2023秋P6·CPU设计文档

Part 1 CPU设计草稿

1.1 总体概述

以下是参考的结构图：

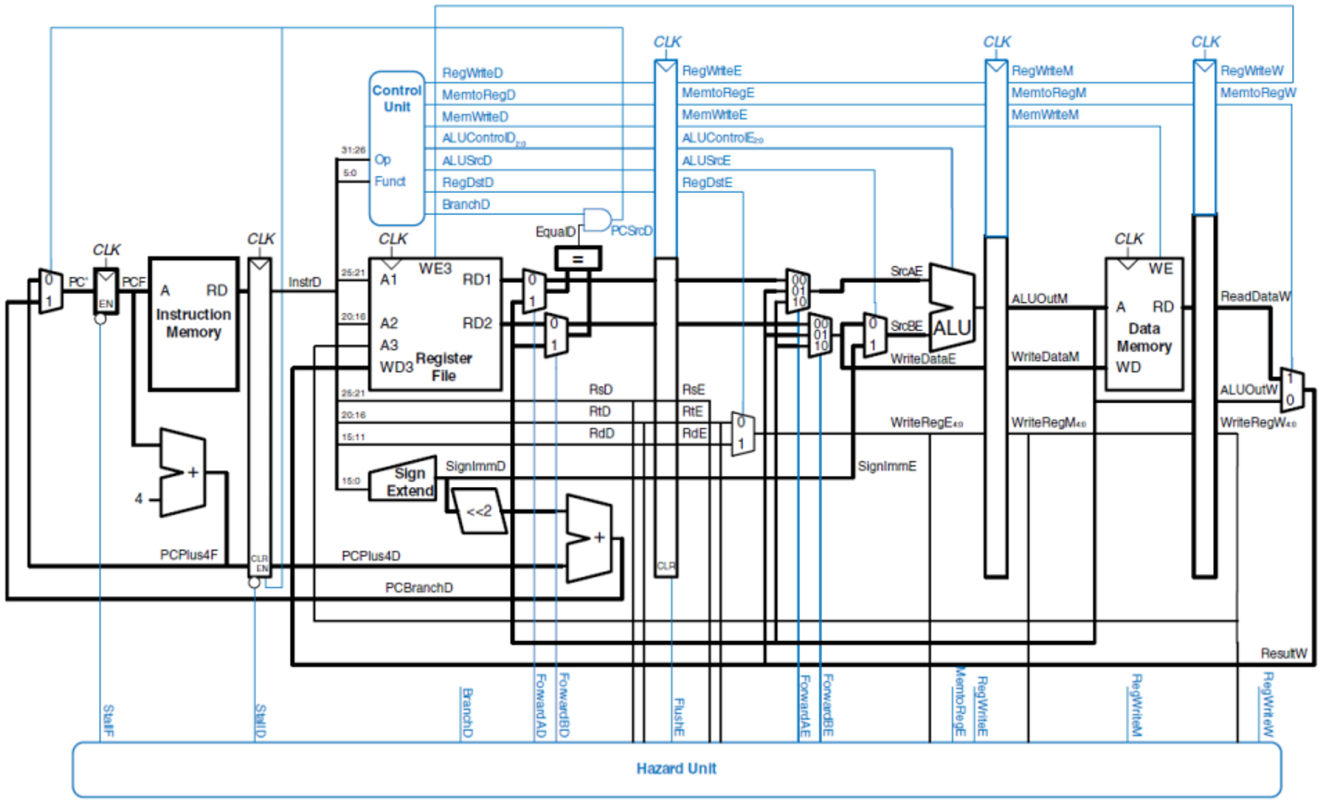

Figure 7.58 Pipelined processor with full hazard handling

1.1.1 CPU类型

本CPU为**五级流水线MIPS - CPU**，目前由**Verilog**的**ISE**工具链实现。

1.1.2 CPU结构

CPU包含**IFU、GRF、CMP、EXT、ALU、DM、MUDI_Processor、Controller、各级之间的流水线寄存器**等多个模块**以及冒险处理单元。**

> `MUXs.v`文件集成了4个多路选择器。
> `forward_MUXs.v`可以看作是冒险处理单元的附属模块，集成了转发机制需要的所有多路选择器。

1.1.3 CPU支持的指令集

为了测试开发方便，本次设计将完全支持**MIPS-C3指令集。**对该指令集按照**方便于冒险的解决模式**进行分类如下：

- **calr**类型指令
  {**add addu sub subu and or xor nor sll sllv sra srav srl srlv slt sltu**}
- **cali**类型指令
  {**lui ori addi addiu andi xori slti sltiu**}
- **calmudi**（乘除计算）类型指令
  {**mult multu div divu**}
- **readhl**（读乘除寄存器）类型指令
  {**mflo mfhi**}
- **writehl**（写乘除寄存器）类型指令
  {**mthi mtlo**}
- **jump1**类型指令
  {**jr jalr**}
- **jump2**类型指令
  {**j jal**}
- **branch**类型指令
  {**beq bgtz blez bgez bltz bne**}
- **load**类型指令
  {**lw lb lbu lh lhu**}
- **store**类型指令
  {**sw sb sh**}
- **nop**类型指令
  {**nop**}

1.2 模块设计

1.2.1 IFU

> 本模块**IM**部分外置，因此内部细节与接口定义较之前有小范围改动。

1.2.1.1 端口定义

端口名	方向	位宽	信号描述
clk	I	1	时钟信号
reset	I	1	同步复位信号，复位使PC的值为0x3000（起始地址）
enable	I	1	使能信号，当enable为1时，IFU开始工作；否则，IFU停止工作
branchEn	I	1	来自CMP部件的输出结果，1表示满足某1种branch型指令的跳转条件，0表示不满足
offset	I	16	跳转偏移量
instr_index	I	26	跳转目标地址
rsIn	I	32	GRF[rs]读出的值
D_pcPlus4	I	32	D阶段(译码阶段)的pc+4值

端口名	方向	位宽	信号描述
jumpOp	I	3	跳转操作类型：000，无跳转；001，branch（beq/bgtz/blez/bgez/bltz/bne）；010；jal/j；011，jr/jalr；
pc	O	32	PC值，用于显示当前指令的地址(按字节)

> 在本文档中，形如D_xxx的信号表示该信号是输入信号且来自D阶段，形如xxx_D的信号表示该信号是输出信号且送往D阶段。对于其他阶段也是如此。
> 该部分被分为**pc、npc、im三个子模块**,本设计将其写在同一个文件中，用注释区分。

1.2.1.2 功能定义

- 同步复位
  - 当reset为1时，将PC的值置为0x3000（起始地址）
- 计算下一条指令地址（调整PC值）
  - 当branch（beq/bgtz/blez/bgez/bltz/bne）指令有效时（branchEn == 1 && jumpOp == 001），PC = PC + 4 + offset * 4
  - 当j/jal指令有效时（jumpOp == 010），PC = pc[31:28] || instr_index || 00
  - 当jr/jalr指令有效时（jumpOp == 011），PC = rsIn
  - 否则，PC = PC + 4
- 取指令
  - 根据PC的值，从IM中读出下一条指令nInstr

1.2.1.3 模块代码

```
`timescale 1ns / 1ps
module IFU (
    input             clk,
    input             reset,
    input             enable,
    input             branchEn,
    input      [15:0] offset,
    input      [25:0] instr_index,
    input      [31:0] rsIn,
    input      [31:0] D_pcPlus4,
    input      [ 2:0] jumpOp,
    //output reg [31:0] nInstr,
    output reg [31:0] pc
);
  initial begin
    pc = 32'h00003000;
  end
  reg [31:0] rom  [0:4095];
  reg [31:0] pc_;
  reg [11:0] addr;

  // Sub-module1 npc
  reg [31:0] npc;
```

```
    always @(*) begin
      case (jumpOp)
        3'd0: begin
          // none
          npc = pc + 4;
        end
        3'd1: begin
          //branch (beq/bgtz/blez/bgez/bltz/bne)
          if (branchEn == 1'b1) begin
            // We have used CMP in D level instead of ALU in E level
            npc = D_pcPlus4 + ({{16{offset[15]}}, offset} << 2);
          end else begin
            npc = pc + 4;
          end
        end
        3'd2: begin
          //j/jal
          npc = {D_pcPlus4[31:28], instr_index, {2{1'b0}}};
        end
        3'd3: begin
          //jr/jalr
          npc = rsIn;
        end
        default: npc = pc + 4;
      endcase
    end

    // Sub-module2 pc
    always @(posedge clk) begin
      if (reset == 1'b1) begin
        pc <= 32'h00003000;
      end else if (enable) begin
        pc <= npc;
      end
    end

    // Sub-module3 im
    // initial begin
    //   $readmemh("code.txt", rom);
    // end
    // always @(*) begin
    //   pc_    = pc - 32'h00003000;
    //   addr   = pc_[13:2];
    //   nInstr = rom[addr];
    // end
  endmodule
```

1.2.2 IF_ID(流水线寄存器)

1.2.2.1 端口定义

端口名	方向	位宽	信号描述
clk	I	1	时钟信号
reset	I	1	同步复位信号
enable	I	1	使能信号
F_pc	I	32	F级（取指阶段）的pc值
F_nInstr	I	32	F级（取指阶段）的nInstr值
pc_D	O	32	D级（译码阶段）的pc值
pcPlus4_D	O	32	D级（译码阶段）的pc+4值
pcPlus8_D	O	32	D级（译码阶段）的pc+8值
nInstr_D	O	32	D级（译码阶段）的nInstr值

1.2.2.2 功能定义

- 暂存F阶段结果，向D阶段提供数据。

1.2.2.3 模块代码

```
`timescale 1ns / 1ps
module IF_ID (
    input             clk,
    input             reset,
    input             enable,
    input      [31:0] F_pc,
    input      [31:0] F_nInstr,
    output reg [31:0] pc_D,
    output reg [31:0] pcPlus4_D,
    output reg [31:0] pcPlus8_D,
    output reg [31:0] nInstr_D
);
  always @(posedge clk) begin
    if (reset == 1) begin
      pc_D      <= 32'h00003000;
      pcPlus4_D <= 32'h00003004;
      pcPlus8_D <= 32'h00003008;
      nInstr_D  <= 32'h00000000;
    end else if (enable) begin
      pc_D      <= F_pc;
      pcPlus4_D <= F_pc + 4;
      pcPlus8_D <= F_pc + 8;
      nInstr_D  <= F_nInstr;
    end
  end
endmodule
```

1.2.3 GRF

> 本模块原本的**display语句**也转移至外置文件中。

1.2.3.1 端口定义

端口名	方向	位宽	信号描述
clk	I	1	时钟信号
reset	I	1	同步复位信号
WE	I	1	写使能信号：1，允许写入；0，不可写入
A1	I	5	读出数据的地址1
A2	I	5	读出数据的地址2
A3	I	5	写入数据的地址
WD	I	32	写入数据
RD1	O	32	读出数据1
RD2	O	32	读出数据2

1.2.3.2 功能定义

- 读数据
  - 读数据1：根据地址A1，读出对应的数据RD1
  - 读数据2：根据地址A2，读出对应的数据RD2
- 写数据
  - 写数据：WE为1、reset不为1且时钟上升来临时，根据地址A3，写入数据WD
  - 会按照教程所要求格式输出写入信息
- 同步复位
  - 当reset为1时，将所有寄存器清零

1.2.3.3 模块代码

```
`timescale 1ns / 1ps
module GRF (
    input         clk,
    input         reset,
    input         WE,
    input  [ 4:0] A1,
    input  [ 4:0] A2,
    input  [ 4:0] A3,
    input  [31:0] WD,
    input  [31:0] pc,
    output [31:0] RD1,
    output [31:0] RD2
);
```

```
    reg     [31:0] regfile[0:31];
    integer        i = 0;
    always @(posedge clk) begin
      if (reset == 1'b1) begin
        for (i = 0; i < 32; i = i + 1) begin
          regfile[i] <= 32'd0;
        end
      end else begin
        if (WE == 1'b1) begin
          if (A3 != 5'd0) begin
            regfile[A3] <= WD;
            //display for test
            //$display("%d@%h: $%d <= %h", $time, pc, A3, WD);
          end else begin
            regfile[0] <= 32'd0;
          end
        end
      end
    end
    assign RD1 = (WE == 1 && A3 != 5'd0 && A3 == A1) ? WD : regfile[A1];
    assign RD2 = (WE == 1 && A3 != 5'd0 && A3 == A2) ? WD : regfile[A2];
  endmodule
```

1.2.4 CMP

1.2.4.1 端口定义

端口名	方向	位宽	信号描述
A	I	32	第一个操作数（A）
B	I	32	第二个操作数（B）
branchType	I	3	D级控制器输出的branch类型指令的具体类别：000，none；001，beq；010，bgtz；011，blez；100，bgez；101，bltz；110；bne
branchEn	O	1	某种branch类型指令是否达到跳转条件

1.2.4.2 功能定义

- 判断相等
  - 判断A和B是否满足某个具体的条件，满足时branchEn为1，不满足时branchEn为0

1.2.4.3 模块代码

```
  `timescale 1ns / 1ps
  module CMP (
      input      [31:0] A,
      input      [31:0] B,
```

```
    input      [ 2:0] branchType,
    output reg        branchEn
);
  always @(*) begin
    case (branchType)
      3'd0: begin
        //none
        branchEn = 0;
      end
      3'd1: begin
        //beq
        if (A == B) begin
          branchEn = 1;
        end else begin
          branchEn = 0;
        end
      end
      3'd2: begin
        //bgtz
        if ($signed(A) > 0) begin
          branchEn = 1;
        end else begin
          branchEn = 0;
        end
      end
      3'd3: begin
        //blez
        if ($signed(A) <= 0) begin
          branchEn = 1;
        end else begin
          branchEn = 0;
        end
      end
      3'd4: begin
        //bgez
        if ($signed(A) >= 0) begin
          branchEn = 1;
        end else begin
          branchEn = 0;
        end
      end
      3'd5: begin
        //bltz
        if ($signed(A) < 0) begin
          branchEn = 1;
        end else begin
          branchEn = 0;
        end
      end
      3'd6: begin
        //bne
        if (A == B) begin
          branchEn = 0;
        end else begin
```

```
            branchEn = 1;
          end
        end
        default: branchEn = 0;
      endcase
    end
  endmodule
```

1.2.5 EXT

1.2.5.1 端口定义

端口名	方向	位宽	信号描述
dataIn	I	16	输入数据,16位立即数
extOp	I	2	扩展操作类型：00，无符号扩展；01，符号扩展
dataOut	O	32	输出数据,32位立即数

1.2.5.2 功能定义

- 位扩展
  - 将16位的立即数扩展为32位的立即数，提供符号扩展、无符号扩展两种方式

1.2.5.3 模块代码

```
`timescale 1ns / 1ps
module EXT (
    input      [15:0] dataIn,
    input      [ 1:0] extOp,
    output reg [31:0] dataOut
);
  always @(*) begin
    if (extOp == 2'b00) begin
      dataOut = {{16{1'b0}}, dataIn};
    end else if (extOp == 2'b01) begin
      dataOut = {{16{dataIn[15]}}, dataIn};
    end else begin
      dataOut = 0;
    end
  end
endmodule
```

1.2.6 ID_EX(流水线寄存器)

1.2.6.1 端口定义

端口名	方向	位宽	信号描述
clk	I	1	时钟信号
reset	I	1	同步复位信号
enable	I	1	使能信号
D_nInstr	I	32	D阶段的nInstr值
D_pc	I	32	D阶段的pc值
D_pcPlus4	I	32	D阶段的pc+4值
D_pcPlus8	I	32	D阶段的pc+8值
D_RD1	I	32	D阶段从rs寄存器读出的数据RD1
D_RD2	I	32	D阶段从rt寄存器读出的数据RD2
D_dataOut	I	32	D阶段的扩展后的立即数dataOut
nInstr_E	O	32	E阶段的nInstr值
pc_E	O	32	E阶段的pc值
pcPlus4_E	O	32	E阶段的pc+4值
pcPlus8_E	O	32	E阶段的pc+8值
rsData_E	O	32	E阶段的从rs寄存器读出的数据
rtData_E	O	32	E阶段的从rt寄存器读出的数据
extImm_E	O	32	E阶段的扩展后的立即数extImm

1.2.6.2 功能定义

- 暂存D阶段结果，向E阶段提供数据。

1.2.6.3 模块代码

```
`timescale 1ns / 1ps
module ID_EX (
    input             clk,
    input             reset,
    input             enable,
    input      [31:0] D_nInstr,
    input      [31:0] D_pc,
    input      [31:0] D_pcPlus4,
    input      [31:0] D_pcPlus8,
    input      [31:0] D_RD1,
    input      [31:0] D_RD2,
    input      [31:0] D_dataOut,
    output reg [31:0] nInstr_E,
    output reg [31:0] pc_E,
```

```
    output reg [31:0] pcPlus4_E,
    output reg [31:0] pcPlus8_E,
    output reg [31:0] rsData_E,
    output reg [31:0] rtData_E,
    output reg [31:0] extImm_E
);

  always @(posedge clk) begin
    if (reset) begin
      nInstr_E  <= 0;
      pc_E      <= 32'h00003000;
      pcPlus4_E <= 32'h00003004;
      pcPlus8_E <= 32'h00003008;
      rsData_E  <= 0;
      rtData_E  <= 0;
      extImm_E  <= 0;
    end else if (enable) begin
      nInstr_E  <= D_nInstr;
      pc_E      <= D_pc;
      pcPlus4_E <= D_pcPlus4;
      pcPlus8_E <= D_pcPlus8;
      rsData_E  <= D_RD1;
      rtData_E  <= D_RD2;
      extImm_E  <= D_dataOut;
    end
  end
endmodule
```

1.2.7 ALU

1.2.7.1 端口定义

端口名	方向	位宽	信号描述
A	I	32	第一个操作数
B	I	32	第二个操作数
shamt	I	5	移位位数
aluOp	I	4	ALU操作类型：0000，加；0001，减；0010，按位与；0011，按位或；0100，异或（xor）；0101，或非（nor）； 0110，逻辑左移；0111，逻辑右移；1000，算数右移；1001，按无符号数相比小于； 1010：按有符号数相比小于 1011：低16位加载到高位且低位补0
R	O	32	运算结果

1.2.7.2 功能定义

- ALU操作类型：
  - 0000，加；

- 0001，减；
- 0010，按位与；
- 0011，按位或；
- 0100，异或（xor）；
- 0101，或非（nor）；
- 0110，逻辑左移；
- 0111，逻辑右移；
- 1000，算数右移；
- 1001，按无符号数相比小于；
- 1010：按有符号数相比小于；
- 1011：低16位加载到高位且低位补0

1.2.7.3 模块代码

```
`timescale 1ns / 1ps
module ALU (
    input      [31:0] A,
    input      [31:0] B,
    input      [ 3:0] aluOp,
    input      [ 4:0] shamt,
    output reg [31:0] R
);
  always @(*) begin
    case (aluOp)
      4'd0: R = A + B;
      4'd1: R = A - B;
      4'd2: R = A & B;
      4'd3: R = A | B;
      4'd4: R = A ^ B;
      4'd5: R = ~(A | B);
      4'd6: R = B << shamt;
      4'd7: R = B >> shamt;
      4'd8: R = ($signed(B)) >>> shamt;
      4'd9: R = (A < B) ? 32'd1 : 32'd0;  // sltu sltiu
      4'd10: R = ($signed(A) < $signed(B)) ? 32'd1 : 32'd0;  // slt slti
      4'd11: R = {B[15:0], {16{1'b0}}};  // lui
      default: R = R;
    endcase
  end
endmodule
```

1.2.8 EX_MEM(流水线寄存器)

1.2.8.1 端口定义

端口名	方向	位宽	信号描述
clk	I	1	时钟信号

端口名	方向	位宽	信号描述
reset	I	1	同步复位信号
enable	I	1	使能信号
E_nInstr	I	32	E阶段的nInstr值
E_pc	I	32	E阶段的pc值
E_pcPlus4	I	32	E阶段的pc+4值
E_pcPlus8	I	32	E阶段的pc+8值
E_aluRes	I	32	E阶段的ALU运算结果aluRes
E_rtData	I	32	E阶段的从rt寄存器读出的数据
E_extImm	I	32	E阶段的扩展后的立即数extImm
E_hiloData	I	32	E阶段的从乘除寄存器读出的数据
nInstr_M	O	32	M阶段的nInstr值
pc_M	O	32	M阶段的pc值
pcPlus4_M	O	32	M阶段的pc+4值
pcPlus8_M	O	32	M阶段的pc+8值
aluRes_M	O	32	M阶段的ALU运算结果aluRes
rtData_M	O	32	M阶段的从rt寄存器读出的数据
extImm_M	O	32	M阶段的扩展后的立即数extImm
hiloData_M	O	32	M阶段的从乘除寄存器读出的数据

1.2.8.2 功能定义

- 暂存E阶段结果，向M阶段提供数据。

1.2.8.3 模块代码

```
`timescale 1ns / 1ps
module EX_MEM (
    input             clk,
    input             reset,
    input             enable,
    input      [31:0] E_nInstr,
    input      [31:0] E_pc,
    input      [31:0] E_pcPlus4,
    input      [31:0] E_pcPlus8,
    input      [31:0] E_rtData,
    input      [31:0] E_aluRes,
    input      [31:0] E_extImm,
```

```
    input      [31:0] E_hiloData,
    output reg [31:0] nInstr_M,
    output reg [31:0] pc_M,
    output reg [31:0] pcPlus4_M,
    output reg [31:0] pcPlus8_M,
    output reg [31:0] rtData_M,
    output reg [31:0] aluRes_M,
    output reg [31:0] extImm_M,
    output reg [31:0] hiloData_M
);
  always @(posedge clk) begin
    if (reset) begin
      nInstr_M   <= 0;
      pc_M       <= 32'h00003000;
      pcPlus4_M  <= 32'h00003004;
      pcPlus8_M  <= 32'h00003008;
      rtData_M   <= 0;
      aluRes_M   <= 0;
      extImm_M   <= 0;
      hiloData_M <= 0;
    end else if (enable) begin
      nInstr_M   <= E_nInstr;
      pc_M       <= E_pc;
      pcPlus4_M  <= E_pcPlus4;
      pcPlus8_M  <= E_pcPlus8;
      rtData_M   <= E_rtData;
      aluRes_M   <= E_aluRes;
      extImm_M   <= E_extImm;
      hiloData_M <= E_hiloData;
    end
  end
endmodule
```

1.2.9 DM

> 本模块**main_dm**部分外置，因此内部细节与接口定义较之前有小范围改动。

1.2.9.1 端口定义

端口名	方向	位宽	信号描述
clk	I	1	时钟信号
reset	I	1	同步复位信号
WE	I	1	写使能信号：1，允许写入；0，不可写入
RE	I	1	读使能信号：1，允许读出；0，不可读出
storeOp	I	2	store型指令具体类别：00，sw（存整字）；01，sh（存半字）；10，sb（存一个字节）

端口名	方向	位宽	信号描述
loadOp	I	3	load型指令具体类别：000，lw；001，lbu；010，lb；011，lhu；100，lh
A	I	32	读写地址：输入值是按字节为单位；容量是4096字；
m_data_rdata	I	32	外置dm读出的32位原始数据（需要经过load_helper处理）
RD	O	32	经过处理后的读出数据
writeBytesEn	O	4	字节写使能信号

> 本模块为了适应多种类型的存取指令进行了扩展，共分为**store_helper、main_dm、load_helper**三个子模块，写在同一个文件中，用注释区分。
> **store_helper**主要功能是根据具体的某条store指令要求，通知**main_dm**写数据的具体位置。
> **load_helper**主要功能是根据具体的某条load指令要求，从**main_dm**取出正确格式的数据。

1.2.9.2 功能定义

- 读数据
  - 读数据：RE为1时且时钟上升来临时，根据地址A，读出数据RD;否则读出0。
- 写数据
  - 写数据：WE为1时且时钟上升来临时，根据地址A，写入数据WD；否则保持原值。
  - 会按照教程所要求格式输出写入信息
- 同步复位
  - 当reset为1时，将所有内容清零

1.2.9.3 模块代码

```
`timescale 1ns / 1ps
module DM (
    input             clk,
    input             reset,
    input             WE,
    input             RE,
    input      [ 1:0] storeOp,
    input      [ 2:0] loadOp,
    input      [31:0] A,
    //input      [31:0] pc,
    //input      [31:0] WD,
    input      [31:0] m_data_rdata,
    output reg [ 3:0] writeBytesEn,
    output reg [31:0] RD
);
  reg     [31:0] ram   [0:4095];
  // real address
  wire    [11:0] addr;
  integer        i = 0;
  // 0-1 bits of input value A
  wire    [ 1:0] A_1_0;
```

```
  assign A_1_0 = A[1:0];

  // Sub-module1 store_helper
  always @(*) begin
    if (WE == 1'b0 || reset) begin
      writeBytesEn = 4'b0000;
    end else begin
      writeBytesEn = 4'b0000;
      case (storeOp)
        2'd0: begin  //sw
          writeBytesEn = 4'b1111;
        end
        2'd1: begin  //sh
          writeBytesEn = (A_1_0[1] == 1'b1) ? 4'b1100 : 4'b0011;
        end
        2'd2: begin  //sb
          writeBytesEn = (A_1_0 == 2'b00) ? 4'b0001 : (A_1_0 == 2'b01) ?
4'b0010 : (A_1_0 == 2'b10) ? 4'b0100 : (A_1_0 == 2'b11) ? 4'b1000 :
4'b0000;
        end
        default: begin
          writeBytesEn = 4'b0000;
        end
      endcase
    end
  end

  // Sub-module2 main_dm
  // assign addr = A[13:2];
  // always @(posedge clk) begin
  //   if (reset == 1'b1) begin
  //     for (i = 0; i < 4096; i = i + 1) begin
  //       ram[i] <= 32'd0;
  //     end
  //   end else begin
  //     if (WE == 1'b1) begin
  //       case (writeBytesEn)
  //         4'b0001: begin
  //           ram[addr][7:0] <= WD[7:0];
  //           //$display("%d@%h: *%h <= %h", $time, pc, A / 4 * 4,
{ram[addr][31:8], WD[7:0]});
  //           $display("@%h: *%h <= %h", pc, A / 4 * 4, {ram[addr][31:8],
WD[7:0]});
  //         end
  //         4'b0010: begin
  //           ram[addr][15:8] <= WD[7:0];
  //           //$display("%d@%h: *%h <= %h", $time, pc, A / 4 * 4,
{ram[addr][31:16], WD[7:0], ram[addr][7:0]});
  //           $display("@%h: *%h <= %h", pc, A / 4 * 4, {ram[addr][31:16],
WD[7:0], ram[addr][7:0]});
  //         end
  //         4'b0100: begin
  //           ram[addr][23:16] <= WD[7:0];
  //           //$display("%d@%h: *%h <= %h", $time, pc, A / 4 * 4,
```

```
{ram[addr][31:24], WD[7:0], ram[addr][15:0]});
  //           $display("@%h: *%h <= %h", pc, A / 4 * 4, {ram[addr][31:24],
WD[7:0], ram[addr][15:0]});
  //         end
  //         4'b1000: begin
  //           ram[addr][31:24] <= WD[7:0];
  //           //$display("%d@%h: *%h <= %h", $time, pc, A / 4 * 4,
{WD[7:0], ram[addr][23:0]});
  //           $display("@%h: *%h <= %h", pc, A / 4 * 4, {WD[7:0],
ram[addr][23:0]});
  //         end
  //         4'b0011: begin
  //           ram[addr][15:0] <= WD[15:0];
  //           //$display("%d@%h: *%h <= %h", $time, pc, A / 4 * 4,
{ram[addr][31:16], WD[15:0]});
  //           $display("@%h: *%h <= %h", pc, A / 4 * 4, {ram[addr][31:16],
WD[15:0]});
  //         end
  //         4'b1100: begin
  //           ram[addr][31:16] <= WD[15:0];
  //           //display for test
  //           //$display("%d@%h: *%h <= %h", $time, pc, A / 4 * 4,
{WD[15:0], ram[addr][15:0]});
  //           $display("@%h: *%h <= %h", pc, A / 4 * 4, {WD[15:0],
ram[addr][15:0]});
  //         end
  //         4'b1111: begin
  //           ram[addr] <= WD;
  //           //display for test
  //           //$display("%d@%h: *%h <= %h", $time, pc, A / 4 * 4, WD);
  //           $display("@%h: *%h <= %h", pc, A / 4 * 4, WD);
  //         end
  //       endcase
  //     end else begin
  //       ram[addr] <= ram[addr];
  //     end
  //   end
  // end

  // Sub-module3 load_helper
  always @(*) begin
    if (RE == 1) begin
      case (loadOp)
        3'd0: begin  //lw
          RD = m_data_rdata;
        end
        3'd1: begin  //lbu
          case (A_1_0)
            2'b00:   RD = {{24{1'b0}}, m_data_rdata[7:0]};
            2'b01:   RD = {{24{1'b0}}, m_data_rdata[15:8]};
            2'b10:   RD = {{24{1'b0}}, m_data_rdata[23:16]};
            2'b11:   RD = {{24{1'b0}}, m_data_rdata[31:24]};
            default: RD = 32'b0;
          endcase
```

```
          end
          3'd2: begin  // lb
            case (A_1_0)
              2'b00:   RD = {{24{m_data_rdata[7]}}, m_data_rdata[7:0]};
              2'b01:   RD = {{24{m_data_rdata[15]}}, m_data_rdata[15:8]};
              2'b10:   RD = {{24{m_data_rdata[23]}}, m_data_rdata[23:16]};
              2'b11:   RD = {{24{m_data_rdata[31]}}, m_data_rdata[31:24]};
              default: RD = 32'b0;
            endcase
          end
          3'd3: begin  //lhu
            case (A_1_0[1])
              1'b0: RD = {{16{1'b0}}, m_data_rdata[15:0]};
              1'b1: RD = {{16{1'b0}}, m_data_rdata[31:16]};
              default: RD = 32'b0;
            endcase
          end
          3'd4: begin  //lh
            case (A_1_0[1])
              1'b0: RD = {{16{m_data_rdata[15]}}, m_data_rdata[15:0]};
              1'b1: RD = {{16{m_data_rdata[31]}}, m_data_rdata[31:16]};
              default: RD = 32'b0;
            endcase
          end
          default: RD = 32'b0;
        endcase
      end else begin
        RD = 32'b0;
      end
    end
  endmodule
```

1.2.10 MEM_WB(流水线寄存器)

1.2.10.1 端口定义

端口名	方向	位宽	信号描述
clk	I	1	时钟信号
reset	I	1	同步复位信号
enable	I	1	使能信号
M_nInstr	I	32	M阶段的nInstr值
M_pc	I	32	M阶段的pc值
M_pcPlus4	I	32	M阶段的pc+4值
M_pcPlus8	I	32	M阶段的pc+8值
M_aluRes	I	32	M阶段的ALU运算结果aluRes

端口名	方向	位宽	信号描述
M_rtData	I	32	M阶段的从rt寄存器读出的数据
M_extImm	I	32	M阶段的扩展后的立即数extImm
M_dmData	I	32	M阶段的从DM读出的数据dmData
M_hiloData	I	32	M阶段从乘除寄存器中读出的结果
nInstr_W	O	32	W阶段的nInstr值
pc_W	O	32	W阶段的pc值
pcPlus4_W	O	32	W阶段的pc+4值
pcPlus8_W	O	32	W阶段的pc+8值
aluRes_W	O	32	W阶段的ALU运算结果aluRes
rtData_W	O	32	W阶段的从rt寄存器读出的数据
extImm_W	O	32	W阶段的扩展后的立即数extImm
dmData_W	O	32	W阶段的从DM读出的数据dmData
hiloData_W	O	32	W阶段从乘除寄存器中读出的结果

1.2.10.2 功能定义

- 暂存M阶段结果，向W阶段提供数据。

1.2.10.3 模块代码

```
`timescale 1ns / 1ps
module MEM_WB (
    input              clk,
    input              reset,
    input              enable,
    input      [31:0] M_nInstr,
    input      [31:0] M_pc,
    input      [31:0] M_pcPlus4,
    input      [31:0] M_pcPlus8,
    input      [31:0] M_rtData,
    input      [31:0] M_aluRes,
    input      [31:0] M_extImm,
    input      [31:0] M_dmData,
    input      [31:0] M_hiloData,
    output reg [31:0] nInstr_W,
    output reg [31:0] pc_W,
    output reg [31:0] pcPlus4_W,
    output reg [31:0] pcPlus8_W,
    output reg [31:0] rtData_W,
    output reg [31:0] aluRes_W,
    output reg [31:0] extImm_W,
```

```
    output reg [31:0] dmData_W,
    output reg [31:0] hiloData_W
);
  always @(posedge clk) begin
    if (reset) begin
      nInstr_W   <= 0;
      pc_W       <= 32'h00003000;
      pcPlus4_W  <= 32'h00003004;
      pcPlus8_W  <= 32'h00003008;
      rtData_W   <= 0;
      aluRes_W   <= 0;
      extImm_W   <= 0;
      dmData_W   <= 0;
      hiloData_W <= 0;
    end else if (enable) begin
      nInstr_W   <= M_nInstr;
      pc_W       <= M_pc;
      pcPlus4_W  <= M_pcPlus4;
      pcPlus8_W  <= M_pcPlus8;
      rtData_W   <= M_rtData;
      aluRes_W   <= M_aluRes;
      extImm_W   <= M_extImm;
      dmData_W   <= M_dmData;
      hiloData_W <= M_hiloData;
    end
  end
endmodule
```

1.2.11 MUDI_Processor

1.2.11.1 端口定义

端口名	方向	位宽	信号描述
clk	I	1	时钟信号
reset	I	1	同步复位信号
A	I	32	第一个操作数
B	I	32	第二个操作数
mudiOp	I	3	calmudi型指令的具体类型：000，mult；001，multu；010，div；011，divu
start	I	1	启动信号
hiRead	I	1	hi寄存器读信号
loRead	I	1	lo寄存器读信号
hiWrite	I	1	hi寄存器写信号
loWrite	I	1	lo寄存器写信号

端口名	方向	位宽	信号描述
HI	O	32	hi寄存器值
LO	O	32	lo寄存器值
mudiRD	O	32	本次读出的值
busy	O	1	该部件忙信号

1.2.11.2 功能定义

- 根据具体指令类型，执行符号乘、无符号乘、符号除和无符号除。
- 根据具体指令类型，读写hi或lo寄存器

1.2.11.3 模块代码

```
`timescale 1ns / 1ps
module MUDI_Processor (
    input              clk,
    input              reset,
    input       [31:0] A,
    input       [31:0] B,
    input              start,
    input       [ 2:0] mudiOp,
    input              hiWrite,
    input              loWrite,
    input              hiRead,
    input              loRead,
    output reg  [31:0] HI,
    output reg  [31:0] LO,
    output wire        busy,
    output wire [31:0] mudiRD
);
  reg [31:0] hi;
  reg [31:0] lo;
  reg [ 3:0] cnt;
  initial begin
    HI  = 0;
    LO  = 0;
    hi  = 0;
    lo  = 0;
    cnt = 0;
  end
  always @(posedge clk) begin
    if (reset) begin
      HI  = 0;
      LO  = 0;
      hi  = 0;
      lo  = 0;
      cnt = 0;
    end else if (start) begin
```

```
        // start
        case (mudiOp)
          3'd0: begin
            // mult
            {hi, lo} <= $signed(A) * $signed(B);
            // simulate 5 cycles
            cnt      <= 5;
          end
          3'd1: begin
            // multu
            {hi, lo} <= A * B;
            cnt      <= 5;
          end
          3'd2: begin
            // div
            lo  <= $signed(A) / $signed(B);
            hi  <= $signed(A) % $signed(B);
            // simulate 10 cycles
            cnt <= 10;
          end
          3'd3: begin
            // divu
            lo  <= A / B;
            hi  <= A % B;
            cnt <= 10;
          end
        endcase
      end else if (hiWrite) begin
        HI <= A;
      end else if (loWrite) begin
        LO <= A;
      end else if (cnt != 0) begin
        // simulate for 5 or 10 cycles
        if (cnt == 4'd1) begin
          LO <= lo;
          HI <= hi;
        end
        cnt <= cnt - 1;
      end
    end
    assign busy   = (start || (cnt) != 0) ? 1 : 0;
    assign mudiRD = (hiRead) ? HI : (loRead) ? LO : 0;
  endmodule
```

1.2.12 Controller

> 在此前历次设计中，均采用了**指令驱动型**的译码方式，原因是这样**表达直观且易于理解**。但是现在C3指令集有50条指令，如果继续按照指令驱动会使代码篇幅过长。故本设计采用**控制信号驱动型**译码，并利用宏文件尽量使代码简洁易懂。

1.2.12.1 端口定义

端口名	方向	位宽	信号描述
nInstr	I	32	当前指令
op	I	6	操作码
fn	I	6	功能码
regDst	O	2	寄存器堆写入地址选通信号：00，选用rt为地址写入；01，选用rd为地址写入；10，选用$ra为目标写入
aluSrc	O	1	ALU第二个操作数（B）选通信号：1，选用立即数；0，选用寄存器堆读出的数据
aluOp	O	4	ALU操作类型：0000，加；0001，减；0010，按位与；0011，按位或；0100，异或（xor）；0101，或非（nor）； 0110，逻辑左移；0111，逻辑右移；1000，算数右移；1001，按无符号数相比小于； 1010：按有符号数相比小于 1011：低16位加载到高位且低位补0
memToR	O	3	数据写入寄存器堆时的数据来源：000，来自ALU运算结果；001，来自DM；010，写入pc+4的值；011，来自hi或者lo寄存器
memWrite	O	1	DM写使能信号：1，允许写入；0，不可写入
memRead	O	1	DM读使能信号：1，允许读出；0，不可读出
regWrite	O	1	寄存器堆写使能信号：1，允许写入；0，不可写入
extOp	O	2	扩展操作类型：00，无符号扩展；01，符号扩展
jumpOp	O	3	跳转操作类型：000，无跳转；001，branch（beq/bgtz/blez/bgez/bltz/bne）；010；jal/j；011，jr/jalr；
mudiOp	O	3	calmudi型指令的具体类型：000，mult；001，multu；010，div；011，divu
start	O	1	乘除部件启动信号
hiRead	O	1	hi寄存器读信号
loRead	O	1	lo寄存器读信号
hiWrite	O	1	hi寄存器写信号
loWrite	O	1	lo寄存器写信号
branchType	O	3	branch类型指令的具体类别：000，none；001，beq；010，bgtz；011，blez；100，bgez；101，bltz；110；bne
storeOp	O	2	store型指令具体类别：00，sw（存整字）；01，sh（存半字）；10，sb（存一个字节）
loadOp	O	3	load型指令具体类别：000，lw；001，lbu；010，lb；011，lhu；100，lh
isShamt	O	1	原本rs处的值是否看作移位值（sll、srl、sra）

1.2.12.2 功能定义

- 控制信号生成
  - 识别指令的操作码和功能码，根据指令的不同，生成不同的控制信号

1.2.12.3 模块代码

```
`timescale 1ns / 1ps
`include "Define.v"
module Controller (
    input  [31:0] nInstr,
    input  [ 5:0] op,
    input  [ 5:0] fn,
    output [ 1:0] regDst,
    output        aluSrc,
    output        regWrite,
    output        memRead,
    output        memWrite,
    output [ 2:0] memToR,
    output [ 1:0] extOp,
    output [ 2:0] branchType,
    output [ 2:0] jumpOp,
    output [ 3:0] aluOp,
    output        isShamt,
    output [ 2:0] mudiOp,
    output        hiWrite,
    output        loWrite,
    output        hiRead,
    output        loRead,
    output        start,
    output [ 2:0] loadOp,
    output [ 1:0] storeOp
);

  // for identify Bgez and Bltz
  wire nInstr_20_16;
  assign nInstr_20_16 = nInstr[20:16];

  // all instrs
  // calr
  reg Addu, Subu, Sub, Add, And, Or, Xor, Nor, Sll, Sllv, Slt, Sltu, Sra,
Srav, Srl, Srlv;
  // jump1 jump2
  reg Jr, Jalr, Jal, J;
  // cali
  reg Lui, Ori, Addi, Addiu, Andi, Xori, Slti, Sltiu;
  // branch
  reg Beq, Bne, Bgtz, Blez, Bgez, Bltz;
  //readhl
  reg Mfhi, Mflo;
  //writehl
  reg Mthi, Mtlo;
  //calmudi
  reg Mult, Multu, Div, Divu;
```

```
    //load
    reg Lw, Lb, Lbu, Lh, Lhu;
    //store
    reg Sw, Sb, Sh;
    //nop
    reg Nop;

    // identify instr
    always @(*) begin
      Addu  = 0;
      Subu  = 0;
      Sub   = 0;
      Add   = 0;
      And   = 0;
      Or    = 0;
      Xor   = 0;
      Nor   = 0;
      Sll   = 0;
      Sllv  = 0;
      Slt   = 0;
      Sltu  = 0;
      Sra   = 0;
      Srav  = 0;
      Srl   = 0;
      Srlv  = 0;
      Beq   = 0;
      Bne   = 0;
      Bgtz  = 0;
      Blez  = 0;
      Bgez  = 0;
      Bltz  = 0;
      Jr    = 0;
      Jalr  = 0;
      Jal   = 0;
      J     = 0;
      Sw    = 0;
      Sb    = 0;
      Sh    = 0;
      Lw    = 0;
      Lb    = 0;
      Lbu   = 0;
      Lh    = 0;
      Lhu   = 0;
      Mfhi  = 0;
      Mflo  = 0;
      Mthi  = 0;
      Mtlo  = 0;
      Mult  = 0;
      Multu = 0;
      Div   = 0;
      Divu  = 0;
      Lui   = 0;
      Ori   = 0;
      Addi  = 0;
```

```
      Addiu = 0;
      Andi  = 0;
      Xori  = 0;
      Slti  = 0;
      Sltiu = 0;
      Nop   = 0;
      case (op)
        // all R-types
        `R: begin
          case (fn)
            `Addu: Addu = 1;
            `Subu: Subu = 1;
            `Sub: Sub = 1;
            `Add: Add = 1;
            `And: And = 1;
            `Or: Or = 1;
            `Xor: Xor = 1;
            `Nor: Nor = 1;
            6'b000000: begin
              if (nInstr != 32'b0) Sll = 1;
              else Nop = 1;
            end
            `Sllv: Sllv = 1;
            `Slt: Slt = 1;
            `Sltu: Sltu = 1;
            `Sra: Sra = 1;
            `Srav: Srav = 1;
            `Srl: Srl = 1;
            `Srlv: Srlv = 1;
            `Jr: Jr = 1;
            `Jalr: Jalr = 1;
            `Mfhi: Mfhi = 1;
            `Mflo: Mflo = 1;
            `Mthi: Mthi = 1;
            `Mtlo: Mtlo = 1;
            `Mult: Mult = 1;
            `Multu: Multu = 1;
            `Div: Div = 1;
            `Divu: Divu = 1;
            default: Nop = 1;
          endcase
        end
        // I-type branch
        `Beq: Beq = 1;
        `Bne: Bne = 1;
        `Bgtz: Bgtz = 1;
        `Blez: Blez = 1;
        `BgezOrBltz: begin
          if (nInstr_20_16 == 5'b0) Bltz = 1;
          else if (nInstr_20_16 == 5'b1) Bgez = 1;
        end
        // I-type cali
        `Lui: Lui = 1;
        `Ori: Ori = 1;
```

```
        `Addi: Addi = 1;
        `Addiu: Addiu = 1;
        `Andi: Andi = 1;
        `Xori: Xori = 1;
        `Slti: Slti = 1;
        `Sltiu: Sltiu = 1;
        // I-type load
        `Lw: Lw = 1;
        `Lb: Lb = 1;
        `Lbu: Lbu = 1;
        `Lh: Lh = 1;
        `Lhu: Lhu = 1;
        // I-type store
        `Sw: Sw = 1;
        `Sb: Sb = 1;
        `Sh: Sh = 1;
        // J-type jump2
        `J: J = 1;
        `Jal: Jal = 1;
        default: Nop = 1;
      endcase
    end

    //category all instrs
    wire cali, calr, calmudi, store, load, branch, jump1, jump2, writehl,
readhl;
    assign cali = Lui | Ori | Addi | Addiu | Andi | Xori | Slti | Sltiu;
    assign calr = Add | Addu | Sub | Subu | And | Or | Xor | Nor | Sll | Sllv
| Sra | Srav | Srl | Srlv | Slt | Sltu;
    assign calmudi = Mult | Multu | Div | Divu;
    assign readhl = Mflo | Mfhi;
    assign writehl = Mthi | Mtlo;
    assign jump1 = Jalr | Jr;
    assign jump2 = J | Jal;
    assign branch = Beq | Bgtz | Blez | Bgez | Bltz | Bne;
    assign load = Lw | Lb | Lbu | Lh | Lhu;
    assign store = Sw | Sb | Sh;

    // assign all control signals
    assign regDst = (calr | Jalr | readhl) ? 2'b01 : (Jal) ? 2'b10 : 2'b00;
    assign regWrite = calr | cali | readhl | Jal | Jalr | load;
    assign memWrite = store;
    assign aluSrc = cali | store | load;
    assign memRead = load;
    assign extOp = (Xori | Andi | Ori) ? 2'b00 : 2'b01;
    assign memToR = (load) ? 3'b001 : (Jal | Jalr) ? 3'b010 : (Mfhi | Mflo) ?
3'b011 : 3'b000;

    assign branchType = Beq ? 3'b001 : Bgtz ? 3'b010 : Blez ? 3'b011 : Bgez ?
3'b100 : Bltz ? 3'b101 : Bne ? 3'b110 : 3'b000;
    assign jumpOp = branch ? 3'b001 : jump2 ? 3'b010 : jump1 ? 3'b011 :
3'b000;
    assign aluOp =  (Addu||Add||Addi||Addiu||load||store)?4'b0000:
      (Subu|Sub)?4'b0001:
```

```
      (And|Andi)?4'b0010:
                          (Ori|Or)?4'b0011:
                          (Xor|Xori)?4'b0100:
                (Nor)?4'b0101:
                          (Sll|Sllv)?4'b0110:
                          (Srl|Srlv)?4'b0111:
                          (Sra|Srav)?4'b1000:
              (Sltu|Sltiu)?4'b1001:
                          (Slt|Slti)?4'b1010:
                          (Lui)?4'b1011:
                          4'b0000;
    assign isShamt = Srl | Sra | Sll;
    assign loadOp = Lbu ? 3'b001 : Lb ? 3'b010 : Lhu ? 3'b011 : Lh ? 3'b100 :
  3'b000;
    assign storeOp = Sh ? 2'b01 : Sb ? 2'b10 : 2'b00;
    assign start = calmudi;
    assign mudiOp = Mult ? 3'b000 : Multu ? 3'b001 : Div ? 3'b010 : Divu ?
  3'b011 : 3'b000;
    assign hiRead = Mfhi;
    assign loRead = Mflo;
    assign hiWrite = Mthi;
    assign loWrite = Mtlo;
  endmodule
```

1.3 冒险单元设计

1.3.1 总体思路

需要实现两个核心模块，**Haz_Decoder（冒险译码器）以及 Haz_Processor （冒险处理器）。**
冒险译码器负责解析各个阶段指令的信息，**本设计采用分布式译码**，传递给冒险处理器来生成暂停和转发逻辑的信号。

> 解析的信息是：某个具体的指令属于**1.1.3分类中的哪一种**

冒险处理器负责生成暂停和转发逻辑的信号，消除冒险现象。

> 根据策略表，确定哪些组合需要stall(S)信号，哪些组合需要对应的哪些转发数据。
> `forward_MUXs`辅助其完成真正的数据转发。

1.3.2 Haz_Decoder 冒险译码器

可以发现，这部分与Controller模块中对指令分类的部分代码近乎重合。这里就不再展示。这样节省了很多工作量，这也是在1.1.3中提出总分类标准的初衷与好处。

1.3.3 Haz_Processor 冒险处理器

另附上冒险单元处理表：

> 说明：
> **对于某一个指令的某一个数据需求，我们定义需求时间**`Tu`: 这条指令位于 D 级的时候，再经过多少个时钟周期就必须要使用相应的数据。beq的`Tu`为0；sw的`rs_Tu`为1，`rt_Tu`为2。

> **对于某个指令的数据产出，我们定义供给时间Tn**:位于某个流水级的某个指令，它经过多少个时钟周期可以算出结果并且存储到流水级寄存器里。例如，对于 add 指令，当它处于 E 级，此时结果还没有存储到流水级寄存器里，所以此时它的Tn为1。
> **递推公式：下一级的Tn' = max{Tn-1 ,0}**

暂停策略表:

D阶段可能需要暂停的指令			E阶段当前指令				M阶段当前指令
译码器信息	源寄存器	Tu	Tn				
			calr-rd	cali-rt	load-rt		load-rt
			1	1	2		1
calr/calmudi	rs/rt	1			S		
cali	rs	1			S		
load	rs	1			S		
store	rs	1			S		
store	rt	2					
branch	rs/rt	0	S	S	S		S
jump1	rs	0	S	S	S		S
writehl	rs	1			S		
allmudi	#	#	when busy S	when busy S	when busy S		when busy S
allmudi	#	#	when calmudi S	when calmudi S	when calmudi S		

转发策略表:

转发目的地阶段	源寄存器	译码器信息	对应多路选择器名	对应选择信号名	默认输出	转发源阶段										
						E		M				W				
						readhl-rd	jal、jalr-rd	calr-rd	cali-rt	jal、jalr-rd	readhl-rd	calr-rd	cali-rt	load-rt	jal、jalr-rd	readhl-rd
D	rs	calr	Drs_fsel	DrsSel	RD1	hiloData_E	pcPlus8_E	aluRes_M	aluRes_M	pcPlus8_M	hiloData_M	WD	WD	WD	pcPlus8_W	WD
		cali														
		branch														
		load														
		store														
		jump1														
	rt	calr	Drt_fsel	DrtSel	RD2	hiloData_E	pcPlus8_E	aluRes_M	aluRes_M	pcPlus8_M	hiloData_M	WD	WD	WD	pcPlus8_W	WD
		store														
		branch														
E	rs	calr	Ers_fsel	ErsSel	rsData_E			aluRes_M	aluRes_M	pcPlus8_M	hiloData_M	WD	WD	WD	pcPlus8_W	WD
		cali														
		load														
		store														
	rt	calr	Ert_fsel	ErtSel	rtData_E			aluRes_M	aluRes_M	pcPlus8_M	hiloData_M	WD	WD	WD	pcPlus8_W	WD
		store														
M	rt	store	Mrt_fsel	MrtSel	rtData_M							WD	WD	WD	pcPlus8_W	

具体设计如下:

```
`timescale 1ns / 1ps
`include "Define.v"
module Haz_Processor (
    input  [31:0] D_nInstr,
    input  [31:0] E_nInstr,
    input  [31:0] M_nInstr,
    input  [31:0] W_nInstr,
    input         busy,
    // stall logic outputs
    output        enableIFU,
    output        enableD,
    output        clearE,
    // forward logic outputs
    output [ 2:0] DrsSel,
    output [ 2:0] DrtSel,
    output [ 2:0] ErsSel,
    output [ 2:0] ErtSel,
    output [ 2:0] MrtSel
```

```
);
  // register Address information
  wire [4:0] rsD, rsE, rtD, rtE, rsM, rtM, rdM, rsW, rtW, rdW, rdE;
  assign rsD = D_nInstr[`rs];
  assign rtD = D_nInstr[`rt];
  assign rsE = E_nInstr[`rs];
  assign rdE = E_nInstr[`rd];
  assign rtE = E_nInstr[`rt];
  assign rsM = M_nInstr[`rs];
  assign rtM = M_nInstr[`rt];
  assign rdM = M_nInstr[`rd];
  assign rsW = W_nInstr[`rs];
  assign rtW = W_nInstr[`rt];
  assign rdW = W_nInstr[`rd];

  // Decoders information
  // D stage
  wire branchD, calrD, caliD, loadD, storeD, jrD, jalD, jalrD, allmudiD,
calmudiD, readhlD, writehlD;
  Haz_Decoder D_Decoder (
      .nInstr (D_nInstr),
      .cali   (caliD),
      .calr   (calrD),
      .calmudi(calmudiD),
      .store  (storeD),
      .load   (loadD),
      .branch (branchD),
      .writehl(writehlD),
      .readhl (readhlD),
      // additional
      .allmudi(allmudiD),
      .Jr     (jrD),
      .Jalr   (jalrD),
      .Jal    (jalD)
  );
  // E stage
  wire branchE, calrE, caliE, loadE, storeE, jrE, jalE, jalrE, allmudiE,
calmudiE, readhlE;
  Haz_Decoder E_Decoder (
      .nInstr (E_nInstr),
      .cali   (caliE),
      .calr   (calrE),
      .calmudi(calmudiE),
      .store  (storeE),
      .load   (loadE),
      .branch (branchE),
      .readhl (readhlE),
      // additional
      .allmudi(allmudiE),
      .Jr     (jrE),
      .Jalr   (jalrE),
      .Jal    (jalE)
  );
  // M stage
```

```
  wire branchM, calrM, caliM, loadM, storeM, jrM, jalM, jalrM, allmudiM,
calmudiM, readhlM;
  Haz_Decoder M_Decoder (
      .nInstr (M_nInstr),
      .cali   (caliM),
      .calr   (calrM),
      .calmudi(calmudiM),
      .store  (storeM),
      .load   (loadM),
      .branch (branchM),
      .readhl (readhlM),
      // additional
      .allmudi(allmudiM),
      .Jr     (jrM),
      .Jalr   (jalrM),
      .Jal    (jalM)
  );
  // W stage
  wire branchW, calrW, caliW, loadW, storeW, jrW, jalW, jalrW, allmudiW,
calmudiW, readhlW;
  Haz_Decoder W_Decoder (
      .nInstr (W_nInstr),
      .cali   (caliW),
      .calr   (calrW),
      .calmudi(calmudiW),
      .store  (storeW),
      .load   (loadW),
      .branch (branchW),
      .readhl (readhlW),
      // additional
      .allmudi(allmudiW),
      .Jr     (jrW),
      .Jalr   (jalrW),
      .Jal    (jalW)
  );

  // stall logic
  wire stall_rs;
  wire stall_rt;
  wire stall_mudi;
  wire stall;
  assign stall_rt = (branchD && loadE && rtE == rtD && rtD) || (branchD &&
caliE && rtE == rtD && rtD) || (branchD && calrE && rdE == rtD && rtD) ||
(branchD && loadM && rtM == rtD && rtD) || (calrD && loadE && rtD == rtE &&
rtE) || (calmudiD && loadE && rtD == rtE && rtD);
  assign stall_rs = (branchD && loadE && rtE == rsD && rsD) ||
                        (branchD && caliE && rtE == rsD && rsD) ||
                        (branchD && calrE && rdE == rsD && rsD) ||
                        (branchD && loadM && rtM == rsD && rsD) ||
                        (calrD && loadE && rsD == rtE && rsD) ||
                        (caliD && loadE && rsD == rtE && rsD) ||
                        (loadD && loadE && rsD == rtE && rsD) ||
                        (storeD && loadE && rsD == rtE && rsD) ||
                        (calmudiD && loadE && rsD==rtE && rsD) ||
```

```
                          (jrD && loadE && rtE == rsD && rsD) ||
                          (jrD && caliE && rtE == rsD && rsD) ||
                          (jrD && calrE && rdE == rsD && rsD) ||
                          (jrD && loadM && rtM == rsD && rsD)||
                          ((jalrD || writehlD) && loadE && rtE == rsD &&
rsD)||
                          (jalrD && caliE && rtE==rsD && rsD)||
                          (jalrD && calrE && rdE==rsD && rsD)||
                          (jalrD && loadM && rtM==rsD && rsD);
  assign stall_mudi = (allmudiD && busy) || (allmudiD && calmudiE);
  assign stall = stall_rs || stall_rt || stall_mudi;
  assign enableIFU = ~stall;
  assign enableD = ~stall;
  assign clearE = stall;

  // forward logic  default:0
  assign DrsSel = (rsD == rdE && readhlE && rsD) ? 1 : (rsD == 5'd31 &&
jalE && rsD || rsD == rdE && jalrE && rsD) ? 2 : (rsD == rtM &&
(caliM||loadM) && rsD) || (rsD == rdM && calrM && rsD) ? 3 : (rsD == 5'd31
&& jalM && rsD || rsD == rdM&& jalrM && rsD) ? 4 : (rsD==rdM && readhlM &&
rsD)? 5 :(rsD == rtW && (loadW || caliW) && rsD) || (rsD == rdW && calrW &&
rsD) || (rsD==rdW && readhlW && rsD)? 6 : (rsD == 31 && jalW && rsD||
rsD==rdW && jalrW && rsD) ? 7 : 0;
  assign DrtSel = (rtD == rdE && readhlE && rtD) ? 1 : (rtD == 5'd31 &&
jalE && rtD || rtD == rdE && jalrE && rtD) ? 2 : (rtD == rtM &&
(caliM||loadM) && rtD) || (rtD == rdM && calrM && rtD) ? 3 : (rtD == 5'd31
&& jalM && rtD || rtD == rdM&& jalrM && rtD) ? 4 : (rtD==rdM && readhlM &&
rtD)? 5 :(rtD == rtW && (loadW || caliW) && rtD) || (rtD == rdW && calrW &&
rtD) || (rtD==rdW && readhlW && rtD)? 6 : (rtD == 31 && jalW && rtD||
rtD==rdW && jalrW && rtD) ? 7 : 0;
  assign ErsSel = (rsE == rtM && (caliM || loadM) && rsE) || (rsE == rdM &&
calrM && rsE) ? 1 : (rsE == 31 && jalM && rsE || rsE == rdM && jalrM &&
rsE) ? 2 : (rsE == rdM && readhlM && rsE ) ? 3 : (rsE == rtW && (loadW ||
caliW) && rsE) || (rsE == rdW && calrW && rsE) || (rsE == rdW && readhlW &&
rsE) ? 4 : (rsE == 31 && jalW && rsE||rsE == rdW && jalrW && rsE) ? 5 : 0;
  assign ErtSel = (rtE == rtM && (caliM || loadM) && rtE) || (rtE == rdM &&
calrM && rtE) ? 1 : (rtE == 31 && jalM && rtE || rtE == rdM && jalrM &&
rtE) ? 2 : (rtE == rdM && readhlM && rtE ) ? 3 : (rtE == rtW && (loadW ||
caliW) && rtE) || (rtE == rdW && calrW && rtE) || (rtE == rdW && readhlW &&
rtE) ? 4 : (rtE == 31 && jalW && rtE||rtE == rdW && jalrW && rtE) ? 5 : 0;
  assign MrtSel = (rtM == rtW && (loadW || caliW) && rtM) || (rtM == rdW &&
calrW && rtM) || (rtM == rdW && readhlW && rtM) ? 1 : (rtM == 31 && jalW &&
rtM || rtM == rdW && jalrW && rtM) ? 2 : 0;

endmodule
```

Part 2 测试方案

基本方法:

- 构造一段MIPS程序，输入Mars中运行；
- 利用Mars导出机器码，形成code.txt文件，用来载入自己的cpu的IM中，仿真运行；

- 比对两者按统一格式输出的内容。即如下两种格式：

```
//grf
$display("@%h: $%d <= %h", WPC, Waddr, WData);
//dm
$display("@%h: *%h <= %h", pc, addr, din);
```

> 注意:
> 需要用到魔改版的Mars来获取标准的输出
> 这里统一规定：对于0号寄存器的写入不进行输出

以下谈谈测试数据应该如何生成。

2.1 针对单指令功能的测试

2.1.1 构造策略

这部分实现比较简单，可以用高级语言写一段程序，通过一些循环语句连续构造大量单种指令的序列。**对于branch型和jump型指令，可以单独划出跳转区进行构造，以防出现死循环。**

2.1.2 示例代码

```
sub $27, $27, $15
sub $28, $28, $15
sub $29, $29, $15
sub $30, $30, $15
sub $31, $31, $15
ori $5, $0, 4
sw $0, 0($5)
sw $1, 4($5)
sw $2, 8($5)
sw $3, 12($5)
sw $4, 16($5)
# there can be more like this...
```

2.2 针对暂停和转发策略的测试

2.2.1 构造策略

根据自己的策略表，明确总共有哪些组合会出现冒险的情况。列出所有的这些情况，逐个构造相应的测试程序，如下所示（下面没有完全列举）。

2.2.2 示例代码

```
#beq cali
ori $1 $0 12
```

```
beq $1 $0 next2
ori $4 $0 1234
next2:addu $1 $1 $1

#beq load
ori $1 $0 12
sw $1 0($0)
lw $2 0($0)
beq $2 $0 next3
ori $4 $0 1234
next3:addu $1 $1 $1

#cal_r load
ori $1 $0 12
sw $1 0($0)
lw $2 0($0)
subu $3 $2 $1
```

2.3 综合测试

利用**1000行左右的测试程序**全面地测试所有的指令。在P4和P5文档中，已经各提供了一份详尽的代码，这里由于篇幅限制不再提供。**该设计经过了10份此等规模测试程序的测试。**

> 可以前往下一部分查看综合测试的部分性能指标。

综合测试分析

对于综合测试中2份千行级别代码，我使用了教程中的`Kernel - Pipeline`工具进行分析，大致结果如下：

1. 分析结果汇总

- test1

```
standard pipeline-cycle: 1923
    slow pipeline-cycle: 2757
   accepted cycle range: [1756, 2506]
```

有效转发数：245
有效暂停数：225

- test2

```
standard pipeline-cycle: 1500
    slow pipeline-cycle: 2222
   accepted cycle range: [1355, 2005]
```

有效转发数：187
有效暂停数：226

2. 本CPU仿真运行结果

- test1:根据输出的`$time`值进行估计，周期数约为1934，性能良好可以接受。
- test2:根据输出的`$time`值进行估计，周期数约为1509，性能良好可以接受。

2.3 结论

经对比内容一致，经分析性能良好，本测试通过，说明CPU的基本功能实现正确。

Part 3 思考题解答

- **为什么需要有单独的乘除法部件而不是整合进 ALU？为何需要有独立的 HI、LO 寄存器？**

  它是一个时序逻辑电路，与ALU的特性并不兼容，这样独立出来遵循了"高内聚，低耦合"的设计思想。而有独立的HI和LO，操作会更加灵活，便于运算与存取。

- **真实的流水线 CPU 是如何使用实现乘除法的？请查阅相关资料进行简单说明。**

  乘法实现原理：利用移位运算进行二进制乘法。*每次对两因数中的一个的第一位进行判断，一共判断32位共循环32次，若该位为1，则将积加上另一因数，若为0，则不用加。*每次循环结束将用于判断的因数右移一位，用于和积相加的因数左移一位，循环完成后，积即为两因数的积。
  除法实现原理：同样用移位运算。*先将被除数前32位拼接上32位的0，一共判断32位循环32次，每次比较被除数前32位与除数相比，若大于除数，则将被除数前32位减去除数，并将整个64位的被除数加上1，若小于除数，则什么也不做，每次循环完成后将被除数左移一位。*循环完成后，被除数前32位为**余数**，后32位为**商。**

- **请结合自己的实现分析，你是如何处理 Busy 信号带来的周期阻塞的。**

  当D级指令为任意一个mudi类型指令且busy信号有效时，全部阻塞，详见上面的策略表。

- **请问采用字节使能信号的方式处理写指令有什么好处？（提示：从清晰性、统一性等角度考虑）**

  清晰性：相较于单一的WE使能信号来说，字节使能让读者可以清晰看出哪一位字节是需要写入的，更加直观明了。
  统一性：统一了sw,sb,sh以及其他不写入内存四种情况的使能信号，只需要更改对应的字节使能即可实现。

- **请思考，我们在按字节读和按字节写时，实际从 DM 获得的数据和向 DM 写入的数据是否是一字节？在什么情况下我们按字节读和按字节写的效率会高于按字读和按字写呢？**

  不为一字节，仍为按字读取。
  如果有这样一种指令，它会读第一个字的最后一个字节的高四位和第二个字的第一个字节低四位，拼成一个字节，那么此时按字读需要读两次，而按字节读只需一次，此时效率会提高。

- **为了对抗复杂性你采取了哪些抽象和规范手段？这些手段在译码和处理数据冲突的时候有什么样的特点与帮助？**

  对指令进行归类：将例如add,sub指令归为calr；addi这种立即数指令归为cali等。详见1.1.3的总分类标准。这样在考虑阻塞与转发时可以统一考虑，也是提高了统一性。

更改控制器架构：改为**控制信号驱动型**，更加简明易懂，避免代码冗杂。

- **在本实验中你遇到了哪些不同指令类型组合产生的冲突？你又是如何解决的？相应的测试样例是什么样的？**

  详见冒险单元和测试方案相应节。

- **如果你是手动构造的样例，请说明构造策略，说明你的测试程序如何保证覆盖了所有需要测试的情况；如果你是完全随机生成的测试样例，请思考完全随机的测试程序有何不足之处；如果你在生成测试样例时采用了特殊的策略，比如构造连续数据冒险序列，请你描述一下你使用的策略如何结合了随机性达到强测的效果。**

  详见测试方案一节。