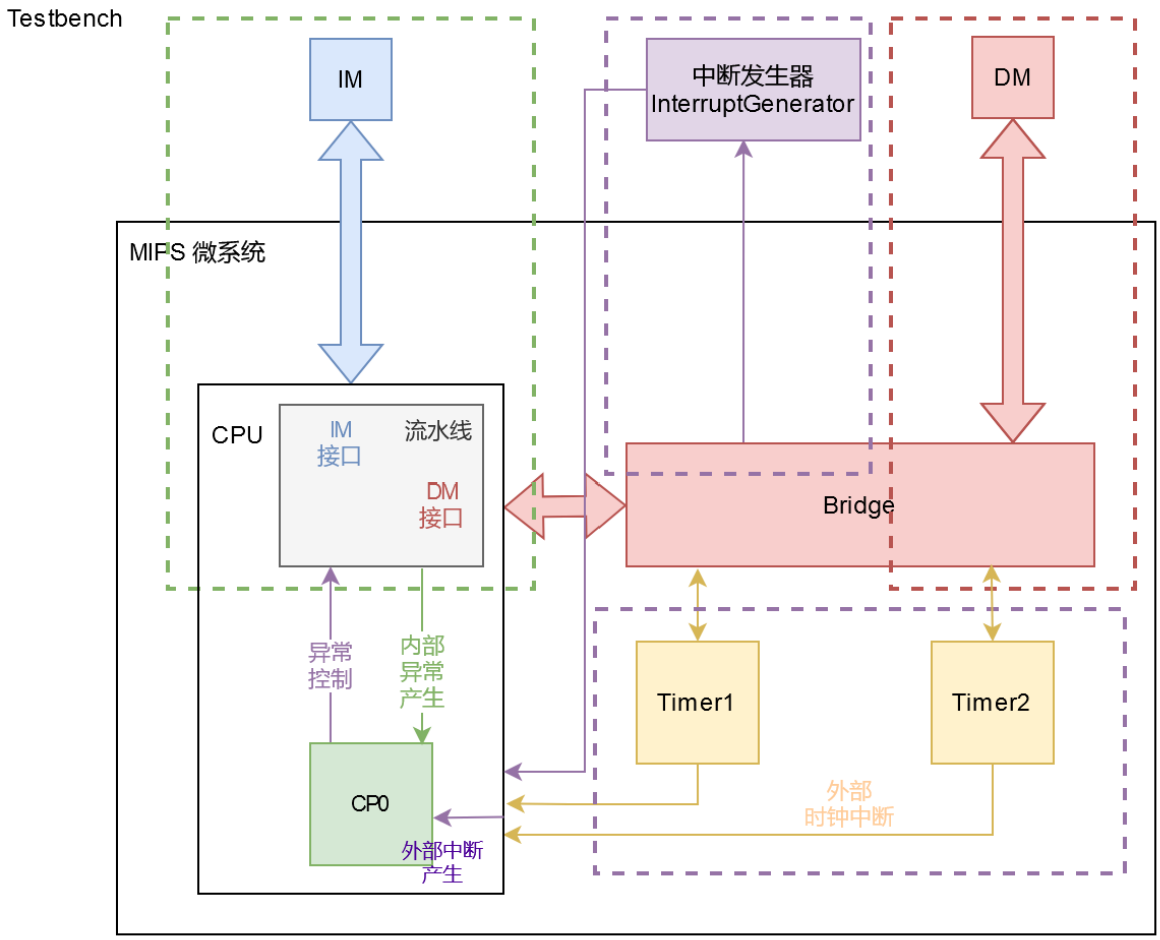
P7设计文档

概述

本次通过Verilog实现MIPS微系统，并支持多种中断、异常处理功能。总体结构如下：



计时器

计时器结构使用教程给出的Timer.v：

控制寄存器说明

ctrl	名称	功能
ctrl[0]	计数使能	当计数使能为1时，倒计时开始运行
ctrl[2:1]	模式	当 ctrl[2:1]==2'b0 时为模式0，否则为模式1
ctrl[3]	中断使能	只有当中断使能为1时，倒计时到0会产生中断信号

异常处理

在IFU、D_controller、ALU等原有模块添加异常判断并将异常信号流水，在M级新增判断模块MCHECK，用于大部分存取异常判断：

MCHECK

```
`include "InStrType.v"
`include "ExcType.v"
`timescale 1ns / 1ps

module MCHECK(
    input [9:0] M_inStrType,
    input [31:0] M_aluResult,
    output [4:0] M_excCode // 异常码用5位记
);

    reg [4:0] temp_excCode;

    wire islw,islh,islb,issw,issh,issb;
    wire hitTC,hitCount,isaddrOv; // 是否以TC为存取地址、是否以count为存取地址
    wire [1:0] byte;
    assign byte = M_aluResult[1:0];

    assign M_excCode = temp_excCode;

    assign islw = (M_inStrType == `lw);
    assign islh = (M_inStrType == `lh);
    assign islb = (M_inStrType == `lb);
    assign issw = (M_inStrType == `sw);
    assign issh = (M_inStrType == `sh);
    assign issb = (M_inStrType == `sb);

    assign isaddrOv = !(M_aluResult < 32'h0000_3000 ||
                        (M_aluResult <= 32'h0000_7f0b && M_aluResult >=
32'h0000_7f00) ||
                        (M_aluResult <= 32'h0000_7f1b && M_aluResult >=
32'h0000_7f10) ||
                        (M_aluResult <= 32'h0000_7f23 && M_aluResult >=
32'h0000_7f20));
    assign hitTC = (M_aluResult[15:4] == 12'h7f0 || M_aluResult[15:4] ==
12'h7f1);
    assign hitCount = (hitTC && (M_aluResult[3:0] == 4'h8));

    always @(*) begin
        if ((islw && byte != 2'b0) || // lw未对齐
            (islh && (byte == 2'b01 || byte == 2'b11)) || // lh未对齐
            ((islb || islh) && hitTC) || // 残字存TC
            ((islw || islh || islb) && isaddrOv)) // 存数据地址越界
            begin
            temp_excCode = 5'h4;
        end
        else if ((issw && byte != 2'b0) ||
                (issh && (byte == 2'b01 || byte == 2'b11)) ||
                ((issb || issh) && hitTC) ||
```

```
                ((issw || issh || issb) && isaddrOv) ||
                (issw && hitCount)) begin // 想写TC的count寄存器
            temp_excCode = 5'h5;
        end
        else begin
            temp_excCode = 5'h0;
        end
    end

endmodule
```

中断处理

将所有外部中断信号合并成六位中断请求信号传入新增M级模块CP0中，与异常一起进行判断：

CP0

```
`include "InStrType.v"
`include "ExcType.v"
`timescale 1ns / 1ps
`define intAccept SR[15:10] // 是否允许外部中断
`define isInExc SR[1] // 是否正处于异常处理中
`define wholeExcEn SR[0] // 全局外部中断使能
`define isBD cause[31] // (是否为延迟槽指令)是时epc指向受害指令前一条
`define intReqReg cause[15:10] // 外部中断请求寄存器
`define excCodeReg cause[6:2] // 异常的编码寄存器

module CP0 (
    input clk,
    input reset,
    input M_writeCP0_EN,
    input BD_IN,  // 是否为延迟槽指令
    input [4:0] excCode_IN,  // 异常编码
    input [4:0] CP0_regNUM,  // 访问的CP0寄存器地址
    input [5:0] intReq,  // 6个外部中断请求
    input [9:0] M_inStrType,  // M级指令类型
    input [31:0] CP0_IN,
    input [31:0] VPC,  // 受害PC
    output [31:0] CP0_OUT,
    output [31:0] EPC_OUT,  // 传递EPC
    output isIntExc  // 是否为中断或异常
);

    reg [31:0] SR, cause, EPC;  // 三个cp0的寄存器,编号分别为12/13/14
    wire isInt, isExc;  // 是否为外部中断，是否为内部异常

    assign isInt = (|(intReq & `intAccept)) && `wholeExcEn && !`isInExc; // 有外部中断请求(isInt随外部中断请求的改变即时改变)
                                                                         // 全局中断使能有效
                                                                         // 未在异常处理中
    assign isExc = (excCode_IN != 5'h0 && !`isInExc); // 随输入端口的异常码的改变即时改变
```

```
    assign CP0_OUT = (CP0_regNUM == 5'd12) ? SR :
                     (CP0_regNUM == 5'd13) ? cause :
                     (CP0_regNUM == 5'd14) ? EPC : 32'h0;
    assign EPC_OUT = (M_writeCP0_EN && CP0_regNUM == 5'd14) ? CP0_IN : EPC; //
epc值的内部转发
    assign isIntExc = isInt | isExc;  // 外部中断请求和内部异常都会引发中断,
                                      // 当异常指令刚进入到M级寄存器或者外部
                                      // 中断刚发出的时候CP0的输出端已经告诉
                                      // cpu下个上升沿开始进行中断处理, 在中
                                      // 断处理的第一个上升沿到来之前,中断信号便已准备好

    always @(posedge clk) begin
        if (reset) begin
            SR <= 0;
            cause <= 0;
            EPC <= 0;
        end else begin
            `intReqReg <= intReq;
            if (isInt) begin
                `isInExc <= 1'b1;
                EPC <= BD_IN ? (VPC - 4) : VPC;
                `isBD <= BD_IN;
                `excCodeReg <= 5'h0;
            end else if (isExc) begin
                `isInExc <= 1'b1;
                EPC <= BD_IN ? (VPC - 4) : VPC;
                `isBD <= BD_IN;
                `excCodeReg <= excCode_IN;
            end

            if (M_inStrType == `eret) begin
                `isInExc <= 1'b0;
            end

            if (M_writeCP0_EN) begin
                case (CP0_regNUM)
                    5'd12: begin
                        `intAccept <= CP0_IN[15:10];
                        `isInExc <= CP0_IN[1];
                        `wholeExcEn <= CP0_IN[0];
                    end
                    5'd13: begin
                        // 保证不写cause
                    end
                    5'd14: begin
                        EPC = CP0_IN;
                    end
                endcase
            end
        end
    end

endmodule
```

测试方案

自行构造中断、异常测试数据进行测试：

中断测试

```
.text
  label_3000: ori $t0, $0, 0x1001
  #label_3000: ori $t0, $0, 0x0001
  #label_3000: ori $t0, $0, 0x1000
  label_3004: mtc0 $t0, $12
  label_3008: addi $t0, $t0, 1
  label_300c: addi $t0, $t0, 1

  #label_3010_mars: lw $t0, -1($0) # MARS
  label_3010: addi $t0, $t0, 1
  label_3014: addi $t0, $t0, 1
  label_3018: addi $t0, $t0, 1
  label_301c: addi $t0, $t0, 1

.ktext 0x4180
  ori $k0, $0, 0x3000
  addi $k0, $k0, -4
  sw $t0, 0($k0)
  mfc0 $t0, $12
  mfc0 $t0, $13
  andi $t0, $t0, 0x1000

  beq $t0, $0, endif_interrupt
  nop
  if_interrupt:
    ori $t0, $0, 0x7F20
    sw $t0, 0($t0)
  endif_interrupt:

  lw $t0, 0($k0)
  eret
  ori $k0, $0, 0xffff
```

异常测试

```
.text
  # exception_example
  ori $t0, $0, 0x0001
  mtc0 $t0, $12
  nop
  nop
  nop
  sw $0, 0x7f20($0)

  ori $s0, $0, 0x0000

  # AdEL
  ori $t0, $0, 0x3001
```

```
jal adel_pcnotaligned
nop
adel_pcnotaligned: addi $s1, $ra, 8 # $ra
jr $t0
nop

ori $t0, $0, 0x2ffc
jal adel_pcoutofrange
nop
adel_pcoutofrange: addi $s1, $ra, 8 # $ra
jr $t0
nop

lw $t0, 2($0)
lh $t0, 1($0)
ori $t0, $0, 0x7f00
lh $t1, 0($t0)
lb $t1, 0($t0)
lui $t0, 0x7fff
ori $t0, $t0, 0xffff
lw $t1, 32767($t0)
ori $t0, $0, 0x7f0c
lw $t1, 0($t0)
ori $t0, $0, 0x7f0e
lh $t1, 0($t0)
ori $t0, $0, 0x7f24
lb $t1, 0($t0)

# AdES
sw $t0, 2($0)
sh $t0, 1($0)
ori $t0, $0, 0x7f00
sh $t1, 0($t0)
sb $t1, 0($t0)
lui $t0, 0x7fff
ori $t0, $t0, 0xffff
sw $t1, 32767($t0)
ori $t0, $0, 0x7f08 # count
sw $t1, 0($t0)
ori $t0, $0, 0x7f0c
sw $t1, 0($t0)
ori $t0, $0, 0x7f0e
sh $t1, 0($t0)
ori $t0, $0, 0x7f24
sb $t1, 0($t0)

# Syscall
syscall
syscall

# RI
addu $t0, $t0, $t0
ori $t0, $0, 0x0001
ori $t0, $0, 0x0002
ori $t0, $0, 0x0003
ori $t0, $0, 0x0004
```

```
  # Ov
  lui $t0, 0x7fff
  add $t1, $t0, $t0
  ori $t0, $0, 0x0001
  ori $t0, $0, 0x0002
  ori $t0, $0, 0x0003
  ori $t0, $0, 0x0004



  loop:
  beq $0, $0, loop
  nop

#################### .ktext ####################
.ktext 0x4180
  # $t0: SR
  # $t1: Cause_ExcCode
  # $t2: EPC
  # $t3: temp

  ori $k0, $0, 0x3000
  addi $k0, $k0, -16
  sw $t3, 12($k0)
  sw $t2, 8($k0)
  sw $t1, 4($k0)
  sw $t0, 0($k0)
  addi $s0, $s0, 1

  mfc0 $t0, $12
  mfc0 $t1, $13
  mfc0 $t2, $14

  andi $t1, $t1, 0x00FC
  switch_exccode:
    ori $t3, $0, 0x0000 # 0 (Int) << 2
    beq $t1, $t3, case_int
    nop
    ori $t3, $0, 0x0010 # 4 (AdEL) << 2
    beq $t1, $t3, case_adel
    nop
    ori $t3, $0, 0x0014 # 5 (AdES) << 2
    beq $t1, $t3, case_ades
    nop
    ori $t3, $0, 0x0020 # 8 (Syscall) << 2
    beq $t1, $t3, case_syscall
    nop
    ori $t3, $0, 0x0028 # 10 (RI) << 2
    beq $t1, $t3, case_ri
    nop
    ori $t3, $0, 0x0030 # 12 (Ov) << 2
    beq $t1, $t3, case_ov
    nop

    case_int:
```

```
      ori $k0, $0, 0xff98
      beq $0, $0, endswitch_exccode
      nop
    case_adel:
      andi $t3, $t2, 0x0003
      bne $t3, $0, if_pcerror
      nop
      ori $t3, $0, 0x3000
      sltu $t3, $t2, $t3 # PC < 0x3000
      bne $t3, $0, if_pcerror
      nop
      ori $t3, $0, 0x6fff
      sltu $t3, $t3, $t2 # PC > 0x6fff
      bne $t3, $0, if_pcerror
      nop

      beq $0, $0, else_pcerror
      nop

      if_pcerror:
        mtc0 $s1, $14
        beq $0, $0, endif_pcerror
        nop
      else_pcerror:
        addi $t2, $t2, 4
        mtc0 $t2, $14
        beq $0, $0, endif_pcerror
        nop
      endif_pcerror:

      beq $0, $0, endswitch_exccode
      nop
    case_ades:
    case_syscall:
    case_ri:
    case_ov:
      addi $t2, $t2, 4
      mtc0 $t2, $14
      beq $0, $0, endswitch_exccode
      nop
  endswitch_exccode:

  lw $t0, 0($k0)
  lw $t1, 4($k0)
  lw $t2, 8($k0)
  lw $t3, 12($k0)
  eret
  ori $k0, $0, 0xff98
```

思考题

1.请查阅相关资料，说明鼠标和键盘的输入信号是如何被 CPU 知晓的?

鼠标和键盘相当于中断发生器，当鼠标和键盘产生输入信号时，其值会被写入相关寄存器，中断发生器将会给cpu一个中断信号，cpu收到中断信号后进入到异常处理程序中，在异常处理程序中获得键鼠的信息。

2.请思考为什么我们的 CPU 处理中断异常必须是已经指定好的地址？如果你的 CPU 支持用户自定义入口地址，即处理中断异常的程序由用户提供，其还能提供我们所希望的功能吗？如果可以，请说明这样可能会出现什么问题？否则举例说明。（假设用户提供的中断处理程序合法）

一种思路是，假如我们的cpu让用户定义入口地址，则可能需要我们新开辟输入端口令cpu输入地址，当用户键入地址时就需要中断来处理，但是此时还用户输入的地址还并未被知晓因此便无法得知从何处进入中断处理程序。而且我们指定入口地址的话，将无需软件工程师再费心从何处进入异常，而只需要编写自己所需要的异常处理程序即可，因此，这会大大降低软件工程师的压力。

3.为何与外设通信需要 Bridge?

因为如果不用桥而直接让cpu与外界设备进行通信，则cpu的端口就会变得很多（原本需要在桥上开的端口都将开到cpu上）这样会大大增加cpu的复杂度，而使用bridge就会让端口变得十分简洁。

4.请阅读官方提供的定时器源代码，阐述两种中断模式的异同，并针对每一种模式绘制状态移图。

模式0和模式1的相同点在于他们都需要“手动”开启计数使能而开始计数而不能自动开始计数，而他们的区别主要体现在：当计时器计时结束时，模式1下的计数器会自动将初值加载到计数器重新开始倒计数；在模式0下的计数器会停止计数并将计数器的计数使能关闭而停止倒计数，当计数使能被设置为1后再次开始计数。

5. 倘若中断信号流入的时候，在检测宏观 PC 的一级如果是一条空泡（你的 CPU 该级所有信息均为空）指令，此时会发生什么问题？在此例基础上请思考：在 P7 中，清空流水线产生的空泡指令应该保留原指令的哪些信息?

这时如果这个空泡是一个延迟槽指令的话，应该向cp0的epc中写入该pc-4，但是由于空泡指令不包含任何信息，这导致会写入到epc一个错误的值，同时不包含isBD这一信息的话，也无法判断该空泡处是否本应是一个延迟槽指令。因此，清空流水线所产生的空泡指令应该保留pc和isBD两个信息。

6. 为什么 `jalr` 指令为什么不能写成 `jalr $31, $31`?

当jalr或jalr的延迟槽异常时需要进入异常处理程序，处理完后需要跳回jalr重新执行，此时$31的值已被改变，使得程序运行出错。

总体电路如下图

```
`include "InStrType.v"
`timescale 1ns / 1ps

module mips (
    input clk,
    input reset,
    input interrupt,  // 外部中断信号
    output [31:0] macroscopic_pc,  // 宏观 PC

    output [31:0] i_inst_addr,  // F级PC
    input  [31:0] i_inst_rdata, // F级对应指令
```

```
    output [31:0] m_data_addr,   // DM写入地址
    input  [31:0] m_data_rdata,  // DM到空壳数据
    output [31:0] m_data_wdata,  // DM写入数据
    output [ 3:0] m_data_byteen, // DM写使能

    output [31:0] m_int_addr,   // 中断发生器待写入地址
    output [ 3:0] m_int_byteen, // 中断发生器字节使能信号

    output [31:0] m_inst_addr,  // M级PC

    output w_grf_we,  // GRF写使能
    output [4:0] w_grf_addr,  // GRF写寄存器编号
    output [31:0] w_grf_wdata,  // GRF写入数据

    output [31:0] w_inst_addr  // W级PC
);

    wire [31:0] PrAddr, PrAddr_out, PrWD, PrWD_out, TC0_data, TC1_data, PrRD;
    wire [31:0] single_pc;
    wire [ 5:0] intReq;
    wire [ 3:0] PrByteEn;
    wire TC0_WE, TC1_WE;
    wire TC0_req, TC1_req;

    assign m_data_wdata = PrWD_out;
    assign macroscopic_pc = single_pc;
    assign m_int_addr = PrAddr_out;
    assign m_data_addr = PrAddr_out;

    assign intReq = {1'b0, 1'b0, 1'b0, interrupt, TC1_req, TC0_req};

    /*
    module CPU (
        input clk,
        input reset,
        input [5:0] intReq,
        input [31:0] i_inst_rdata,  // F级对应指令
        input [31:0] m_data_rdata,  // DM到空壳数据
        output [31:0] i_inst_addr,  // F级PC
        output [31:0] m_data_addr,  // DM写入地址
        output [31:0] m_data_wdata,  // DM写入数据
        output [3:0] m_data_byteen,  // DM写使能
        output [31:0] m_inst_addr,  // M级PC
        output w_grf_we,  // GRF写使能
        output [4:0] w_grf_addr,  // GRF写寄存器编号
        output [31:0] w_grf_wdata,  // GRF写入数据
        output [31:0] w_inst_addr,  // W级PC
        output [31:0] macro_PC
    );
    */

    CPU cpu (
        .clk(clk),
        .reset(reset),
        .intReq(intReq),
        .i_inst_rdata(i_inst_rdata),
```

```
        .m_data_rdata(PrRD),
        .i_inst_addr(i_inst_addr),
        .m_data_addr(PrAddr),
        .m_data_wdata(PrWD),
        .m_data_byteen(PrByteEn),
        .m_inst_addr(m_inst_addr),
        .w_grf_we(w_grf_we),
        .w_grf_addr(w_grf_addr),
        .w_grf_wdata(w_grf_wdata),
        .w_inst_addr(w_inst_addr),
        .macro_PC(single_pc)
    );

    /*
    module BRIDGE (
        input [31:0] PrAddr,  // processer
        input [31:0] PrWD,  // cpu 要写的数据
        input [3:0] PrByteEn,

        input [31:0] TC0_RD,
        input [31:0] TC1_RD,
        input [31:0] DM_RD,
        input [31:0] Int_RD,  // 恒为0

        output [31:0] PrRD,  // cpu 要读的数据

        output [3:0] IntByteEn_OUT,
        output [3:0] DMByteEn_OUT,

        output TC0_WE,
        output TC1_WE,

        output [31:0] PrWD_OUT,
        output [31:0] PrAddr_OUT
    );
    */

    BRIDGE bridge (
        .PrAddr(PrAddr),
        .PrWD(PrWD),
        .PrByteEn(PrByteEn),
        .TC0_RD(TC0_data),
        .TC1_RD(TC1_data),
        .DM_RD(m_data_rdata),
        .Int_RD(32'h0),
        .PrRD(PrRD),
        .IntByteEn_OUT(m_int_byteen),
        .DMByteEn_OUT(m_data_byteen),
        .TC0_WE(TC0_WE),
        .TC1_WE(TC1_WE),
        .PrWD_OUT(PrWD_out),
        .PrAddr_OUT(PrAddr_out)
    );

    /*
    module TC (
```

```
        input clk,
        input reset,
        input [31:2] Addr,
        input WE,
        input [31:0] Din,
        output [31:0] Dout,
        output IRQ
    );
    */

    TC TC0 (
        .clk(clk),
        .reset(reset),
        .Addr(PrAddr_out[31:2]),
        .WE(TC0_WE),
        .Din(PrWD_out),
        .Dout(TC0_data),
        .IRQ(TC0_req)
    );

    TC TC1 (
        .clk(clk),
        .reset(reset),
        .Addr(PrAddr_out[31:2]),
        .WE(TC1_WE),
        .Din(PrWD_out),
        .Dout(TC1_data),
        .IRQ(TC1_req)
    );

endmodule
```