和修改 32 个 32 位寄存器中的数据，并能同时读取 2 个寄存器中的 32 位数据和在 RegWrite 信号的控制下将 32 位数据写入一个 32 位寄存器。对应的文件名为 RF.v，在顶层设计中的实体名为 rf。

3.6.2 模块接口

信号名	方向	描述
clk	输入	时钟信号
rst	输入	重置信号
RFWr	输入	写使能信号
A1[4:0]	输入	读取的第一个寄存器的寄存器号
A2[4:0]	输入	读取的第二个寄存器的寄存器号
A3[4:0]	输入	待写入数据的寄存器的寄存器号
WD[31:0]	输入	待写入寄存器的 32 位数据
RD1[31:0]	输出	读取的第一个寄存器的数据
RD2[31:0]	输出	读取的第二个寄存器的数据

3.7 CONTROL（控制单元）

3.7.1 功能描述

根据 32 位指令的功能码和操作码字段生成控制信号，对应的文件名为 CONTROL.v,顶层设计中的实体名为 control

3.7.2 模块接口

信号名	方向	描述
Op[5:0]	输入	指令的操作码
Func[5:0]	输入	指令的功能码
RegDst[1:0]	输出	写寄存器时的目标寄存器
Jump	输出	跳转信号
Branch	输出	分支信号
MemRead	输出	读数据存储器的信号
MemtoReg[1:0]	输出	写寄存器时写入数据的来源
ALUop[4:0]	输出	控制 ALU 运算的类型
MemWrite	输出	写数据存储器的信号
ALUSrc	输出	控制 ALU 的第二个操作数的来源
RegWrite	输出	写寄存器信号
Shamt	输出	控制 ALU 的第一个操作数是否来自 shamt
Ext[1:0]	输出	符号扩展的类型
PCSrc[2:0]	输出	分支或跳转的类型
MemControl[3:0]	输出	存储和加载指令的类型

3.8 ALU（算术运算单元）

3.8.1 功能描述

根据输入 ALU 运算类型的控制信号，对两个 ALU 的源操作数进行不同类型的运算，对应的文件名为 ALU.v,顶层设计中的实体名为 alu

3.8.2 模块接口

信号名	方向	描述
A	输入	第一个操作数
B	输入	第二个操作数
ALUOp	输入	控制 ALU 运算类型的控制信号
C	输出	运算结果
Zero	输出	判断两个源操作数是否相等的零信号

3.9 MEM（数据存储器）

3.9.1 功能描述

写入、读取和存储数据，其中写入和读取的端口是共用的，只输入一个地址，在写使能和读使能的信号的控制下分别进行写和读的操作。对应的文件名为 MEM.v,顶层设计中的实体名为 mem。

3.9.2 模块接口

信号名	方向	描述
clk	输入	时钟信号
rst	输入	重置信号
Input_address[31:0]	输入	写入或读取的端口号
Input_data[31:0]	输入	写入的 32 位数据
Wr	输入	写使能信号
MemControl	输入	存储和加载指令的类型
Output_data	输出	读取出来的 32 位数据

3.10 EXT（符号扩展单元）

3.10.1 功能描述

根据符号扩展的控制信号将 16 位立即数扩展成 32 位的数据，对应的文件名为 EXT.v,顶层设计中的实体名为 ext。

3.10.2 模块接口

信号名	方向	描述
Imm16	输入	16 位立即数
EXTOp	输入	控制符号扩展类型的控制信号
Imm32	输出	符号扩展后的 32 位数据

3.11 MIPS（顶层文件）

3.11.1 功能描述

CPU 设计中的顶层文件，负责将上述各部件组装起来。对应的文件名为 MIPS.v，顶层设计中的实体名为 mips。

3.11.2 模块接口

信号名	方向	描述
clk	输入	时钟信号
rst	输入	重置信号

4　单周期详细设计

4.1 CPU 总体结构

代码实现：

```
`include "ENCODE.v"

module MIPS(clk, rst);

input clk;
input rst;

//控制器相关
wire[1: 0] RegDst;
wire Jump;
wire Branch; //在本 cpu 设计中所有地址相关操作都由 PCsrc 控制，
branch 和 jump 都为冗余信号，可以去掉
wire MemRead;
wire[1: 0] MemtoReg;
wire[4: 0] ALUop;
wire MemWrite;
wire ALUSrc;
wire RegWrite;
wire Shamt;
wire[1: 0] Ext;
wire[2: 0] PCsrc;
wire[3: 0] MemControl;

//数据内存相关
wire [31: 0] DM_out;

//算术运算相关
wire [31: 0] Alu1; //第一个操作数
wire [31: 0] AluMux_Result;
//算术运算相关
```

```
wire zero;
wire [31: 0] ALU_Result;

//符号扩展相关
wire [31: 0] Imm32;

//寄存器相关
wire [31: 0] RD1;
wire [31: 0] RD2; //从 RF 中读出来的数据
wire [31: 0] RF_wd; //等待写入 RF 的数据

//目标寄存器相关
wire [4: 0] RF_rd;

//指令本体
wire [31: 0]Instruction;

//拆分指令
wire [5: 0] Op;
wire [5: 0] Funct;
wire [4: 0] rs;
wire [4: 0] rt;
wire [4: 0] rd;
wire [15: 0] Imm16;
wire [25: 0] IMM;
wire [5: 0] shamt;

//指令地址相关
wire [31: 0] PC;
wire [31: 0] NPC;

assign Op = Instruction[31: 26];
assign Funct = Instruction[5: 0];
assign rs = Instruction[25: 21];
```

```
assign rt = Instruction[20: 16];
assign rd = Instruction[15: 11];
assign Imm16 = Instruction[15: 0];
assign IMM = Instruction[25: 0];
assign shamt = Instruction[10: 6];


NPC npc (
      .PC(PC),
      .NPCop(PCsrc),
      .Zero(zero),
      .IMM(IMM),
      .Imm16(Imm16),
      .rs(RD1),
      .NPC(NPC)
    );

PC pc(
     .clk(clk),
     .rst(rst),
     .NPC(NPC),
     .PC(PC)
   );

IM im(
     .addr(PC),
     .instr(Instruction)
   );

MUX4 #(32) mux4_RFWD(
       .d0(ALU_Result),
       .d1(DM_out),
       .d2((PC + 4)),
       .d3(32'd0),
```

```
       .select(MemtoReg),
       .dout(RF_wd)
     );

MUX4 #(5) mux4_RFRD(
       .d0(rt),
       .d1(rd),
       .d2(5'd31),
       .d3(5'd0),
       .select(RegDst),
       .dout(RF_rd)
     );

RF rf(
     .clk(clk),
     .rst(rst),
     .RFWr(RegWrite),
     .A1(rs),
     .A2(rt),
     .A3(RF_rd),
     .WD(RF_wd),
     .RD1(RD1),
     .RD2(RD2)
   );

EXT ext(
      .Imm16(Imm16),
      .EXTOp(Ext),
      .Imm32(Imm32)
    );

MUX2 #(32) mux2_ALUB(
       .d0(RD2),
       .d1(Imm32),
```

```
       .select(ALUSrc),
       .dout(AluMux_Result)
     );

MUX2 #(32) mux2_ALUA(
       .d0(RD1),
       .d1({26'd0, shamt[5: 0]}),
       .select(Shamt),
       .dout(Alu1)
     );

ALU alu(
      .A(Alu1),
      .B(AluMux_Result),
      .ALUOp(ALUop),
      .C(ALU_Result),
      .Zero(zero)
    );

MEM mem(
      .clk(clk),
      .rst(rst),
      .input_address(ALU_Result),
      .input_data(RD2),
      .Wr(MemWrite),
      .MemControl(MemControl),
      .output_data(DM_out)
    );

CONTROL control(
          .Op(Op),
          .Func(Funct),
          .RegDst(RegDst),
          .Jump(Jump),
```

```
          .Branch(Branch),
          .MemRead(MemRead),
          .MemtoReg(MemtoReg),
          .ALUop(ALUop),
          .MemWrite(MemWrite),
          .ALUSrc(ALUSrc),
          .RegWrite(RegWrite),
          .Shamt(Shamt),
          .Ext(Ext),
          .PCSrc(PCsrc),
          .MemControl(MemControl)
        );

endmodule
```

描述说明：顶层设计上已尽量做到部件化，但仍有一些瑕疵，比如在拆分指令上仍使用了 assign 语句，而没有将拆分指令的功能放到一个单独设计的部件上。

4.2 ENCODE（预定义单元）

实现代码：

```
//instrution operation code
`define R_OP        6'b000000

`define LB_OP         6'b100000
`define LH_OP         6'b100001
`define LBU_OP        6'b100100
`define LHU_OP        6'b100101
`define LW_OP         6'b100011

`define SB_OP         6'b101000
`define SH_OP         6'b101001
`define SW_OP         6'b101011
```

```
`define ADDI_OP       6'b001000
`define ADDIU_OP      6'b001001
`define ANDI_OP       6'b001100
`define ORI_OP        6'b001101
`define XORI_OP       6'b001110
`define LUI_OP        6'b001111
`define SLTI_OP       6'b001010
`define SLTIU_OP      6'b001011

`define BEQ_OP        6'b000100
`define BNE_OP        6'b000101
`define BGEZ_OP       6'b000001
`define BGTZ_OP       6'b000111
`define BLEZ_OP       6'b000110
`define BLTZ_OP       6'b000001

`define J_OP          6'b000010
`define JAL_OP        6'b000011

//R type Instruction Funct
`define ADD_FUNCT     6'b100000
`define ADDU_FUNCT    6'b100001
`define SUB_FUNCT     6'b100010
`define SUBU_FUNCT    6'b100011
`define AND_FUNCT     6'b100100
`define NOR_FUNCT     6'b100111
`define OR_FUNCT      6'b100101
`define XOR_FUNCT     6'b100110
`define SLT_FUNCT     6'b101010
`define SLTU_FUNCT    6'b101011
`define SLL_FUNCT     6'b000000
`define SRL_FUNCT     6'b000010
`define SRA_FUNCT     6'b000011
```

```
`define SLLV_FUNCT    6'b000100
`define SRLV_FUNCT    6'b000110
`define SRAV_FUNCT    6'b000111
`define JR_FUNCT      6'b001000
`define JALR_FUNCT    6'b001001

//ALU control signal
`define ALU_NOP   5'b00000
`define ALU_ADDU  5'b00001
`define ALU_ADD   5'b00010
`define ALU_SUBU  5'b00011
`define ALU_SUB   5'b00100
`define ALU_AND   5'b00101
`define ALU_OR    5'b00110
`define ALU_NOR   5'b00111
`define ALU_XOR   5'b01000
`define ALU_SLT   5'b01001
`define ALU_SLTU  5'b01010
`define ALU_EQL   5'b01011
`define ALU_BNE   5'b01100
`define ALU_GT0   5'b01101
`define ALU_GE0   5'b01110
`define ALU_LT0   5'b01111
`define ALU_LE0   5'b10000
`define ALU_SLL   5'b10001
`define ALU_SRL   5'b10010
`define ALU_SRA   5'b10011
`define ALU_SLLV  5'b10100
`define ALU_SRLV  5'b10101
`define ALU_SRAV  5'b10110

//NPC control signal
`define NPC_PLUS4   3'b000
`define NPC_BRANCH  3'b001
```

```
`define NPC_JUMP    3'b010
`define NPC_JAL     3'b011
`define NPC_JALR    3'b100
`define NPC_BNE     3'b101
`define NPC_JR      3'b110

//Extend control signal
`define EXT_ZERO    2'b01
`define EXT_SIGNED  2'b00
`define EXT_HIGHPOS 2'b10

//MEM control signal
`define MEM_NOP     4'b0000
`define MEM_LW      4'b0001
`define MEM_SW      4'b0010
`define MEM_LB      4'b0011
`define MEM_LH      4'b0100
`define MEM_LBU     4'b0101
`define MEM_LHU     4'b0110
`define MEM_SB      4'b0111
`define MEM_SH      4'b1000
```

描述说明：该文件实现了对指令的功能码和操作码以及一些取值范围较大的控制信号的预定义宏，这样做的好处是能提高代码的可读性，同时尽量减少和避免弄混淆控制信号的值与控制信号意义的对应。

4.3 PC（程序计数器）

实现代码：

```
module PC( clk, rst, NPC, PC );

input clk;
input rst;
input [31: 0] NPC;
output reg [31: 0] PC;
```

```
always @(posedge clk, posedge rst)
  if (rst)
    PC <= 32'h0000_0000;
//      PC <= 32'h0000_3000
//      上一行表示程序段从内存中的0000_3000 作为起始的情况。
  else
    PC <= NPC;

endmodule
```

描述说明：通过重置信号有效时对 PC 赋不同的值，可以模拟程序从内存中的 0 作为起始存储地址和 3000 作为起始存储地址的不同情况。

4.4 NPC（程序计数器更新单元）

实现代码：

```
`include "ENCODE.v"

module NPC(PC, NPCop, Zero, IMM, Imm16, rs, NPC);  // next pc
module

input [31: 0] PC;        // pc
input [2: 0] NPCop;     // next pc operation
input Zero;
input [25: 0] IMM;       // immediate
input [15: 0] Imm16;
input [31: 0] rs;
output reg [31: 0] NPC;   // next pc

wire [31: 0] PCPLUS4;

assign PCPLUS4 = PC + 4; // pc + 4

always @( * )
    begin
      case (NPCop)
```

```
      `NPC_PLUS4:
        NPC = PCPLUS4;
      `NPC_BRANCH:
          begin
          if (Zero === 1'b1)
            NPC = PCPLUS4 + {{14{Imm16[15]}}, Imm16[15: 0], 2'
b00};
          else
            NPC = PCPLUS4;
          end
      `NPC_BNE:
          begin
          if (Zero != 1'b1)
            NPC = PCPLUS4 + {{14{Imm16[15]}}, Imm16[15: 0], 2'
b00};
          else
            NPC = PCPLUS4;
          end
      `NPC_JUMP:
        NPC = {PCPLUS4[31: 28], IMM[25: 0], 2'b00};
      `NPC_JAL:
        NPC = {PCPLUS4[31: 28], IMM[25: 0], 2'b00};
      `NPC_JALR:
        NPC = rs;
      `NPC_JR:
        NPC = rs;
      default:
        NPC = PCPLUS4;
      endcase
    end // end always

endmodule
```

描述说明：该部件的主要设计思想把所有和更新地址相关的控制信号和立即数等数据都送入该部件，然后在控制信号的控制下进行地址的更新。

4.5 IM（程序存储器）

实现代码：

```
module IM(addr, instr);

input [31: 0] addr;
output reg[31: 0] instr;

reg[31: 0] Instrs[1023: 0];

initial
    begin
    $readmemh("../test_codes/extendedtest.dat", Instrs, 0);
    //$readmemh("../test_codes/extendedtest.dat",Instrs,3000);
    //$readmemh("../test_codes/mipstest_extloop.dat",Instrs,0)
;
    //$readmemh("../test_codes/mipstest_extloop.dat",Instrs,30
00);
    //$readmemh("../test_codes/mipstestloop_sim.dat",Instrs,0)
;
    //$readmemh("../test_codes/mipstestloop_sim.dat",Instrs,30
00);
    //$readmemh("../test_codes/mipstestloopjal_sim.dat",Instrs
,0);
    //$readmemh("../test_codes/mipstestloopjal_sim.dat",Instrs
,3000);
    end

always@(addr)
    begin
    instr = Instrs[addr >> 2];
    //$display("Instrs[%4d]=0x%8h",pc,Instrs[pc]);
    end
```

```
endmodule
```

描述说明：和 PC 部件的设计类似，通过 initial 语句块中不同的语句可以以不同的起始地址将程序读取到指令存储器中，但该起始地址必须和 PC 中的起始地址匹配。同时将字节地址偏移成字地址的工作也是在该部件中完成。

4.6 RF（寄存器组）

实现代码：

```
module RF( input clk,
           input rst,
           input RFWr,
           input [4: 0] A1, A2, A3,
           input [31: 0] WD,
           output [31: 0] RD1, RD2
         );
//clk 是时钟信号，WD 是写会寄存器的数据，reset 是清零信号,RFWr 是写使能信号，A3 是写入的寄存器号

reg [31: 0] rf[31: 0];

integer i;

always @(posedge clk or posedge rst)
  if (rst)
      begin    //  reset
      for (i = 0; i < 32; i = i + 1)
        rf[i] <= 0;
      end

  else
    if (RFWr)
        begin
        rf[A3] <= WD;
```

```
        end


assign RD1 = (A1 != 0) ? rf[A1] : 0;
assign RD2 = (A2 != 0) ? rf[A2] : 0;

endmodule
```

描述说明：在单周期的 RF 部件设计中，并没有考虑结构冒险，所有并没有专门设计成为前半周期写后半周期读的方式。而是在整个时钟周期中都可以读写，但必须明确的是该设计只适用于单周期处理器，并不适用于流水线。

4.7 CONTROL（控制单元）

实现代码：

```
`include "ENCODE.v"
module CONTROL(Op, Func, RegDst, Jump, Branch, MemRead, MemtoR
eg, ALUop, MemWrite, ALUSrc, RegWrite, Shamt, Ext, PCSrc, MemC
ontrol);
//理论上来说 PCSrc 是来间接信号，但在现有设计中 PCSrc 由 CONTROL 产生

input [5: 0] Op;
input [5: 0] Func;

output reg[1: 0] RegDst; //写寄存器时的目标寄存器，为 01 时写 rd,为
00 时写 rt,为 10 时写第 31 号寄存器
output reg Jump; //跳转指令
output reg Branch; //分支指令
output reg MemRead; //读存储器
output reg[1: 0] MemtoReg; //为 01 时写入的数据来自数据存储器,为 00
时来自 ALU 计算的结果,为 10 时来自 PC+4
output reg[4: 0] ALUop;
output reg MemWrite; //为 1 时数据存储器写使能有效
output reg ALUSrc; //为 1 时 ALU 的第二个操作数来自符号扩展，为 0 时
来自 rt 默认为 0
output reg RegWrite; //为 1 时写入 register 有效
```

```
output reg Shamt; //为1 时ALU 的第一个操作数来自 shamt 字段
output reg[1: 0] Ext; //决定符号扩展的类型
output reg[2: 0] PCSrc;
output reg[3: 0] MemControl;

always@( * )
    begin
    RegDst <= 2'b01;
    Jump <= 1'b0;
    Branch <= 1'b0;
    MemRead <= 1'b0;
    MemtoReg <= 2'b00;
    ALUop <= `ALU_NOP;
    MemWrite <= 1'b0;
    ALUSrc <= 1'b0;
    RegWrite <= 1'b0;
    Shamt <= 1'b0;
    Ext <= `EXT_ZERO;
    PCSrc <= `NPC_PLUS4;
    MemControl <= `MEM_NOP;

      case (Op)
      `R_OP:
          begin
          RegDst <= 2'b01;
          RegWrite <= 1'b1;
            case (Func)
            `ADD_FUNCT:
                begin
                ALUop <= `ALU_ADD;
                end
            `ADDU_FUNCT:
                begin
                ALUop <= `ALU_ADDU;
```

```
            end
        `AND_FUNCT:
            begin
            ALUop <= `ALU_AND;
            end
        `SUB_FUNCT:
            begin
            ALUop <= `ALU_SUB;
            end
        `SUBU_FUNCT:
            begin
            ALUop <= `ALU_SUBU;
            end
        `NOR_FUNCT:
            begin
            ALUop <= `ALU_NOR;
            end
        `OR_FUNCT:
            begin
            ALUop <= `ALU_OR;
            end
        `XOR_FUNCT:
            begin
            ALUop <= `ALU_XOR;
            end
        `SLT_FUNCT:
            begin
            ALUop <= `ALU_SLT;
            end
        `SLTU_FUNCT:
            begin
            ALUop <= `ALU_SLTU;
            end
        `SLL_FUNCT:
```

```
            begin
            ALUop <= `ALU_SLL;
            Shamt <= 1'b1;
            end
        `SRL_FUNCT:
            begin
            ALUop <= `ALU_SRL;
            Shamt <= 1'b1;
            end
        `SRA_FUNCT:
            begin
            ALUop <= `ALU_SRA;
            Shamt <= 1'b1;
            end
        `SLLV_FUNCT:
            begin
            ALUop <= `ALU_SLLV;
            end
        `SRLV_FUNCT:
            begin
            ALUop <= `ALU_SRLV;
            end
        `SRAV_FUNCT:
            begin
            ALUop <= `ALU_SRAV;
            end
        `JALR_FUNCT:
            begin
            MemtoReg <= 2'b10;
            PCSrc <= `NPC_JALR;
            end
        `JR_FUNCT:
            begin
            PCSrc <= `NPC_JR;
```

```
                RegWrite <= 1'b0;
                end
          endcase
        end
    `ADDI_OP:
        begin
        RegDst <= 2'b00;
        ALUop <= `ALU_ADD;
        ALUSrc <= 1;
        RegWrite <= 1'b1;
        Ext <= `EXT_SIGNED;
        end
    `ORI_OP:
        begin
        RegDst <= 2'b00;
        ALUop <= `ALU_OR;
        ALUSrc <= 1;
        RegWrite <= 1'b1;
        Ext <= `EXT_SIGNED;
        end
    `ANDI_OP:
        begin
        RegDst <= 2'b00;
        ALUop <= `ALU_AND;
        ALUSrc <= 1;
        RegWrite <= 1'b1;
        Ext <= `EXT_SIGNED;
        end
    `LUI_OP:
        begin
        RegDst <= 2'b00;
        ALUop <= `ALU_ADD;
        ALUSrc <= 1'b1;
        RegWrite <= 1'b1;
```

```
            Ext <= `EXT_HIGHPOS;
            end
        `SLTI_OP:
            begin
            RegDst <= 2'b00;
            ALUop <= `ALU_SLT;
            ALUSrc <= 1'b1;
            RegWrite <= 1'b1;
            Ext <= `EXT_SIGNED;
            end
        `BEQ_OP:
            begin
            PCSrc <= `NPC_BRANCH;
            end
        `BNE_OP:
            begin
            PCSrc <= `NPC_BNE;
            end
        `SW_OP:
            begin
            ALUop <= `ALU_ADD;
            MemWrite <= 1'b1;
            ALUSrc <= 1'b1;
            Ext <= `EXT_SIGNED;
            MemControl <= `MEM_SW;
            end
        `SB_OP:
            begin
            ALUop <= `ALU_ADD;
            MemWrite <= 1'b1;
            ALUSrc <= 1'b1;
            Ext <= `EXT_SIGNED;
            MemControl <= `MEM_SB;
            end
```

```
        `SH_OP:
            begin
            ALUop <= `ALU_ADD;
            MemWrite <= 1'b1;
            ALUSrc <= 1'b1;
            Ext <= `EXT_SIGNED;
            MemControl <= `MEM_SH;
            end
        `LW_OP:
            begin
            RegDst <= 2'b00;
            MemRead <= 1'b1;
            MemtoReg <= 2'b01;
            ALUop <= `ALU_ADD;
            ALUSrc <= 1'b1;
            RegWrite <= 1'b1;
            Ext <= `EXT_SIGNED;
            MemControl <= `MEM_LW;
            end
        `LB_OP:
            begin
            RegDst <= 2'b00;
            MemRead <= 1'b1;
            MemtoReg <= 2'b01;
            ALUop <= `ALU_ADD;
            ALUSrc <= 1'b1;
            RegWrite <= 1'b1;
            Ext <= `EXT_SIGNED;
            MemControl <= `MEM_LB;
            end
        `LH_OP:
            begin
            RegDst <= 2'b00;
            MemRead <= 1'b1;
```

```
            MemtoReg <= 2'b01;
            ALUop <= `ALU_ADD;
            ALUSrc <= 1'b1;
            RegWrite <= 1'b1;
            Ext <= `EXT_SIGNED;
            MemControl <= `MEM_LH;
            end
        `LBU_OP:
            begin
            RegDst <= 2'b00;
            MemRead <= 1'b1;
            MemtoReg <= 2'b01;
            ALUop <= `ALU_ADD;
            ALUSrc <= 1'b1;
            RegWrite <= 1'b1;
            Ext <= `EXT_SIGNED;
            MemControl <= `MEM_LBU;
            end
        `LHU_OP:
            begin
            RegDst <= 2'b00;
            MemRead <= 1'b1;
            MemtoReg <= 2'b01;
            ALUop <= `ALU_ADD;
            ALUSrc <= 1'b1;
            RegWrite <= 1'b1;
            Ext <= `EXT_SIGNED;
            MemControl <= `MEM_LHU;
            end
        `J_OP:
            begin
            PCSrc <= `NPC_JUMP;
            end
        `JAL_OP:
```

```
            begin
            RegDst <= 2'b10;
            MemtoReg <= 2'b10;
            RegWrite <= 1'b1;
            PCSrc <= `NPC_JAL;
            end

        endcase
    end

endmodule
```

描述说明：

该部件主要的实现思路是在一个部件中生成所有的控制信号，而不像教材中对于 ALUOp 这样采取的多级译码的方式。同时在部件的具体实现上采用了较为抽象和语义明确的 case 语句，从而避免了复杂而庞大的卡诺图和逻辑函数。

4.8 ALU（算术运算单元）

实现代码：

```
`include "ENCODE.v"

module ALU(A, B, ALUOp, C, Zero);

input signed [31: 0] A, B;
input [4: 0] ALUOp;
output signed [31: 0] C;
output Zero;
//ALU 的输入和输出均为有符号数，在 ALU 内部运算的时候会根据情况转换成无符号数

reg [31: 0] C;
integer i;


always @( A or B or ALUOp )
```

```
    begin
      case ( ALUOp )
      `ALU_NOP:
        C = A;
      `ALU_ADDU:
        C = $unsigned(A) + $unsigned(B);
      `ALU_ADD:
        C = A + B;
      `ALU_SUBU:
        C = $unsigned(A) - $unsigned(B);
      `ALU_SUB:
        C = A - B;
      `ALU_AND:
        C = A & B;
      `ALU_OR:
        C = A | B;
      `ALU_NOR:
        C = ~(A | B);
      `ALU_XOR:
        C = (A ^ B);
      `ALU_SLT:
        C = A < B ? 1 : 0;
      `ALU_SLTU:
        C = $unsigned(A) < $unsigned(B) ? 1 : 0;
      `ALU_SLL:
        C = B << A;
      `ALU_SLLV:
        C = B << A[4: 0];
      `ALU_SRL:
        C = B >> A;
      `ALU_SRLV:
        C = B >> A[4: 0];
      `ALU_SRA:
        C = B >>> A;
```

```
      `ALU_SRAV:
        C = B >>> A[4: 0];
      //其中>>>是verilog原生运算符，表示算术右移

      default:
        C = A;                          // Undefined
      endcase
    end

assign Zero = (C == 32'b0);

endmodule
```

描述说明：ALU 运算单元的设计尽量采用 verilog 语法中的原生运算符，最为典型的例子是“>>>”,算术右移运算符。这样做的好处是能尽量简化代码减少 bug，将硬件较为底层的实现交由 verilog 编译器来完成，从而保证了 ALU 单元设计的可靠性。

4.9 MEM（数据存储器）

实现代码：

```
`include "ENCODE.v"

module MEM(clk, rst, input_address, input_data, Wr, MemControl
, output_data);

input clk;
input rst;
input[31: 0] input_address;
input[31: 0] input_data;
input Wr;
input[3: 0] MemControl;
output reg[31: 0] output_data;

reg [9: 0] word_addr;
reg [1: 0] byte_addr;
```

```
reg [31: 0] word;
reg [31: 0] data_memory[1023: 0];
integer i;

always@( * )
    begin
    word_addr = input_address[11: 2];
    byte_addr = input_address[1: 0];
    word = data_memory[word_addr[9: 0]];
      case (MemControl)
      `MEM_LW:
          begin
          output_data = data_memory[word_addr[9: 0]];
          end
      `MEM_LB:
          begin
            case (byte_addr)
            2'b00:
                begin
                output_data = {{24{word[7]}}, word[7: 0]};
                end
            2'b01:
                begin
                output_data = {{24{word[15]}}, word[15: 8]};
                end
            2'b10:
                begin
                output_data = {{24{word[23]}}, word[23: 16]};
                end
            2'b11:
                begin
                output_data = {{24{word[31]}}, word[31: 24]};
                end
            endcase
```

```
        end
    `MEM_LH:
        begin
          case (byte_addr)
          2'b00:
              begin
              output_data = {{16{word[15]}}, word[15: 0]};
              end
          2'b10:
              begin
              output_data = {{16{word[31]}}, word[31: 16]};
              end
          endcase
        end
    `MEM_LBU:
        begin
          case (byte_addr)
          2'b00:
              begin
              output_data = {{24{1'b0}}, word[7: 0]};
              end
          2'b01:
              begin
              output_data = {{24{1'b0}}, word[15: 8]};
              end
          2'b10:
              begin
              output_data = {{24{1'b0}}, word[23: 16]};
              end
          2'b11:
              begin
              output_data = {{24{1'b0}}, word[31: 24]};
              end
          endcase
```

```
          end
      `MEM_LHU:
          begin
            case (byte_addr)
            2'b00:
                begin
                output_data = {{16{1'b0}}, word[15: 0]};
                end
            2'b10:
                begin
                output_data = {{16{1'b0}}, word[31: 16]};
                end
            endcase
          end
      endcase
    end

always @(posedge clk or posedge rst)
    begin
    word_addr = input_address[11: 2];
    byte_addr = input_address[1: 0];
    word = data_memory[word_addr[9: 0]];
    if (rst)
        begin
        for (i = 0;i < 1024;i = i + 1)
          data_memory[i] <= 32'h0000_0000;
        end
    if (Wr == 1'b1)
        begin
          case (MemControl)
          `MEM_SW:
              begin
              data_memory[word_addr[9: 0]] <= input_data[31: 0
];
```

```
            end
        `MEM_SH:
            begin
              case (byte_addr)
              2'b00:
                  begin
                  data_memory[word_addr[9: 0]] = {word[31: 16], input_data[15: 0]};
                  end
              2'b10:
                  begin
                  data_memory[word_addr[9: 0]] = {input_data[15: 0], word[15: 0]};
                  end
              endcase
            end
        `MEM_SB:
            begin
              case (byte_addr)
              2'b00:
                  begin
                  data_memory[word_addr[9: 0]] = {word[31: 8], input_data[7: 0]};
                  end
              2'b01:
                  begin
                  data_memory[word_addr[9: 0]] = {word[31: 16], input_data[7: 0], word[7: 0]};
                  end
              2'b10:
                  begin
                  data_memory[word_addr[9: 0]] = {word[31: 24], input_data[7: 0], word[15: 0]};
                  end
```

```
                2'b11:
                    begin
                    data_memory[word_addr[9: 0]] = {input_data
[7: 0], word[23: 0]};
                    end
                endcase
            end
          endcase
        end
    end

endmodule
```

描述说明：必须承认这个 MEM 部件设计的并不是很优雅，主要是因为按照课堂上的要求，存储器的最小存储单元位一个字，故为了添加对 lb, sb, sh, lh 这些非整个字的存取指令的支持，必须设置额外的控制信号，和一些额外的逻辑，比如将一个字读取出来后修改其中的一部分字节再存进去，在代码层面设计得并不好，实际运行的时候可想而知效率也会很低。

4.10 EXT（符号扩展单元）

实现代码：

```
`include "ENCODE.v"

module EXT( Imm16, EXTOp, Imm32 );

input [15: 0] Imm16;
input [1: 0] EXTOp;
output [31: 0] Imm32;

reg [31: 0] Imm32;

always@( * )
    begin
      case (EXTOp)
      `EXT_ZERO:
```

```
        Imm32 = {16'd0, Imm16};
      `EXT_SIGNED:
        Imm32 = {{16{Imm16[15]}}, Imm16};
      `EXT_HIGHPOS:
        Imm32 = {Imm16, 16'd0};
      default:
        ;
      endcase
    end
endmodule
```

描述说明：该部件较为简单，主要是实现了对不同符号扩展类型的支持。

5 单周期测试及结果分析

5.1 仿真代码及分析

仿真代码如图 5-1 所示

```
# Attention: Mars, Settings -> Memory Configuration -> Compact, Data at address 0
# lui ori subu addu add sub nor or and slt stlu addi
# sll srl sra sllv srlv srav
# sw sh sb
# lw lh lhu lb lbu

lui   $3, 0x9876       # $3=0x98760000                          # 3c039876
ori   $2, $0, 0x1234   # $2=0x1234                              # 34021234
subu  $8, $3, $2       # $8=0x98760000-0x1234=0x9875edcc        # 00624023
xor   $9, $8, $3       # $9=0x9875edcc^0x98760000=0x0003edcc    # 01034826
addu  $10,$9, $8       # $10=0x0003edcc+0x9875edcc=0x9879db98   # 01285021
add   $10, $10, $2     # $10=0x9879db98+0x1234=0x9879edcc       # 01425020
sub   $11, $10, $3     # $11=0x9879edcc-0x98760000=0x0003edcc   # 01435822
nor   $12, $11, $10    # $12=~(0x0003edcc|0x9879edcc)=0x67841233  # 016a6027
or    $13, $11, $10    # $13=0x0003edcc|0x9879edcc=0x987bedcc   # 016a6825
and   $14, $11, $10    # $14=0x0003edcc&0x9879edcc=0x0001edcc   # 016a7024
slt   $19, $13, $12    # $19=(0x987bedcc<0x67841233)=1          # 01ac982a
sltu  $20, $13, $12    # $20=(0x987bedcc<0x67841233)=0          # 01aca02b
sll   $8, $8, 3        # $8=0x9875edcc<<3=0xc3af6e60            # 000840c0
srl   $9, $8, 0x10     # $9=0xc3af6e60>>16=0xc3af               # 00084c02
sra   $10, $8, 0x1d    # $10=0xc3af6e60>>>29=0xfffffffe         # 00085743
ori   $11, $0, 0x3410  # $11=0x3410                             # 340b3410
sllv  $12, $8, $11     # $12=0xc3af6e60<<16=0x6e600000          # 01686004
srlv  $13, $8, $11     # $13=0xc3af6e60>>16=0xc3af              # 01686806
srav  $14, $8, $11     # $14=0xc3af6e60>>>16=0xffffc3af         # 01687007
addu  $4, $2, $3       # $4=0x1234+0x98760000=0x98761234        # 00432021
addi  $29, $0, 0x0     # $29=0                                  # 201d0000
sw    $4, 0x00($29)    # [0]=0x98761234                         # afa40000
sw    $4, 0x04($29)    # [4]=0x98761234                         # afa40004
sw    $4, 0x08($29)    # [8]=0x98761234                         # afa40008
sh    $8, 0x04($29)    # [4]=0x98766e60 $8=0xc3af6e60           # a7a80004  little endian
sh    $9, 0x0a($29)    # [8]=0xc3af1234 $9=0xc3af               # a7a9000a  little endian
sb    $10, 0x07($29)   # [4]=0xfe766e60 $10=0xfffffffe          # a3aa0007  little endian
sb    $8, 0x09($29)    # [8]=0xc3af6034 $8=0xc3af6e60           # a3a80009  little endian
sb    $9, 0x08($29)    # [8]=0xc3af60af $9=0xc3af               # a3a90008  little endian
lw    $8, 0x00($29)    # $8=0x98761234                          # 8fa80000
sw    $8, 0x0c($29)    # [0xc]=0x98761234                       # afa8000c
lh    $9, 0x02($29)    # $9=0xffff9876 [0]=0x98761234           # 87a90002  little endian, sign extension
sw    $9, 0x10($29)    # [0x10]=0xffff9876                      # afa90010
lhu   $9, 0x02($29)    # $9=0x00009876 [0]=0x98761234           # 97a90002  little endian, zero extension
sw    $9, 0x14($29)    # [0x14]=0x00009876                      # afa90014
lb    $10, 0x03($29)   # $10=0xffffff98  [0]=0x98761234         # 83aa0003  little endian, sign extension
sw    $10, 0x18($29)   # [0x18]=0xffffff98                      # afaa0018
lbu   $10, 0x03($29)   # $10=0x00000098 [0]=0x98761234          # 93aa0003  little endian, zero extension
sw    $10, 0x1c($29)   # [0x1c]=0x00000098                      # afaa001c
lbu   $10, 0x01($29)   # $10=0x00000012 [0]=0x98761234          # 93aa0001  little endian, zero extension
sw    $10, 0x20($29)   # [0x20]=0x00000012                      # afaa0020


# $0=0              $1=0              $2=0x00001234     $3=0x98760000
# $4=0x98761234     $5=0              $6=0              $7=0
# $8=0x98761234     $9=0x00009876     $10=0x00000012    $11=0x00003410
# $12=0x6e600000    $13=0x0000c3af    $14=0xffffc3af    $19=0x0000001
# $29=0
# [0]=0x98761234    [4]=0xfe766e60    [8]=0xc3af60af    [0xc]=0x98761234
# [0x10]=0xffff9876 [0x14]=0x00009876 [0x18]=0xffffff98 [0x1c]=0x00000098
```

图 5-1

由于是单周期 CPU 的仿真，故不存在数据依赖，数据冒险，控制冒险等问题。

5.2 仿真测试结果

其中在 Mars 中的仿真结果如图 5-2 所示：

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x00000007
$v1	3	0x0000000c
$a0	4	0x00000001
$a1	5	0x0000000b
$a2	6	0x00000000
$a3	7	0x00000007
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x00001800
$sp	29	0x00003ffc
$fp	30	0x00000000
$ra	31	0x00000014
pc		0x00000084
hi		0x00000000
lo		0x00000000

图 5-2

其中在 modelsim 中的仿真结果如图 5-3 所示

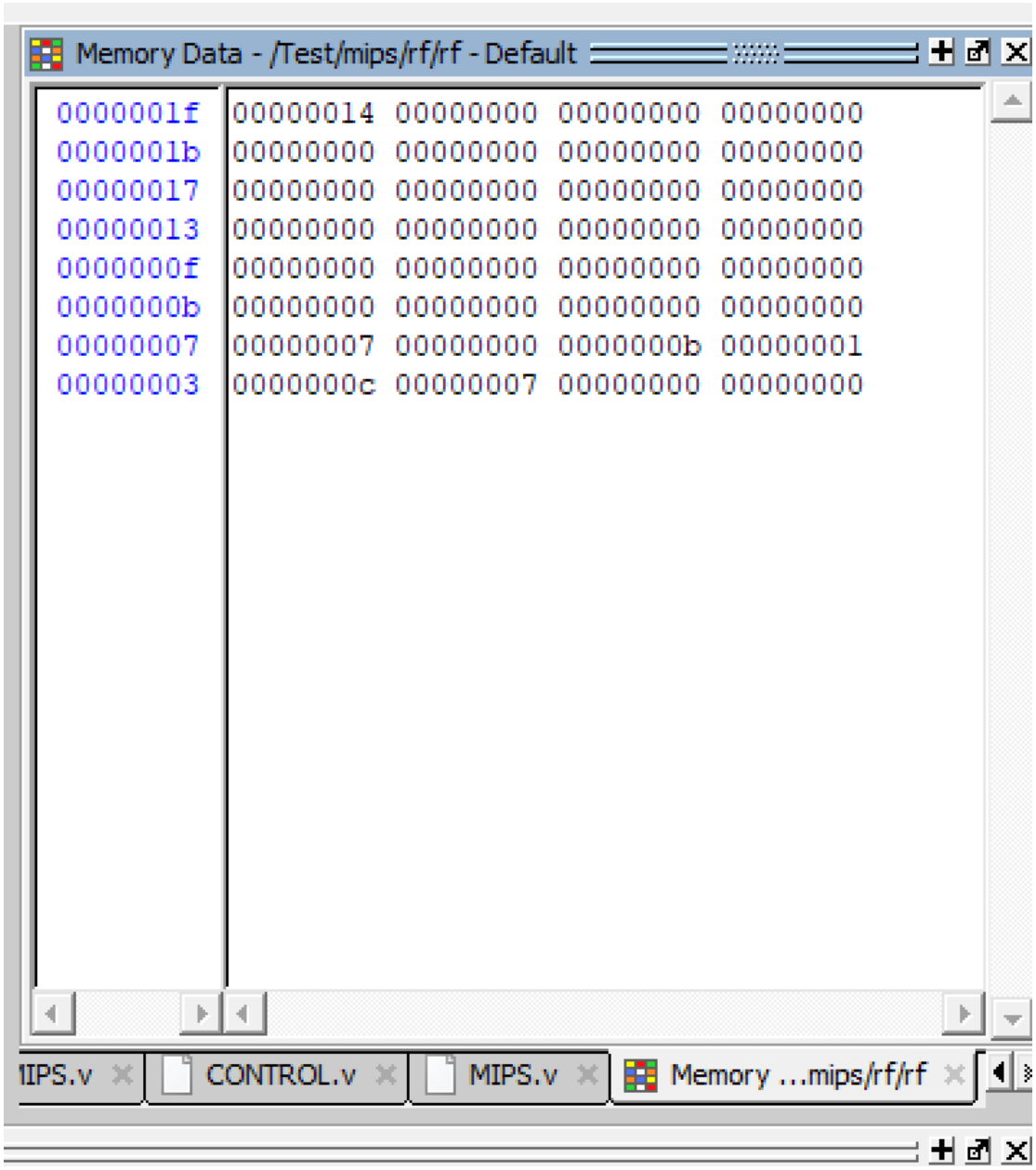

图 5-3

分析如下：可以看到寄存器中的结果一致，故可以证明支持测试程序集中的指令。

6 流水线概要设计

6.1 总体设计

该 CPU 采取流水线的设计，流水线共分为 5 级，即 IF/ID/EX/MEM/WB 五个阶

段，其中 IF 阶段进行取指令和更新 PC，在 ID 阶段进行译码和访问寄存器堆，在 EX 阶段进行 ALU 的运算，在 MEM 阶段访问数据存储器，写入或读取出数据，在 WB 阶段将数据写回到寄存器中。该流水线实现了完全的旁路，能解决大部分的数据冒险（但仍有少数数据冒险需要通过阻塞流水线来解决），同时实现了静态预测，假定分支不发生，能够解决控制冒险。支持的基本指令如下：add/sub/and/or/slt/sltu/addu/subu/addi/ori/lw/sw/beq/j/jal。支持的扩展指令如下：sll/nor/lui/slti/bne/andi/srl/sllv/srlv/jr/jalr/xor/sra/srav/lb/lh/lbu/lhu/sb/sh

6.2 ENCODE（预定义单元）

6.2.1 功能描述

通过 verilog 中的预编译语句，将指令的操作码，功能码，以及 CPU 设计中的控制信号从二进制的数字唯一映射成字符标识的预定义宏以提高代码的可读性。对应的文件为 ENCODE.v。该部件完全复用自单周期 CPU 中的部件设计，功能和设计与单周期 CPU 中完全一样。

6.2.2 模块接口

该模块并不是具体的实体部件设计，故没有模块接口表。

6.3 PC（程序计数器）

6.3.1 功能描述

PC 主要负责接受 NPC 部件所更新的程序地址，以产生下一条指令的地址。对应的文件为 PC.v，在顶层设计中的实体名为 pc。该部件完全复用自单周期 CPU 中的部件设计，功能和设计与单周期 CPU 中完全一样。

6.3.2 模块接口

信号名	方向	描述
clk	输入	时钟信号
rst	输入	重置信号
NPC[31:0]	输入	由 NPC 部件产生的 pc 更新的值
PC[31:0]	输出	更新后的 pc

6.4 NPC（程序计数器更新单元）

6.4.1 功能描述

NPC 主要负责接受当前 PC 的值以及如 16 位和 26 位立即数等和 PC 更新相关的值，来进行 PC 的更新。除了实现 PC+4 之外，还有根据 PC 更新的控制信号以实现分支和跳转的功能。对应的文件名为 NPC.v，在顶层设计中的实体名为 npc。该部件部分复用于单周期 CPU 中的 NPC，改动主要是加入了 ID_PC 和 harzard，主要为了解决数据冒险和控制冒险中的地址更新问题。

6.4.2 模块接口

信号名	方向	描述
PC[31:0]	输入	需要更新的 pc 的值
ID_PC[31:0]	输入	处于 ID 阶段的 pc 的值
NPCop[2:0]	输入	控制 pc 进行+4 或分支或跳转的控制信号
Zero	输入	由 ALU 产生的零信号
IMM	输入	分支指令中的 16 位立即数
Imm16	输入	跳转指令中的 26 位立即数
rs	输入	从寄存器中读取出的跳转的目的地址
harzard	输入	表明是否有冒险发生
NPC	输出	更新后的 pc 的值

6.5 IM（程序存储器）

6.5.1 功能描述

存储机器指令，并根据 pc 的值读取出当前执行的指令。对应的文件名为 IM.v，在顶层设计中的实体名为 im。该部件完全复用自单周期 CPU 中的部件设计，功能和设计与单周期 CPU 中完全一样。

6.5.2 模块接口

信号名	方向	描述
Addr[31:0]	输入	当前 pc 的值
Instr[31:0]	输出	由 pc 的值读取出来的指令

6.6 RF（寄存器组）

6.6.1 功能描述

存储和修改 32 个 32 位寄存器中的数据，并能同时读取 2 个寄存器中的 32 位数据和在 RegWrite 信号的控制下将 32 位数据写入一个 32 位寄存器。对应的文件名为 RF.v,在顶层设计中的实体名为 rf。该部件部分复用自单周期 CPU 中的部件设计，主要改动在于修改了其允许写入寄存器的时机，从而实现前半周期写寄存器，后半周期读寄存器的功能，其他功能与单周期 CPU 中完全一样。

6.6.2 模块接口

信号名	方向	描述
clk	输入	时钟信号
rst	输入	重置信号
RFWr	输入	写使能信号
A1[4:0]	输入	读取的第一个寄存器的寄存器号
A2[4:0]	输入	读取的第二个寄存器的寄存器号
A3[4:0]	输入	待写入数据的寄存器的寄存器号
WD[31:0]	输入	待写入寄存器的 32 位数据
RD1[31:0]	输出	读取的第一个寄存器的数据
RD2[31:0]	输出	读取的第二个寄存器的数据

6.7 CONTROL（控制单元）

6.7.1 功能描述

根据 32 位指令的功能码和操作码字段生成控制信号，对应的文件名为 CONTROL.v,顶层设计中的实体名为 control。部件完全复用自单周期 CPU 中的部件设计，功能和设计与单周期 CPU 中完全一样。

6.7.2 模块接口

信号名	方向	描述
Op[5:0]	输入	指令的操作码
Func[5:0]	输入	指令的功能码
RegDst[1:0]	输出	写寄存器时的目标寄存器
Jump	输出	跳转信号
Branch	输出	分支信号
MemRead	输出	读数据存储器的信号
MemtoReg[1:0]	输出	写寄存器时写入数据的来源
ALUop[4:0]	输出	控制 ALU 运算的类型
MemWrite	输出	写数据存储器的信号
ALUSrc	输出	控制 ALU 的第二个操作数的来源
RegWrite	输出	写寄存器信号
Shamt	输出	控制 ALU 的第一个操作数是否来自 shamt
Ext[1:0]	输出	符号扩展的类型
PCSrc[2:0]	输出	分支或跳转的类型
MemControl[3:0]	输出	存储和加载指令的类型

6.8 ALU（算术运算单元）

6.8.1 功能描述

根据输入 ALU 运算类型的控制信号，对两个 ALU 的源操作数进行不同类型的运算，对应的文件名为 ALU.v,顶层设计中的实体名为 alu。部件完全复用自单周期 CPU 中的部件设计，功能和设计与单周期 CPU 中完全一样。

6.8.2 模块接口

信号名	方向	描述
A	输入	第一个操作数
B	输入	第二个操作数
ALUOp	输入	控制 ALU 运算类型的控制信号
C	输出	运算结果
Zero	输出	判断两个源操作数是否相等的零信号

6.9 MEM（数据存储器）

6.9.1 功能描述

写入、读取和存储数据，其中写入和读取的端口是共用的，只输入一个地址，在写使能和读使能的信号的控制下分别进行写和读的操作。对应的文件名为MEM.v,顶层设计中的实体名为mem。该部件部分复用自单周期CPU中的部件设计，主要改动在于修改了其允许写入存储器的时机，从而实现前半周期写存储器，后半周期读存储器的功能，其他功能与单周期CPU中完全一样。

6.9.2 模块接口

信号名	方向	描述
clk	输入	时钟信号
rst	输入	重置信号
Input_address[31:0]	输入	写入或读取的端口号
Input_data[31:0]	输入	写入的32位数据
Wr	输入	写使能信号
MemControl	输入	存储和加载指令的类型
Output_data	输出	读取出来的32位数据

6.10 EXT（符号扩展单元）

6.10.1 功能描述

根据符号扩展的控制信号将16位立即数扩展成32位的数据，对应的文件名为EXT.v,顶层设计中的实体名为ext。部件完全复用自单周期CPU中的部件设计，功能和设计与单周期CPU中完全一样。

6.10.2 模块接口

信号名	方向	描述
Imm16	输入	16位立即数

EXTOp	输入	控制符号扩展类型的控制信号
Imm32	输出	符号扩展后的 32 位数据

6.11 IF_ID（IF_ID 流水线寄存器）

6.11.1 功能描述

该寄存器中暂时存储 IF 阶段执行的指令的地址和指令内容。同时根据 harzard 信号来选择是否清空流水线寄存器中的内容，也即生成一条空指令（nop），阻塞流水线一个时钟周期。

6.11.2 模块接口

信号名	方向	描述
clk	输入	时钟信号
rst	输入	重置信号
PC[31:0]	输入	IF 阶段执行的指令的地址
Instr[31:0]	输入	IF 阶段执行的指令
harzard	输入	表明是否有冒险
PC_out	输出	IF 阶段执行的指令的地址
Instr_out	输出	IF 阶段执行的指令

6.12 ID_EX（ID_EX 流水线寄存器）

6.12.1 功能描述

该流水线寄存器主要暂存来自 ID 阶段的数据和所有后面阶段要用到的控制信号。

6.12.2 模块接口

信号名	方向	描述

clk	输入	时钟信号
rst	输入	重置信号
PC[31:0]	输入	ID 阶段执行的指令的地址
RD1[31:0]	输入	ID 阶段寄存器组中读取出的第一个数据
RD2[31:0]	输入	ID 阶段寄存器组中读取出的第二个数据
IMM32[31:0]	输入	立即数符号扩展后的 32 位数据
rs[4:0]	输入	第一个源操作数寄存器号
rt[4:0]	输入	第二个源操作数寄存器号
RF_rd[4:0]	输入	写入寄存器的寄存器号
shamt[4:0]	输入	偏移量
MemRead	输入	读数据存储器的信号
MemtoReg[1:0]	输入	写寄存器时写入数据的来源
ALUop[4:0]	输入	控制 ALU 运算的类型
MemWrite	输入	写数据存储器的信号
ALUSrc	输入	控制 ALU 的第二个操作数的来源
RegWrite	输入	写寄存器信号
Shamt	输入	控制 ALU 的第一个操作数是否来自 shamt
PCSrc[2:0]	输入	分支或跳转的类型
MemControl[3:0]	输入	存储和加载指令的类型
PC_out[31:0]	输出	ID 阶段执行的指令的地址
RD1_out[31:0]	输出	ID 阶段寄存器组中读取出的第一个数据
RD2_out[31:0]	输出	ID 阶段寄存器组中读取出的第二个数据
IMM32_out[31:0]	输出	立即数符号扩展后的 32 位数据
rs_out[4:0]	输出	第一个源操作数寄存器号
rt_out[4:0]	输出	第二个源操作数寄存器号
RF_rd_out[4:0]	输出	写入寄存器的寄存器号
shamt_out[4:0]	输出	偏移量
MemRead_out	输出	读数据存储器的信号
MemtoReg_out[1:0]	输出	寄存器时写入数据的来源
ALUop_out [4:0]	输出	控制 ALU 运算的类型
MemWrite_out	输出	写数据存储器的信号
ALUSrc_out	输出	控制 ALU 的第二个操作数的来源
RegWrite_out	输出	写寄存器信号
Shamt_out	输出	控制 ALU 的第一个操作数是否来自 shamt
PCSrc_out [2:0]	输出	分支或跳转的类型
MemControl_out [3:0]	输出	存储和加载指令的类型

6.13 EX_MEM（EX_MEM 流水线寄存器）

6.13.1 功能描述

该寄存器主要暂存来自 EX 阶段的数据和后面所有阶段要用到的控制信号。

6.13.2 模块接口

信号名	方向	描述
clk	输入	时钟信号
rst	输入	重置信号
PC[31:0]	输入	EX 阶段执行的指令的地址
ALU_result[31:0]	输入	ALU 的计算机结果
RD2[31:0]	输入	寄存器组中读取出的第二个数据
RF_rd[4:0]	输入	写入寄存器的寄存器号
MemRead	输入	读数据存储器的信号
MemtoReg[1:0]	输入	写寄存器时写入数据的来源
MemWrite	输入	写数据存储器的信号
RegWrite	输入	写寄存器信号
PCSrc[2:0]	输入	分支或跳转的类型
MemControl[3:0]	输入	存储和加载指令的类型
PC_out[31:0]	输出	EX 阶段执行的指令的地址
ALU_result_out[31:0]	输出	ALU 的计算机结果
RD2_out[31:0]	输出	寄存器组中读取出的第二个数据
RF_rd_out[4:0]	输出	写入寄存器的寄存器号
MemRead_out	输出	读数据存储器的信号
MemtoReg_out[1:0]	输出	寄存器时写入数据的来源
MemWrite_out	输出	写数据存储器的信号
RegWrite_out	输出	写寄存器信号
PCSrc_out [2:0]	输出	分支或跳转的类型
MemControl_out [3:0]	输出	存储和加载指令的类型

6.14 MEM_WB（MEM_WB 流水线寄存器）

6.14.1 功能描述

该寄存器主要暂存来自 MEM 阶段的数据和 WB 阶段要用到的控制信号。

6.14.2 模块接口

信号名	方向	描述
clk	输入	时钟信号
rst	输入	重置信号
PC[31:0]	输入	EX 阶段执行的指令的地址
ALU_result[31:0]	输入	ALU 的计算机结果
Mem_data[31:0]	输入	从数据存储器中读取出来的数据
RF_rd[4:0]	输入	写入寄存器的寄存器号
MemtoReg[1:0]	输入	写寄存器时写入数据的来源
RegWrite	输入	写寄存器信号
PCSrc[2:0]	输入	分支或跳转的类型
PC_out[31:0]	输出	EX 阶段执行的指令的地址
ALU_result_out[31:0]	输出	ALU 的计算机结果
Mem_data _out[31:0]	输出	从数据存储器中读取出来的数据
RF_rd_out[4:0]	输出	写入寄存器的寄存器号
MemtoReg_out[1:0]	输出	寄存器时写入数据的来源
RegWrite_out	输出	写寄存器信号
PCSrc_out [2:0]	输出	分支或跳转的类型

6.15 FORWARD（旁路单元）

6.15.1 功能描述

进行旁路的检测，同时工作在 EX 和 ID 级，分别生成四个旁路信号 ForwardA~D

6.15.2 模块接口

信号名	方向	描述
ID_rs	输入	ID 阶段第一个源操作数寄存器号
ID_rt	输入	ID 阶段第二个源操作数寄存器号
EX_rs	输入	EX 阶段第一个源操作数寄存器号
EX_rt	输入	EX 阶段第二个源操作数寄存器号
EX_RF_rd	输入	EX 阶段目的寄存器号
MEM_RF_rd	输入	MEM 阶段目的寄存器号
WB_RF_rd	输入	WB 阶段目的寄存器号
EX_RegWrite	输入	EX 阶段写寄存器信号
MEM_RegWrite	输入	MEM 阶段写寄存器信号
WB_RegWrite	输入	WB 阶段写寄存器信号
MEM_MemRead	输入	MEM 阶段读存储器信号
ForwardA	输出	旁路信号 A
ForwardB	输出	旁路信号 B

ForwardC	输出	旁路信号 C
ForwardD	输出	旁路信号 D

6.16 HARZARD（冒险检测单元）

6.16.1 功能描述

进行控制冒险和数据冒险的检测，分别生成两个与冒险相关的信号，分别控制清空 IF_ID 流水线寄存器中的内容，生成一个空指令(nop)，和控制 pc 的值保持不变的信号。

6.16.2 模块接口

信号名	方向	描述
ID_rs	输入	ID 阶段第一个源操作数寄存器号
ID_rt	输入	ID 阶段第二个源操作数寄存器号
EX_rt	输入	EX 阶段第二个源操作数寄存器号
ID_PCSrc	输入	ID 阶段 PC 跳转和分支的控制信号
EX_MemRead	输入	EX 阶段读数据存储器信号
IF_ID_flush	输出	清空 IF_ID 流水线寄存器信号
PC_unchanged	输出	保持 PC 值不变的信号

6.17 MIPS（顶层文件）

6.17.1 功能描述

CPU 设计中的顶层文件，负责将上述各部件组装起来。对应的文件名为 MIPS.v，顶层设计中的实体名为 mips。

6.17.2 模块接口

信号名	方向	描述
clk	输入	时钟信号
rst	输入	重置信号

7　流水线详细设计

7.1 CPU 总体结构

代码实现：

```
`include "ENCODE.v"

module MIPS(clk, rst);

input clk;
input rst;

//IF 阶段
wire [31: 0] IF_PC;
wire [31: 0] IF_NPC;
wire [31: 0] IF_Instr;

//ID 阶段
wire [31: 0] ID_PC;
wire [31: 0] ID_Instr;
wire [31: 0] ID_Imm32;
wire [31: 0] ID_RD1;
wire [31: 0] ID_RD2;
//拆分指令
wire [5: 0] ID_Op;
wire [5: 0] ID_Funct;
wire [4: 0] ID_rs;
wire [4: 0] ID_rt;
wire [4: 0] ID_rd;
wire [4: 0] ID_RF_rd;
wire [15: 0] ID_Imm16;
wire [25: 0] ID_IMM;
wire [4: 0] ID_shamt;
//由控制器产生的控制信号
wire[1: 0] ID_RegDst;
```

```
wire ID_MemRead;
wire[1: 0] ID_MemtoReg;
wire[4: 0] ID_ALUop;
wire ID_MemWrite;
wire ID_ALUSrc;
wire ID_RegWrite;
wire ID_Shamt;
wire[1: 0] ID_Ext;
wire[2: 0] ID_PCSrc;
wire[3: 0] ID_MemControl;
wire ID_Jump;
wire ID_Branch;

wire IF_ID_flush;
wire PC_unchanged;
wire ID_zero;

wire [31: 0] ID_RD1_forward;
wire [31: 0] ID_RD2_forward;

//EX 阶段
//从 ID/EX 寄存器中读取出来的控制信号和两个源操作数
wire [31: 0] EX_PC;
wire [31: 0] EX_RD1;
wire [31: 0] EX_RD2;
wire [31: 0] EX_IMM32;
wire [4: 0] EX_rs;
wire [4: 0] EX_rt;
wire [4: 0] EX_RF_rd;
wire [4: 0] EX_shamt;
wire[1: 0] EX_RegDst;
wire EX_MemRead;
wire[1: 0] EX_MemtoReg;
wire[4: 0] EX_ALUop;
```

```
wire EX_MemWrite;
wire EX_ALUSrc;
wire EX_RegWrite;
wire EX_Shamt;
wire[2: 0] EX_PCSrc;
wire[3: 0] EX_MemControl;

wire [31: 0] EX_RD1_forward;
wire [31: 0] EX_RD2_forward;
wire [31: 0] EX_ALU_Result;
wire EX_zero;

wire [1: 0] ForwardA;
wire [1: 0] ForwardB;
wire [1: 0] ForwardC;
wire [1: 0] ForwardD;
wire [31: 0] EX_ALU1_forward;
wire [31: 0] EX_ALU2_forward;

//MEM 阶段
//从 EX/MEM 寄存器中读取出来的控制信号,ALU 运算结果和第二个源操作数
wire [31: 0] MEM_PC;
wire [31: 0] MEM_ALU_result;
wire [31: 0] MEM_RD2;
wire [4: 0] MEM_RF_rd;
wire MEM_MemRead;
wire[1: 0] MEM_MemtoReg;
wire MEM_MemWrite;
wire MEM_RegWrite;
wire[2: 0] MEM_PCSrc;
wire[3: 0] MEM_MemControl;

wire [31: 0] MEM_Mem_data;
```

```
//WB 阶段
//从 MEM/WB 寄存器中读取出来的控制信号,ALU 运算结果和从 MEM 中读取出来
的数据
wire [31: 0] WB_PC;
wire [31: 0] WB_ALU_result;
wire [31: 0] WB_Mem_data;
wire [4: 0] WB_RF_rd;
wire [1: 0] WB_MemtoReg;
wire WB_RegWrite;
wire [2: 0] WB_PCSrc;

wire [31: 0] WB_RF_wd;

//拆分指令
assign ID_Op = ID_Instr[31: 26];
assign ID_Funct = ID_Instr[5: 0];
assign ID_rs = ID_Instr[25: 21];
assign ID_rt = ID_Instr[20: 16];
assign ID_rd = ID_Instr[15: 11];
assign ID_Imm16 = ID_Instr[15: 0];
assign ID_IMM = ID_Instr[25: 0];
assign ID_shamt = ID_Instr[10: 6];

assign ID_zero = (ID_RD1_forward == ID_RD2_forward);

NPC npc(
      .PC(IF_PC),
      .ID_PC(ID_PC),
      .NPCop(ID_PCSrc),
      .Zero(ID_zero),
      .IMM(ID_IMM),
      .Imm16(ID_Imm16),
      .rs(ID_RD1_forward),
      .harzard(PC_unchanged),
```

```
      .NPC(IF_NPC)
    );

PC pc(
     .clk(clk),
     .rst(rst),
     .NPC(IF_NPC),
     .PC(IF_PC)
   );

IM im(
     .addr(IF_PC),
     .instr(IF_Instr)
   );

IF_ID if_id(
        .clk(clk),
        .rst(rst),
        .PC(IF_PC),
        .Instr(IF_Instr),
        .harzard(IF_ID_flush),
        .PC_out(ID_PC),
        .Instr_out(ID_Instr)
      );

CONTROL control(
          .Op(ID_Op),
          .Func(ID_Funct),
          .RegDst(ID_RegDst),
          .Jump(ID_Jump),
          .Branch(ID_Branch),
          .MemRead(ID_MemRead),
          .MemtoReg(ID_MemtoReg),
          .ALUop(ID_ALUop),
```

```
          .MemWrite(ID_MemWrite),
          .ALUSrc(ID_ALUSrc),
          .RegWrite(ID_RegWrite),
          .Shamt(ID_Shamt),
          .Ext(ID_Ext),
          .PCSrc(ID_PCSrc),
          .MemControl(ID_MemControl)
        );

EXT ext(
      .Imm16(ID_Imm16),
      .EXTOp(ID_Ext),
      .Imm32(ID_Imm32)
    );

RF rf(
     .clk(clk),
     .rst(rst),
     .RFWr(WB_RegWrite),
     .A1(ID_rs),
     .A2(ID_rt),
     .A3(WB_RF_rd),
     .WD(WB_RF_wd),
     .RD1(ID_RD1),
     .RD2(ID_RD2)
   );

MUX4 #(32) mux4_RD1_forward(
       .d0(ID_RD1),
       .d1(EX_ALU_Result),
       .d2(MEM_ALU_result),
       .d3(WB_RF_wd),
       .select(ForwardC),
       .dout(ID_RD1_forward)
```

```
    );

MUX4 #(32) mux4_RD2_forward(
      .d0(ID_RD2),
      .d1(EX_ALU_Result),
      .d2(MEM_ALU_result),
      .d3(WB_RF_wd),
      .select(ForwardD),
      .dout(ID_RD2_forward)
    );

HARZARD hardzard(
        .ID_rs(ID_rs),
        .ID_rt(ID_rt),
        .EX_rt(EX_rt),
        .ID_PCSrc(ID_PCSrc),
        .EX_MemRead(EX_MemRead),
        .IF_ID_flush(IF_ID_flush),
        .PC_unchanged(PC_unchanged)
       );

MUX4 #(5) mux4_RFRD(
      .d0(ID_rt),
      .d1(ID_rd),
      .d2(5'd31),
      .d3(5'd0),
      .select(ID_RegDst),
      .dout(ID_RF_rd)
    );

ID_EX id_ex(
       .clk(clk),
       .rst(rst),
       .PC(ID_PC),
```

```
        .RD1(ID_RD1),
        .RD2(ID_RD2),
        .IMM32(ID_Imm32),
        .rs(ID_rs),
        .rt(ID_rt),
        .RF_rd(ID_RF_rd),
        .shamt(ID_shamt),
        .MemRead(ID_MemRead),
        .MemtoReg(ID_MemtoReg),
        .ALUop(ID_ALUop),
        .MemWrite(ID_MemWrite),
        .ALUSrc(ID_ALUSrc),
        .RegWrite(ID_RegWrite),
        .Shamt(ID_Shamt),
        .PCSrc(ID_PCSrc),
        .MemControl(ID_MemControl),
        .PC_out(EX_PC),
        .RD1_out(EX_RD1),
        .RD2_out(EX_RD2),
        .IMM32_out(EX_IMM32),
        .rs_out(EX_rs),
        .rt_out(EX_rt),
        .RF_rd_out(EX_RF_rd),
        .shamt_out(EX_shamt),
        .MemRead_out(EX_MemRead),
        .MemtoReg_out(EX_MemtoReg),
        .ALUop_out(EX_ALUop),
        .MemWrite_out(EX_MemWrite),
        .ALUSrc_out(EX_ALUSrc),
        .RegWrite_out(EX_RegWrite),
        .Shamt_out(EX_Shamt),
        .PCSrc_out(EX_PCSrc),
        .MemControl_out(EX_MemControl)
    );
```

```
MUX2 #(32) mux2_ALUA(
       .d0(EX_RD1_forward),
       .d1({27'd0, EX_shamt[4: 0]}),
       .select(EX_Shamt),
       .dout(EX_ALU1_forward)
     );

MUX2 #(32) mux2_ALUB(
       .d0(EX_RD2_forward),
       .d1(EX_IMM32),
       .select(EX_ALUSrc),
       .dout(EX_ALU2_forward)
     );

FORWARD forward(
          .ID_rs(ID_rs),
          .ID_rt(ID_rt),
          .EX_rs(EX_rs),
          .EX_rt(EX_rt),
          .EX_RF_rd(EX_RF_rd),
          .MEM_RF_rd(MEM_RF_rd),
          .WB_RF_rd(WB_RF_rd),
          .EX_RegWrite(EX_RegWrite),
          .MEM_RegWrite(MEM_RegWrite),
          .WB_RegWrite(WB_RegWrite),
          .MEM_MemRead(MEM_MemRead),
          .ForwardA(ForwardA),
          .ForwardB(ForwardB),
          .ForwardC(ForwardC),
          .ForwardD(ForwardD)
        );

MUX4 #(32) mux4_ALUA_forward(
```

```
        .d0(EX_RD1),
        .d1(MEM_ALU_result),
        .d2(WB_RF_wd),
        .d3(MEM_Mem_data),
        .select(ForwardA),
        .dout(EX_RD1_forward)
      );

MUX4 #(32) mux4_ALUB_forward(
        .d0(EX_RD2),
        .d1(MEM_ALU_result),
        .d2(WB_RF_wd),
        .d3(MEM_Mem_data),
        .select(ForwardB),
        .dout(EX_RD2_forward)
      );

ALU alu(
       .A(EX_ALU1_forward),
       .B(EX_ALU2_forward),
       .ALUOp(EX_ALUop),
       .C(EX_ALU_Result),
       .Zero(EX_zero)
     );

EX_MEM ex_mem(
          .clk(clk),
          .rst(rst),
          .PC(EX_PC),
          .ALU_result(EX_ALU_Result),
          .RD2(EX_RD2_forward),
          .RF_rd(EX_RF_rd),
          .MemRead(EX_MemRead),
          .MemtoReg(EX_MemtoReg),
```

```
        .MemWrite(EX_MemWrite),
        .RegWrite(EX_RegWrite),
        .PCSrc(EX_PCSrc),
        .MemControl(EX_MemControl),
        .PC_out(MEM_PC),
        .ALU_result_out(MEM_ALU_result),
        .RD2_out(MEM_RD2),
        .RF_rd_out(MEM_RF_rd),
        .MemRead_out(MEM_MemRead),
        .MemtoReg_out(MEM_MemtoReg),
        .MemWrite_out(MEM_MemWrite),
        .RegWrite_out(MEM_RegWrite),
        .PCSrc_out(MEM_PCSrc),
        .MemControl_out(MEM_MemControl)
      );

MEM mem(
     .clk(clk),
     .rst(rst),
     .input_address(MEM_ALU_result),
     .input_data(MEM_RD2),
     .Wr(MEM_MemWrite),
     .MemControl(MEM_MemControl),
     .output_data(MEM_Mem_data)
   );

MEM_WB mem_wb(
        .clk(clk),
        .rst(rst),
        .PC(MEM_PC),
        .ALU_result(MEM_ALU_result),
        .Mem_data(MEM_Mem_data),
        .RF_rd(MEM_RF_rd),
        .MemtoReg(MEM_MemtoReg),
```

```
        .RegWrite(MEM_RegWrite),
        .PCSrc(MEM_PCSrc),
        .PC_out(WB_PC),
        .ALU_result_out(WB_ALU_result),
        .Mem_data_out(WB_Mem_data),
        .RF_rd_out(WB_RF_rd),
        .MemtoReg_out(WB_MemtoReg),
        .RegWrite_out(WB_RegWrite),
        .PCSrc_out(WB_PCSrc)
      );

MUX4 #(32) mux4_RFWD(
      .d0(WB_ALU_result),
      .d1(WB_Mem_data),
      .d2((WB_PC + 4)),
      .d3(32'd0),
      .select(WB_MemtoReg),
      .dout(WB_RF_wd)
    );



endmodule
```

描述说明： 变量命名上尽量遵循以该变量出现的阶段+下划线+变量的意义的命名方式，和冒险与旁路相关的变量不遵循上述方式，单独命名。

7.2 ENCODE（预定义单元）

实现代码：

```
//instrution operation code
`define R_OP        6'b000000

`define LB_OP         6'b100000
```

```
`define LH_OP          6'b100001
`define LBU_OP         6'b100100
`define LHU_OP         6'b100101
`define LW_OP          6'b100011

`define SB_OP          6'b101000
`define SH_OP          6'b101001
`define SW_OP          6'b101011

`define ADDI_OP        6'b001000
`define ADDIU_OP       6'b001001
`define ANDI_OP        6'b001100
`define ORI_OP         6'b001101
`define XORI_OP        6'b001110
`define LUI_OP         6'b001111
`define SLTI_OP        6'b001010
`define SLTIU_OP       6'b001011

`define BEQ_OP         6'b000100
`define BNE_OP         6'b000101
`define BGEZ_OP        6'b000001
`define BGTZ_OP        6'b000111
`define BLEZ_OP        6'b000110
`define BLTZ_OP        6'b000001

`define J_OP           6'b000010
`define JAL_OP         6'b000011

//R type Instruction Funct
`define ADD_FUNCT      6'b100000
`define ADDU_FUNCT     6'b100001
`define SUB_FUNCT      6'b100010
`define SUBU_FUNCT     6'b100011
`define AND_FUNCT      6'b100100
```

```
`define NOR_FUNCT     6'b100111
`define OR_FUNCT      6'b100101
`define XOR_FUNCT     6'b100110
`define SLT_FUNCT     6'b101010
`define SLTU_FUNCT    6'b101011
`define SLL_FUNCT     6'b000000
`define SRL_FUNCT     6'b000010
`define SRA_FUNCT     6'b000011
`define SLLV_FUNCT    6'b000100
`define SRLV_FUNCT    6'b000110
`define SRAV_FUNCT    6'b000111
`define JR_FUNCT      6'b001000
`define JALR_FUNCT    6'b001001

//ALU control signal
`define ALU_NOP   5'b00000
`define ALU_ADDU  5'b00001
`define ALU_ADD   5'b00010
`define ALU_SUBU  5'b00011
`define ALU_SUB   5'b00100
`define ALU_AND   5'b00101
`define ALU_OR    5'b00110
`define ALU_NOR   5'b00111
`define ALU_XOR   5'b01000
`define ALU_SLT   5'b01001
`define ALU_SLTU  5'b01010
`define ALU_EQL   5'b01011
`define ALU_BNE   5'b01100
`define ALU_GT0   5'b01101
`define ALU_GE0   5'b01110
`define ALU_LT0   5'b01111
`define ALU_LE0   5'b10000
`define ALU_SLL   5'b10001
`define ALU_SRL   5'b10010
```

```
`define ALU_SRA   5'b10011
`define ALU_SLLV  5'b10100
`define ALU_SRLV  5'b10101
`define ALU_SRAV  5'b10110

//NPC control signal
`define NPC_PLUS4   3'b000
`define NPC_BRANCH  3'b001
`define NPC_JUMP    3'b010
`define NPC_JAL     3'b011
`define NPC_JALR    3'b100
`define NPC_BNE     3'b101
`define NPC_JR      3'b110

//Extend control signal
`define EXT_ZERO    2'b01
`define EXT_SIGNED  2'b00
`define EXT_HIGHPOS 2'b10

//MEM control signal
`define MEM_NOP     4'b0000
`define MEM_LW      4'b0001
`define MEM_SW      4'b0010
`define MEM_LB      4'b0011
`define MEM_LH      4'b0100
`define MEM_LBU     4'b0101
`define MEM_LHU     4'b0110
`define MEM_SB      4'b0111
`define MEM_SH      4'b1000
```

描述说明：该文件实现了对指令的功能码和操作码以及一些取值范围较大的控制信号的预定义宏，这样做的好处是能提高代码的可读性，同时尽量减少和避免弄混淆控制信号的值与控制信号意义的对应。该部件完全复用自单周期 CPU 中的部件，功能完全一样。

7.3 PC（程序计数器）

实现代码：

```
module PC( clk, rst, NPC, PC );

input clk;
input rst;
input [31: 0] NPC;
output reg [31: 0] PC;

always @(posedge clk, posedge rst)
  if (rst)
    PC <= 32'h0000_0000;
//      PC <= 32'h0000_3000
//      上一行表示程序段从内存中的0000_3000 作为起始的情况。
  else
    PC <= NPC;

endmodule
```

描述说明：通过重置信号有效时对 PC 赋不同的值，可以模拟程序从内存中的 0 作为起始存储地址和 3000 作为起始存储地址的不同情况。该部件完全复用自单周期 CPU 中的部件，功能完全一样。

7.4 NPC（程序计数器更新单元）

实现代码：

```
`include "ENCODE.v"

module NPC(PC, NPCop, Zero, IMM, Imm16, rs, NPC);  // next pc
module

input [31: 0] PC;        // pc
input [2: 0] NPCop;     // next pc operation
input Zero;
input [25: 0] IMM;       // immediate
```

```
input [15: 0] Imm16;
input [31: 0] rs;
output reg [31: 0] NPC;   // next pc

wire [31: 0] PCPLUS4;

assign PCPLUS4 = PC + 4; // pc + 4

always @( * )
    begin
      case (NPCop)
      `NPC_PLUS4:
        NPC = PCPLUS4;
      `NPC_BRANCH:
          begin
          if (Zero === 1'b1)
            NPC = PCPLUS4 + {{14{Imm16[15]}}, Imm16[15: 0], 2'
b00};
          else
            NPC = PCPLUS4;
          end
      `NPC_BNE:
          begin
          if (Zero != 1'b1)
            NPC = PCPLUS4 + {{14{Imm16[15]}}, Imm16[15: 0], 2'
b00};
          else
            NPC = PCPLUS4;
          end
      `NPC_JUMP:
        NPC = {PCPLUS4[31: 28], IMM[25: 0], 2'b00};
      `NPC_JAL:
        NPC = {PCPLUS4[31: 28], IMM[25: 0], 2'b00};
      `NPC_JALR:
```

```
        NPC = rs;
      `NPC_JR:
        NPC = rs;
      default:
        NPC = PCPLUS4;
      endcase
    end // end always

endmodule
```

描述说明：该部件的主要设计思想把所有和更新地址相关的控制信号和立即数等数据都送入该部件，然后在控制信号的控制下进行地址的更新。该部件部分复用自单周期 CPU，主要改动在于同时送入了 IF 级的 PC 和 ID 级的 PC，这样做的主要思想是缩短决定分支的时间，将决定分支提早到 ID 级，这样对于所有的分支和跳转指令都最多只需要丢弃一条指令即可解决冒险的问题。为了实现分支的提前，故需要把 ID 级的 PC 和冒险相关的控制信号送入到该部件中。

7.5 IM（程序存储器）

实现代码：

```
module IM(addr, instr);

input [31: 0] addr;
output reg[31: 0] instr;

reg[31: 0] Instrs[1023: 0];

initial
    begin
    //$readmemh("../test_codes/extendedtest.dat", Instrs, 0);
    //$readmemh("../test_codes/extendedtest.dat",Instrs,3000);
    //$readmemh("../test_codes/mipstest_extloop.dat",Instrs,0)
;
    //$readmemh("../test_codes/mipstest_extloop.dat",Instrs,30
00);
```

```
    //$readmemh("../test_codes/mipstestloop_sim.dat",Instrs,0)
;
    //$readmemh("../test_codes/mipstestloop_sim.dat",Instrs,30
00);
    //$readmemh("../test_codes/mipstestloopjal_sim.dat", Instr
s, 0);
    //$readmemh("../test_codes/mipstestloopjal_sim.dat",Instrs
,3000);
    $readmemh("../test_codes/mipstest_pipelinedloop.dat", Inst
rs, 0);
    //$readmemh("../test_codes/mipstest_pipelinedloop.dat",Ins
trs,3000);

    end

always@(addr)
    begin
    instr = Instrs[addr >> 2];
    //$display("Instrs[%4d]=0x%8h",pc,Instrs[pc]);
    end

endmodule
```

描述说明：和 PC 部件的设计类似，通过 initial 语句块中不同的语句可以以不同的起始地址将程序读取到指令存储器中，但该起始地址必须和 PC 中的起始地址匹配。同时将字节地址偏移成字地址的工作也是在该部件中完成。该部件完全复用自单周期 CPU 中的部件，功能完全一样。

7.6 RF（寄存器组）

实现代码：

```
module RF( input clk,
           input rst,
           input RFWr,
           input [4: 0] A1, A2, A3,
```

```
            input [31: 0] WD,
            output [31: 0] RD1, RD2
          );
//clk 是时钟信号，WD 是写会寄存器的数据，reset 是清零信号,RFWr 是写使能信号，A3 是写入的寄存器号，reg_sel 和reg_data 是作为调试信号，主要是用于读出某一个寄存器中的值

reg [31: 0] rf[31: 0];

integer i;

always @(negedge clk or posedge rst)
  if (rst)
      begin    //  reset
      for (i = 0; i < 32; i = i + 1)
        rf[i] <= 0;
      end

  else
    if (RFWr)
        begin
        rf[A3] <= WD;
        end


assign RD1 = (A1 != 0) ? rf[A1] : 0;
assign RD2 = (A2 != 0) ? rf[A2] : 0;

endmodule
```

描述说明：该部件部分复用自单周期 CPU，但为了解决结构冒险，将写入寄存器组的时机选择在了时钟的下降沿，以实现前半周期写，后半周期读的功能。

7.7 CONTROL（控制单元）

实现代码：

```
`include "ENCODE.v"
module CONTROL(Op, Func, RegDst, Jump, Branch, MemRead, MemtoR
eg, ALUop, MemWrite, ALUSrc, RegWrite, Shamt, Ext, PCSrc, MemC
ontrol);
//理论上来说 PCSrc 是来间接信号，但在现有设计中 PCSrc 由 CONTROL 产生

input [5: 0] Op;
input [5: 0] Func;

output reg[1: 0] RegDst; //写寄存器时的目标寄存器，为 01 时写 rd,为
00 时写 rt,为 10 时写第 31 号寄存器
output reg Jump; //跳转指令
output reg Branch; //分支指令
output reg MemRead; //读存储器
output reg[1: 0] MemtoReg; //为 01 时写入的数据来自数据存储器,为 00
时来自 ALU 计算的结果,为 10 时来自 PC+4
output reg[4: 0] ALUop;
output reg MemWrite; //为 1 时数据存储器写使能有效
output reg ALUSrc; //为 1 时 ALU 的第二个操作数来自符号扩展，为 0 时
来自 rt 默认为 0
output reg RegWrite; //为 1 时写入 register 有效
output reg Shamt; //为 1 时 ALU 的第一个操作数来自 shamt 字段
output reg[1: 0] Ext; //决定符号扩展的类型
output reg[2: 0] PCSrc;
output reg[3: 0] MemControl;

always@( * )
    begin
    RegDst <= 2'b01;
    Jump <= 1'b0;
    Branch <= 1'b0;
    MemRead <= 1'b0;
    MemtoReg <= 2'b00;
    ALUop <= `ALU_NOP;
```

```
    MemWrite <= 1'b0;
    ALUSrc <= 1'b0;
    RegWrite <= 1'b0;
    Shamt <= 1'b0;
    Ext <= `EXT_ZERO;
    PCSrc <= `NPC_PLUS4;
    MemControl <= `MEM_NOP;

      case (Op)
      `R_OP:
          begin
          RegDst <= 2'b01;
          RegWrite <= 1'b1;
            case (Func)
            `ADD_FUNCT:
                begin
                ALUop <= `ALU_ADD;
                end
            `ADDU_FUNCT:
                begin
                ALUop <= `ALU_ADDU;
                end
            `AND_FUNCT:
                begin
                ALUop <= `ALU_AND;
                end
            `SUB_FUNCT:
                begin
                ALUop <= `ALU_SUB;
                end
            `SUBU_FUNCT:
                begin
                ALUop <= `ALU_SUBU;
                end
```

```
            `NOR_FUNCT:
                begin
                ALUop <= `ALU_NOR;
                end
            `OR_FUNCT:
                begin
                ALUop <= `ALU_OR;
                end
            `XOR_FUNCT:
                begin
                ALUop <= `ALU_XOR;
                end
            `SLT_FUNCT:
                begin
                ALUop <= `ALU_SLT;
                end
            `SLTU_FUNCT:
                begin
                ALUop <= `ALU_SLTU;
                end
            `SLL_FUNCT:
                begin
                ALUop <= `ALU_SLL;
                Shamt <= 1'b1;
                end
            `SRL_FUNCT:
                begin
                ALUop <= `ALU_SRL;
                Shamt <= 1'b1;
                end
            `SRA_FUNCT:
                begin
                ALUop <= `ALU_SRA;
                Shamt <= 1'b1;
```

```
            end
        `SLLV_FUNCT:
            begin
            ALUop <= `ALU_SLLV;
            end
        `SRLV_FUNCT:
            begin
            ALUop <= `ALU_SRLV;
            end
        `SRAV_FUNCT:
            begin
            ALUop <= `ALU_SRAV;
            end
        `JALR_FUNCT:
            begin
            MemtoReg <= 2'b10;
            PCSrc <= `NPC_JALR;
            end
        `JR_FUNCT:
            begin
            PCSrc <= `NPC_JR;
            RegWrite <= 1'b0;
            end
        endcase
    end
`ADDI_OP:
    begin
    RegDst <= 2'b00;
    ALUop <= `ALU_ADD;
    ALUSrc <= 1;
    RegWrite <= 1'b1;
    Ext <= `EXT_SIGNED;
    end
`ORI_OP:
```

```
            begin
            RegDst <= 2'b00;
            ALUop <= `ALU_OR;
            ALUSrc <= 1;
            RegWrite <= 1'b1;
            Ext <= `EXT_SIGNED;
            end
        `ANDI_OP:
            begin
            RegDst <= 2'b00;
            ALUop <= `ALU_AND;
            ALUSrc <= 1;
            RegWrite <= 1'b1;
            Ext <= `EXT_SIGNED;
            end
        `LUI_OP:
            begin
            RegDst <= 2'b00;
            ALUop <= `ALU_ADD;
            ALUSrc <= 1'b1;
            RegWrite <= 1'b1;
            Ext <= `EXT_HIGHPOS;
            end
        `SLTI_OP:
            begin
            RegDst <= 2'b00;
            ALUop <= `ALU_SLT;
            ALUSrc <= 1'b1;
            RegWrite <= 1'b1;
            Ext <= `EXT_SIGNED;
            end
        `BEQ_OP:
            begin
            PCSrc <= `NPC_BRANCH;
```

```
            end
        `BNE_OP:
            begin
            PCSrc <= `NPC_BNE;
            end
        `SW_OP:
            begin
            ALUop <= `ALU_ADD;
            MemWrite <= 1'b1;
            ALUSrc <= 1'b1;
            Ext <= `EXT_SIGNED;
            MemControl <= `MEM_SW;
            end
        `SB_OP:
            begin
            ALUop <= `ALU_ADD;
            MemWrite <= 1'b1;
            ALUSrc <= 1'b1;
            Ext <= `EXT_SIGNED;
            MemControl <= `MEM_SB;
            end
        `SH_OP:
            begin
            ALUop <= `ALU_ADD;
            MemWrite <= 1'b1;
            ALUSrc <= 1'b1;
            Ext <= `EXT_SIGNED;
            MemControl <= `MEM_SH;
            end
        `LW_OP:
            begin
            RegDst <= 2'b00;
            MemRead <= 1'b1;
            MemtoReg <= 2'b01;
```

```
            ALUop <= `ALU_ADD;
            ALUSrc <= 1'b1;
            RegWrite <= 1'b1;
            Ext <= `EXT_SIGNED;
            MemControl <= `MEM_LW;
            end
        `LB_OP:
            begin
            RegDst <= 2'b00;
            MemRead <= 1'b1;
            MemtoReg <= 2'b01;
            ALUop <= `ALU_ADD;
            ALUSrc <= 1'b1;
            RegWrite <= 1'b1;
            Ext <= `EXT_SIGNED;
            MemControl <= `MEM_LB;
            end
        `LH_OP:
            begin
            RegDst <= 2'b00;
            MemRead <= 1'b1;
            MemtoReg <= 2'b01;
            ALUop <= `ALU_ADD;
            ALUSrc <= 1'b1;
            RegWrite <= 1'b1;
            Ext <= `EXT_SIGNED;
            MemControl <= `MEM_LH;
            end
        `LBU_OP:
            begin
            RegDst <= 2'b00;
            MemRead <= 1'b1;
            MemtoReg <= 2'b01;
            ALUop <= `ALU_ADD;
```

```
            ALUSrc <= 1'b1;
            RegWrite <= 1'b1;
            Ext <= `EXT_SIGNED;
            MemControl <= `MEM_LBU;
            end
        `LHU_OP:
            begin
            RegDst <= 2'b00;
            MemRead <= 1'b1;
            MemtoReg <= 2'b01;
            ALUop <= `ALU_ADD;
            ALUSrc <= 1'b1;
            RegWrite <= 1'b1;
            Ext <= `EXT_SIGNED;
            MemControl <= `MEM_LHU;
            end
        `J_OP:
            begin
            PCSrc <= `NPC_JUMP;
            end
        `JAL_OP:
            begin
            RegDst <= 2'b10;
            MemtoReg <= 2'b10;
            RegWrite <= 1'b1;
            PCSrc <= `NPC_JAL;
            end

        endcase
    end

endmodule
```

描述说明：

该部件主要的实现思路是在一个部件中生成所有的控制信号，而不像教材中

对于 ALUOp 这样采取的多级译码的方式。同时在部件的具体实现上采用了较为抽象和语义明确的 case 语句，从而避免了复杂而庞大的卡诺图和逻辑函数。该部件完全复用自单周期 CPU 中的部件，功能完全一样。

7.8 ALU（算术运算单元）

实现代码：

```
`include "ENCODE.v"

module ALU(A, B, ALUOp, C, Zero);

input signed [31: 0] A, B;
input [4: 0] ALUOp;
output signed [31: 0] C;
output Zero;
//ALU 的输入和输出均为有符号数，在 ALU 内部运算的时候会根据情况转换成无符号数

reg [31: 0] C;
integer i;


always @( A or B or ALUOp )
    begin
      case ( ALUOp )
      `ALU_NOP:
        C = A;
      `ALU_ADDU:
        C = $unsigned(A) + $unsigned(B);
      `ALU_ADD:
        C = A + B;
      `ALU_SUBU:
        C = $unsigned(A) - $unsigned(B);
      `ALU_SUB:
        C = A - B;
```

```
      `ALU_AND:
        C = A & B;
      `ALU_OR:
        C = A | B;
      `ALU_NOR:
        C = ~(A | B);
      `ALU_XOR:
        C = (A ^ B);
      `ALU_SLT:
        C = A < B ? 1 : 0;
      `ALU_SLTU:
        C = $unsigned(A) < $unsigned(B) ? 1 : 0;
      `ALU_SLL:
        C = B << A;
      `ALU_SLLV:
        C = B << A[4: 0];
      `ALU_SRL:
        C = B >> A;
      `ALU_SRLV:
        C = B >> A[4: 0];
      `ALU_SRA:
        C = B >>> A;
      `ALU_SRAV:
        C = B >>> A[4: 0];
      //其中>>>是verilog原生运算符，表示算术右移

      default:
        C = A;                          // Undefined
      endcase
    end

assign Zero = (C == 32'b0);

endmodule
```

描述说明：ALU 运算单元的设计尽量采用 verilog 语法中的原生运算符，最为典型的例子是“>>>”,算术右移运算符。这样做的好处是能尽量简化代码减少 bug，将硬件较为底层的实现交由 verilog 编译器来完成，从而保证了 ALU 单元设计的可靠性。该部件完全复用自单周期 CPU 中的部件，功能完全一样。

7.9 MEM（数据存储器）

实现代码：

```
`include "ENCODE.v"

module MEM(clk, rst, input_address, input_data, Wr, MemControl
, output_data);

input clk;
input rst;
input[31: 0] input_address;
input[31: 0] input_data;
input Wr;
input[3: 0] MemControl;
output reg[31: 0] output_data;

reg [9: 0] word_addr;
reg [1: 0] byte_addr;
reg [31: 0] word;
reg [31: 0] data_memory[1023: 0];
integer i;

always@( * )
    begin
    word_addr = input_address[11: 2];
    byte_addr = input_address[1: 0];
    word = data_memory[word_addr[9: 0]];
      case (MemControl)
      `MEM_LW:
          begin
```

```
        output_data = data_memory[word_addr[9: 0]];
        end
    `MEM_LB:
        begin
          case (byte_addr)
          2'b00:
              begin
              output_data = {{24{word[7]}}, word[7: 0]};
              end
          2'b01:
              begin
              output_data = {{24{word[15]}}, word[15: 8]};
              end
          2'b10:
              begin
              output_data = {{24{word[23]}}, word[23: 16]};
              end
          2'b11:
              begin
              output_data = {{24{word[31]}}, word[31: 24]};
              end
          endcase
        end
    `MEM_LH:
        begin
          case (byte_addr)
          2'b00:
              begin
              output_data = {{16{word[15]}}, word[15: 0]};
              end
          2'b10:
              begin
              output_data = {{16{word[31]}}, word[31: 16]};
              end
```

```
            endcase
        end
    `MEM_LBU:
        begin
            case (byte_addr)
            2'b00:
                begin
                output_data = {{24{1'b0}}, word[7: 0]};
                end
            2'b01:
                begin
                output_data = {{24{1'b0}}, word[15: 8]};
                end
            2'b10:
                begin
                output_data = {{24{1'b0}}, word[23: 16]};
                end
            2'b11:
                begin
                output_data = {{24{1'b0}}, word[31: 24]};
                end
            endcase
        end
    `MEM_LHU:
        begin
            case (byte_addr)
            2'b00:
                begin
                output_data = {{16{1'b0}}, word[15: 0]};
                end
            2'b10:
                begin
                output_data = {{16{1'b0}}, word[31: 16]};
                end
```

```
            endcase
          end
      endcase
    end

always @(negedge clk or posedge rst)
    begin
    word_addr = input_address[11: 2];
    byte_addr = input_address[1: 0];
    word = data_memory[word_addr[9: 0]];
    if (rst)
        begin
        for (i = 0;i < 1024;i = i + 1)
          data_memory[i] <= 32'h0000_0000;
        end
    if (Wr == 1'b1)
        begin
          case (MemControl)
          `MEM_SW:
              begin
              data_memory[word_addr[9: 0]] <= input_data[31: 0
];
              end
          `MEM_SH:
              begin
                case (byte_addr)
                2'b00:
                    begin
                    data_memory[word_addr[9: 0]] = {word[31: 1
6], input_data[15: 0]};
                    end
                2'b10:
                    begin
```

```
                    data_memory[word_addr[9: 0]] = {input_data
[15: 0], word[15: 0]};
                    end
                endcase
            end
        `MEM_SB:
            begin
                case (byte_addr)
                2'b00:
                    begin
                    data_memory[word_addr[9: 0]] = {word[31: 8
], input_data[7: 0]};
                    end
                2'b01:
                    begin
                    data_memory[word_addr[9: 0]] = {word[31: 1
6], input_data[7: 0], word[7: 0]};
                    end
                2'b10:
                    begin
                    data_memory[word_addr[9: 0]] = {word[31: 2
4], input_data[7: 0], word[15: 0]};
                    end
                2'b11:
                    begin
                    data_memory[word_addr[9: 0]] = {input_data
[7: 0], word[23: 0]};
                    end
                endcase
            end
        endcase
    end
end
```

```
endmodule
```

描述说明：该部件部分复用自单周期 CPU，但为了解决结构冒险，将写入数据存储器的时机选择在了时钟的下降沿，以实现前半周期写，后半周期读的功能。

7.10 EXT（符号扩展单元）

实现代码：

```
`include "ENCODE.v"

module EXT( Imm16, EXTOp, Imm32 );

input [15: 0] Imm16;
input [1: 0] EXTOp;
output [31: 0] Imm32;

reg [31: 0] Imm32;

always@( * )
    begin
      case (EXTOp)
      `EXT_ZERO:
        Imm32 = {16'd0, Imm16};
      `EXT_SIGNED:
        Imm32 = {{16{Imm16[15]}}, Imm16};
      `EXT_HIGHPOS:
        Imm32 = {Imm16, 16'd0};
      default:
        ;
      endcase
    end
endmodule
```

描述说明：该部件较为简单，主要是实现了对不同符号扩展类型的支持。该部件完全复用自单周期 CPU 中的部件，功能完全一样。

7.11 IF_ID（IF_ID 流水线寄存器）

实现代码：

```
module IF_ID(clk, rst, PC, Instr, harzard, PC_out, Instr_out);

input clk;
input rst;
input[31: 0] PC;
input[31: 0] Instr;
input harzard;
output reg[31: 0] PC_out;
output reg[31: 0] Instr_out;

initial
    begin
    PC_out <= 0;
    Instr_out <= 0;
    end

always @(posedge rst)
    begin
    PC_out <= 0;
    Instr_out <= 0;
    end

always @(posedge clk)
  if (harzard)
      begin
      PC_out = 0;
      Instr_out = 0;
      end
  else
      begin
      PC_out = PC;
```

```
      Instr_out = Instr;
      end

endmodule
```

描述说明：该流水线寄存器中的内容较为简单，但需额外注意的是这个流水线寄存器是唯一需要收到冒险信号控制的寄存器，产生空指令阻塞流水线也是在这个流水线寄存器中完成的

7.12 ID_EX（ID_EX 流水线寄存器）

实现代码：

```
`include "ENCODE.v"

module ID_EX(clk, rst, PC, RD1, RD2, IMM32, rs, rt, RF_rd, sha
mt, MemRead, MemtoReg, ALUop, MemWrite, ALUSrc, RegWrite, Sham
t, PCSrc, MemControl, PC_out, RD1_out, RD2_out, IMM32_out, rs_
out, rt_out , RF_rd_out, shamt_out, MemRead_out, MemtoReg_out,
 ALUop_out, MemWrite_out, ALUSrc_out, RegWrite_out, Shamt_out,
 PCSrc_out, MemControl_out);

input clk;
input rst;
input [31: 0] PC;
input [31: 0] RD1;
input [31: 0] RD2;
input [31: 0] IMM32;
input [4: 0] rs;
input [4: 0] rt;
input [4: 0] RF_rd;
input [4: 0] shamt;
input MemRead;
input [1: 0] MemtoReg;
input [4: 0] ALUop;
```

```
input MemWrite;
input ALUSrc;
input RegWrite;
input Shamt;
input [2: 0] PCSrc;
input [3: 0] MemControl;

output reg[31: 0] PC_out;
output reg[31: 0] RD1_out;
output reg[31: 0] RD2_out;
output reg[31: 0] IMM32_out;
output reg[4: 0] rs_out;
output reg[4: 0] rt_out;
output reg[4: 0] RF_rd_out;
output reg[4: 0] shamt_out;
output reg MemRead_out;
output reg[1: 0] MemtoReg_out;
output reg[4: 0] ALUop_out;
output reg MemWrite_out;
output reg ALUSrc_out;
output reg RegWrite_out;
output reg Shamt_out;
output reg[2: 0] PCSrc_out;
output reg[3: 0] MemControl_out;

initial
    begin
    PC_out <= 0;
    RD1_out <= 0;
    RD2_out <= 0;
    IMM32_out <= 0;
    rs_out <= 0;
    rt_out <= 0;
    RF_rd_out <= 0;
```

```
    shamt_out <= 0;
    MemRead_out <= 1'b0;
    MemtoReg_out <= 2'b00;
    ALUop_out <= `ALU_NOP;
    MemWrite_out <= 1'b0;
    ALUSrc_out <= 1'b0;
    RegWrite_out <= 1'b0;
    Shamt_out <= 1'b0;
    PCSrc_out <= `NPC_PLUS4;
    MemControl_out <= `MEM_NOP;
    end

always @(posedge rst)
    begin
    PC_out <= 0;
    RD1_out <= 0;
    RD2_out <= 0;
    IMM32_out <= 0;
    rs_out <= 0;
    rt_out <= 0;
    RF_rd_out <= 0;
    shamt_out <= 0;
    MemRead_out <= 1'b0;
    MemtoReg_out <= 2'b00;
    ALUop_out <= `ALU_NOP;
    MemWrite_out <= 1'b0;
    ALUSrc_out <= 1'b0;
    RegWrite_out <= 1'b0;
    Shamt_out <= 1'b0;
    PCSrc_out <= `NPC_PLUS4;
    MemControl_out <= `MEM_NOP;
    end

always @(posedge clk)
```

```
    begin
    PC_out <= PC;
    RD1_out <= RD1;
    RD2_out <= RD2;
    IMM32_out <= IMM32;
    rs_out <= rs;
    rt_out <= rt;
    RF_rd_out <= RF_rd;
    shamt_out <= shamt;
    MemRead_out <= MemRead;
    MemtoReg_out <= MemtoReg;
    ALUop_out <= ALUop;
    MemWrite_out <= MemWrite;
    ALUSrc_out <= ALUSrc;
    RegWrite_out <= RegWrite;
    Shamt_out <= Shamt;
    PCSrc_out <= PCSrc;
    MemControl_out <= MemControl;
    end

endmodule
```

描述说明： 该流水线寄存器中的内容较为复杂，但已经是尽力简化过的结果。所有后面阶段不会用到的数据和信号都没有放到该寄存器中。

7.13 EX_MEM（EX_MEM 流水线寄存器）

实现代码：

```
`include "ENCODE.v"

module EX_MEM(clk, rst, PC, ALU_result, RD2, RF_rd, MemRead, M
emtoReg, MemWrite, RegWrite, PCSrc, MemControl, PC_out, ALU_re
sult_out, RD2_out, RF_rd_out, MemRead_out, MemtoReg_out, MemWr
ite_out, RegWrite_out, PCSrc_out, MemControl_out);
```

```
input clk;
input rst;
input [31: 0] PC;
input [31: 0] ALU_result;
input [31: 0] RD2;
input [4: 0] RF_rd;
input MemRead;
input [1: 0] MemtoReg;
input MemWrite;
input RegWrite;
input [2: 0] PCSrc;
input [3: 0] MemControl;

output reg[31: 0] PC_out;
output reg[31: 0] ALU_result_out;
output reg[31: 0] RD2_out;
output reg[4: 0] RF_rd_out;
output reg MemRead_out;
output reg[1: 0] MemtoReg_out;
output reg MemWrite_out;
output reg RegWrite_out;
output reg[2: 0] PCSrc_out;
output reg[3: 0] MemControl_out;

initial
    begin
    PC_out <= 0;
    ALU_result_out <= 0;
    RD2_out <= 0;
    RF_rd_out <= 0;
    MemRead_out <= 1'b0;
    MemtoReg_out <= 2'b00;
    MemWrite_out <= 1'b0;
```

```
    RegWrite_out <= 1'b0;
    PCSrc_out <= `NPC_PLUS4;
    MemControl_out <= `MEM_NOP;
    end

always @(posedge rst)
    begin
    PC_out <= 0;
    ALU_result_out <= 0;
    RD2_out <= 0;
    RF_rd_out <= 0;
    MemRead_out <= 1'b0;
    MemtoReg_out <= 2'b00;
    MemWrite_out <= 1'b0;
    RegWrite_out <= 1'b0;
    PCSrc_out <= `NPC_PLUS4;
    MemControl_out <= `MEM_NOP;
    end

always @(posedge clk)
    begin
    PC_out <= PC;
    ALU_result_out <= ALU_result;
    RD2_out <= RD2;
    RF_rd_out <= RF_rd;
    MemRead_out <= MemRead;
    MemtoReg_out <= MemtoReg;
    MemWrite_out <= MemWrite;
    RegWrite_out <= RegWrite;
    PCSrc_out <= PCSrc;
    MemControl_out <= MemControl;
    end

endmodule
```

描述说明：该寄存器存储来自 EX 阶段的数据和控制信号

7.14 MEM_WB（MEM_WB 流水线寄存器）

实现代码：

```
`include "ENCODE.v"

module MEM_WB(clk, rst, PC, ALU_result, Mem_data, RF_rd, Memto
Reg, RegWrite, PCSrc, PC_out, ALU_result_out, Mem_data_out, RF
_rd_out, MemtoReg_out, RegWrite_out, PCSrc_out);

input clk;
input rst;
input [31: 0] PC;
input [31: 0] ALU_result;
input [31: 0] Mem_data;
input [4: 0] RF_rd;
input [1: 0] MemtoReg;
input RegWrite;
input [2: 0] PCSrc;

output reg[31: 0] PC_out;
output reg[31: 0] ALU_result_out;
output reg[31: 0] Mem_data_out;
output reg[4: 0] RF_rd_out;
output reg[1: 0] MemtoReg_out;
output reg RegWrite_out;
output reg[2: 0] PCSrc_out;

initial
    begin
    PC_out <= 0;
    ALU_result_out <= 0;
    Mem_data_out <= 0;
```

```
    RF_rd_out <= 0;
    MemtoReg_out <= 2'b00;
    RegWrite_out <= 1'b0;
    PCSrc_out <= `NPC_PLUS4;
    end

always @(posedge rst)
    begin
    PC_out <= 0;
    ALU_result_out <= 0;
    Mem_data_out <= 0;
    RF_rd_out <= 0;
    MemtoReg_out <= 2'b00;
    RegWrite_out <= 1'b0;
    PCSrc_out <= `NPC_PLUS4;
    end

always @(posedge clk)
    begin
    PC_out <= PC;
    ALU_result_out <= ALU_result;
    Mem_data_out <= Mem_data;
    RF_rd_out <= RF_rd;
    MemtoReg_out <= MemtoReg;
    RegWrite_out <= RegWrite;
    PCSrc_out <= PCSrc;
    end

endmodule
```

描述说明：该寄存器存储来自 MEM 阶段的数据和控制信号。

7.15 HARZARD（冒险检测单元）

实现代码：

```
`include "ENCODE.v"

module HARZARD(ID_rs, ID_rt, EX_rt, ID_PCSrc, EX_MemRead, IF_I
D_flush, PC_unchanged);

input [4: 0] ID_rs;
input [4: 0] ID_rt;
input [4: 0] EX_rt;
input [2: 0] ID_PCSrc;
input EX_MemRead;

output reg IF_ID_flush;
output reg PC_unchanged;
initial
    begin
    IF_ID_flush <= 1'b0;
    PC_unchanged <= 1'b0;
    end
always @( * )
    begin
    if (EX_MemRead && ((EX_rt == ID_rt) || (EX_rt == ID_rs)))
        begin
        IF_ID_flush <= 1'b1;
        PC_unchanged <= 1'b1;
        end
    else if (ID_PCSrc != `NPC_PLUS4)
        begin
        IF_ID_flush <= 1'b1;
        PC_unchanged <= 1'b0;
        end
    else
```

```
        begin
        IF_ID_flush <= 1'b0;
        PC_unchanged <= 1'b0;
        end
    end

endmodule
```

描述说明：该冒险检测单元同时完成数据冒险和控制冒险的检测，其中数据冒险主要解决 sw 指令后紧跟 lw 指令的冒险，该冒险需要同时阻塞流水线一个时钟周期并保持 PC 的值不变，而解决控制冒险则只需要阻塞流水线一个时钟周期，PC 的值在阻塞流水线一个时钟周期后会被写入正确的值。

7.16 FORWARD（旁路单元）

实现代码：

```
module FORWARD(ID_rs, ID_rt, EX_rs, EX_rt, EX_RF_rd, MEM_RF_rd
, WB_RF_rd, EX_RegWrite, MEM_RegWrite, WB_RegWrite, MEM_MemRea
d, ForwardA, ForwardB, ForwardC, ForwardD);

input [4: 0] ID_rs;
input [4: 0] ID_rt;
input [4: 0] EX_rs;
input [4: 0] EX_rt;
input [4: 0] EX_RF_rd;
input [4: 0] MEM_RF_rd;
input [4: 0] WB_RF_rd;
input EX_RegWrite;
input MEM_RegWrite;
input WB_RegWrite;
input MEM_MemRead;
```

```
output reg [1: 0] ForwardA;
output reg [1: 0] ForwardB;
output reg [1: 0] ForwardC;
output reg [1: 0] ForwardD;
//ForwardA=00 时，第一个操作数来自寄存器堆，ForwardA=01 时，第一个操作数从上一个 ALU 运算结果旁路获得，ForwardA=10 时，第一个操作数从 WB 阶段数据存储器或者前面的 ALU 结果中旁路获得,ForwardA=11 时,第一个操作数从 MEM 阶段数据存储器中获得，其他的 Forward 信号同理

always@( * )
    begin
    if (MEM_RegWrite && (MEM_RF_rd != 0) && (MEM_RF_rd == EX_rs))
        begin
        if (MEM_MemRead)
          ForwardA <= 2'b11;
        else
          ForwardA <= 2'b01;
        end
    else if (WB_RegWrite && (WB_RF_rd != 0) && (WB_RF_rd == EX_rs))
      ForwardA <= 2'b10;
    else
      ForwardA <= 2'b00;

    if (MEM_RegWrite && (MEM_RF_rd != 0) && (MEM_RF_rd == EX_rt))
        begin
        if (MEM_MemRead)
          ForwardB <= 2'b11;
        else
          ForwardB <= 2'b01;
        end
```

```
    else if (WB_RegWrite && (WB_RF_rd != 0) && (WB_RF_rd == EX
_rt))
      ForwardB <= 2'b10;
    else
      ForwardB <= 2'b00;

    if (EX_RegWrite && (EX_RF_rd != 0) && (EX_RF_rd == ID_rs))
      ForwardC <= 2'b01;
    else if (MEM_RegWrite && (MEM_RF_rd != 0) && (MEM_RF_rd ==
 ID_rs))
      ForwardC <= 2'b10;
    else if (WB_RegWrite && (WB_RF_rd != 0) && (WB_RF_rd == ID
_rs))
      ForwardC <= 2'b11;
    else
      ForwardC <= 2'b00;

    if (EX_RegWrite && (EX_RF_rd != 0) && (EX_RF_rd == ID_rt))
      ForwardD <= 2'b01;
    else if (MEM_RegWrite && (MEM_RF_rd != 0) && (MEM_RF_rd ==
 ID_rt))
      ForwardD <= 2'b10;
    else if (WB_RegWrite && (WB_RF_rd != 0) && (WB_RF_rd == ID
_rt))
      ForwardD <= 2'b11;
    else
      ForwardD <= 2'b00;



    end
endmodule
```

描述说明：该旁路单元同时实现了 ID 阶段的旁路和 EX 阶段的旁路，其中

ForwardA 和 ForwardB 信号控制 EX 阶段的旁路，ForwardC 和 ForwardD 控制 ID 阶段的旁路。EX 阶段的旁路较为常见，ID 阶段的旁路主要是为了帮助将决定分支提前的 ID 阶段进行，由于分支指令需要读取寄存器中的数据，故在 ID 阶段也可能产生数据冒险，所以需要额外的旁路支持。

8 流水线测试及结果分析

8.1 仿真代码及分析

测试代码如图 8-1 所示：

```
# mipstest_pipelined.asm

### Make sure the following Settings :
# Settings -> Memory Configuration -> Compact, Text at address 0
# You could use it to test if there is data hazard and control hazard.
# If successful, it should write value 12 to address 80 and 84, and register $7 should be 12

#       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 = 0xFFFFFFFC      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

```

图 8-1

指令间的数据依赖主要包括第 11 和第 12 条指令存在写后读的数据依赖，第 15 与 16，第 16 与 17 条指令间存在写后读的数据依赖，第 26 与 27 条指令间存在写后读的数据依赖，第 37 与 38 条指令间存在写后读的数据依赖，第 39、40、41、42 条指令间依次存在写后读的数据依赖。

8.2 仿真测试结果

首先展示测试结果

其中在 Mars 中仿真结果如图 8-2 所示

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x00000007
$v1	3	0x0000000c
$a0	4	0x00000001
$a1	5	0x0000000b
$a2	6	0x00000017
$a3	7	0x0000000c
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x00001800
$sp	29	0x00003ffc
$fp	30	0x00000000
$ra	31	0x00000014
pc		0x00000098
hi		0x00000000
lo		0x00000000

图 8-2

其中在 modelsim 中的仿真结果如图 8-3 所示

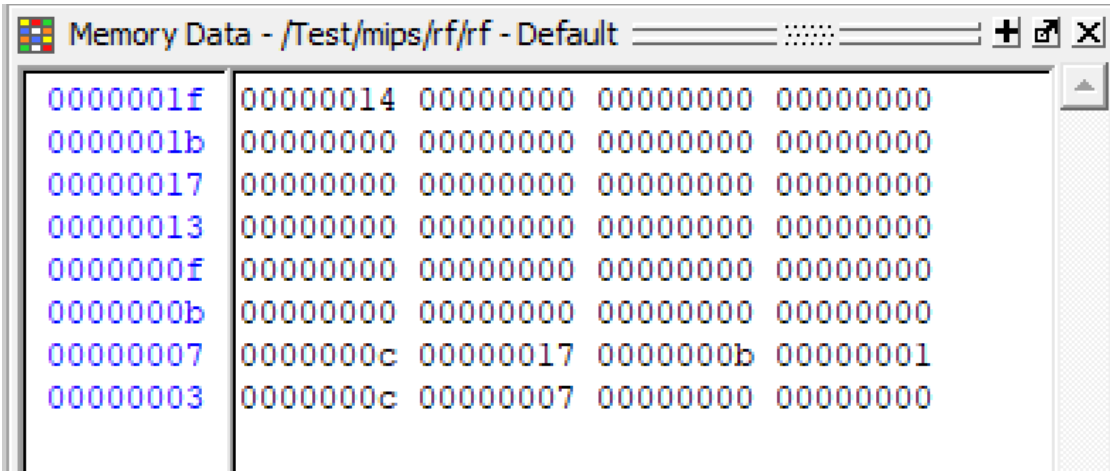

图 8-3

可以看到在 Mars 中和 modelsim 中仿真的结果一致，可以验证该 CPU 支持测试程序集上的指令。

下面将详细的分析数据冒险和控制冒险，首先分析数据冒险的情况，数据冒险可大致分为两种，一种是可通过旁路直接解决的数据冒险，一种是需要通过旁路和阻塞一个时钟周期才能解决的数据冒险，首先分析第一种。产生第一种数据冒险的指令如图 8-4 所示：

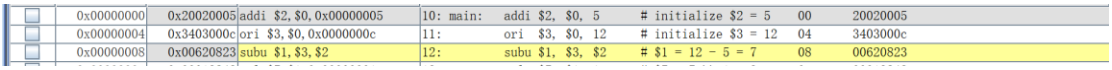

图 8-4

可以看到 subu 指令的第一个寄存器号和 addi 指令的目的寄存器号相同，同时第二个寄存器号和上一条指令 ori 指令的目的寄存器号相同，故存在数据依赖。subu 指令在 EX 阶段具体执行情况如图 8-5 所示

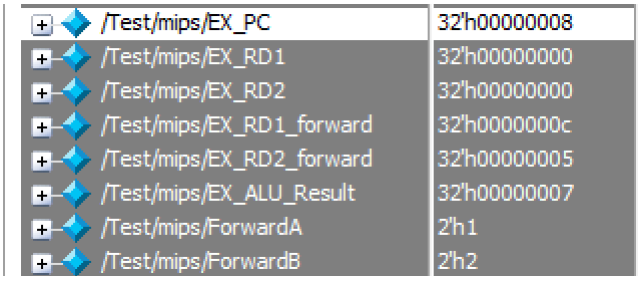

图 8-5

由图中可以看到旁路信号 ForwardA 和 ForwardB 产生正确的信号值 1 和 2（具体信号的意义已在旁路单元的代码实现中阐明），同时旁路的多路选择器的结果 EX_RD1_forward 和 EX_RD2_forward 也产生了正确的选择结果 12 和 5，将

旁路的数据送入运算器后得到的运算器结果 EX_ALU_Result 结果正确，故该旁路实现了设计的功能。

接下来分析另一种数据冒险，需通过旁路和阻塞共同来解决。产生数据冒险的指令如图 8-6 所示：

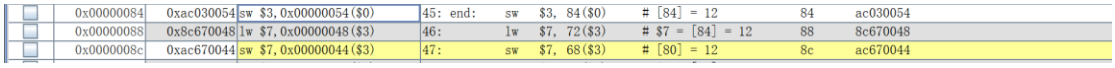

图 8-6

其中第一条 sw 和 lw 指令间的数据冒险可以通过旁路解决，其过程和上一种数据冒险的解决方式基本相同，在此不赘述。主要分析 lw 指令和第二条 sw 指令间的数据冒险，由流水线的知识可以知道，lw 在 MEM 阶段才能读取出待写入寄存器中的数据，而 sw 在 ID 阶段就需要访问寄存器获取数据，故存在数据冒险且无法完全通过旁路解决，需阻塞流水线一个时钟周期。

其指令执行情况如图 8-7 所示

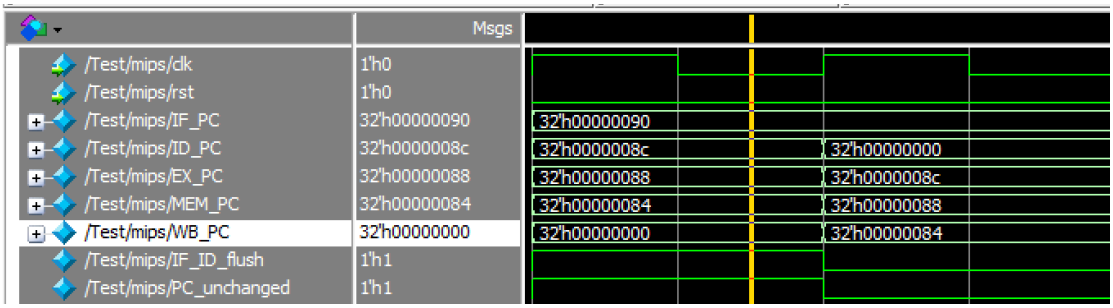

图 8-7

可以看到 lw 指令(pc 值为 0x88)执行到 EX 阶段时，冒险检测单元检测到数据冒险，故将 IF_ID_flush 和 PC_unchanged 这两个和冒险相关的控制信号都设置为有效，在下一个时钟周期 ID_PC 的值被设置为全 0，即一个空指令 nop 开始在流水线上传递，同时 IF_PC 的值保持不变，即被阻塞了一个时钟周期。

阻塞一个时钟周期后第二条 sw 便可以通过旁路获取到正确的数据，其执行情况如图 8-8 所示

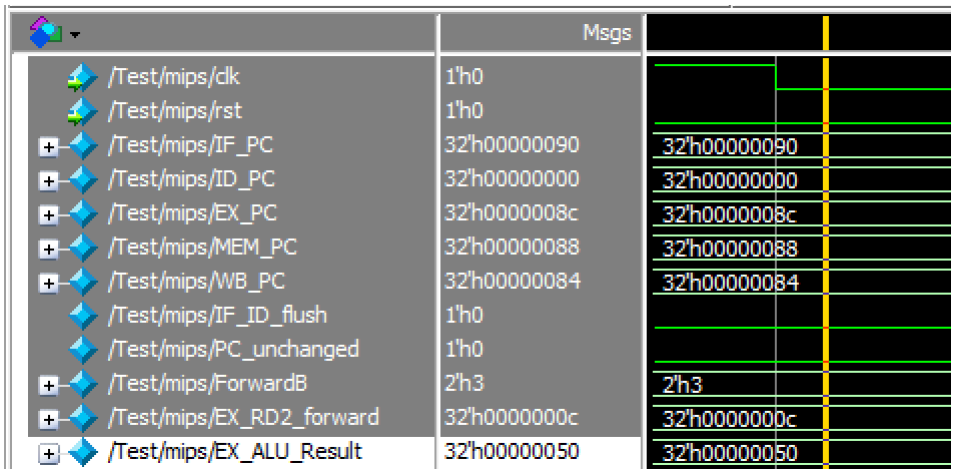

图 8-8

由图中可以看到旁路信号 ForwardB 产生正确的信号值 3（具体信号的意义已在旁路单元的代码实现中阐明），同时旁路的多路选择器的结果 EX_RD2_forward 产生了正确的选择结果 12，该数据和上一条 lw 指令读取出来的数据一致，故该旁路实现了设计的功能。

接下来分析控制冒险，产生控制冒险的指令如图 8-9 所示：

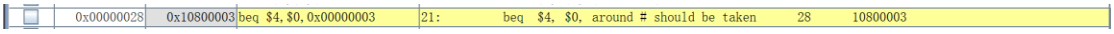

图 8-9

由注释可以看到，该分支指令将会执行，其具体执行情况如图 8-10 所示

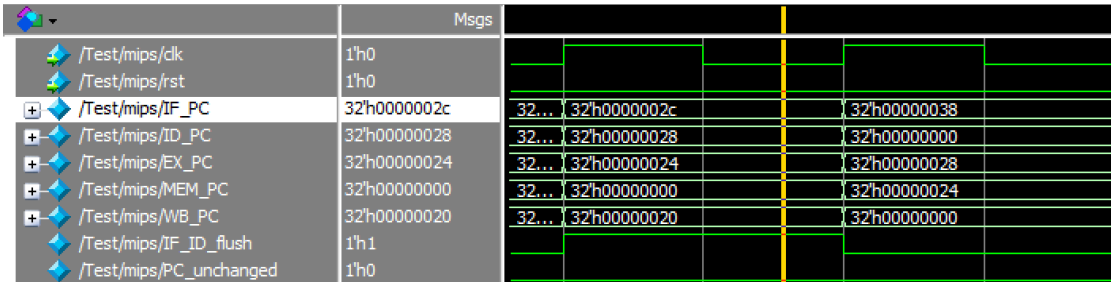

图 8-10

有图中可以看出，当 beq 指令执行到 ID 阶段时，冒险检测单元检测出了控制冒险，故将 IF_ID_flush 信号设置为有效，同时 NPC 采取静态预测的方式，预测分支不发生，故 NPC 产生的 PC 值为 0x2c，即 beq 指令的 PC 值+4，但由于 IF_ID_flush 信号为有效，故这个错误的 PC 值会被清楚，取而代之的是一条空指令在流水线中传递。从图 8-10 的第二个时钟周期可以看到，流水线中传递的是空指令 nop 而非错误的 PC+4 的值，同时此时 NPC 已经产生出正确的 PC 值 38，该控制冒险

通过阻塞流水线一个时钟周期的方式得以解决。

教师评语评分

评语：

评分：

评阅人：

年　　月　　日

（备注：对该实验报告给予优点和不足的评价，并给出百分之评分。）