2021年北航计组P7课下设计报告

一 CPU设计方案简述

（一）总体设计概述

本CPU为Verilog实现的五级流水线MIPS - CPU，支持的指令集包含{LB、LBU、LH、LHU、LW、SB、SH、SW、ADD、ADDU、 SUB、 SUBU、 MULT、 MULTU、 DIV、 DIVU、 SLL、 SRL、 SRA、 SLLV、 SRLV、SRAV、AND、OR、XOR、NOR、ADDI、ADDIU、ANDI、ORI、 XORI、LUI、SLT、 SLTI、SLTIU、SLTU、BEQ、BNE、BLEZ、BGTZ、 BLTZ、BGEZ、J、JAL、JALR、JR、MFHI、MFLO、MTHI、MTLO、ERET、MFC0、MTC0}。为了实现这些功能，CPU主要包含了CP0、Bridge、IFU、GRF、ALU、NPC、EXT、CMP、CU、SU、MDU乘除模块、BE扩展模块，数据扩展模块、流水线寄存器 D_Reg、E_Reg等模块。

（二）关键模块设计

1. GRF

GRF端口定义

端口名	方向	描述
clk	I	时钟信号
reset	I	复位信号， 1 复位，0无效
WE	I	写使能信号，1写入，0不能写入
A1	I	5位地址输入信号，将对应的寄存器数据读出到RD1
A2	I	5位地址输入信号，将对应的寄存器数据读出到RD2
A3	I	5位地址输入信号，将数WD存储到所对应的寄存器
WD3	I	32位输入数据
RD1	O	输出A1地址所对应的寄存器数据
RD2	O	输出A2地址所对应的寄存器数据
pc	I	用于输出$display中的地址指令信息

功能定义

序号	功能名称	描述
1	复位	reset信号有效时，所有寄存器存储的数据清零
2	读数据(附带内部转发)	读A1,A2所对应的寄存器数据到RD1，RD2，当A3=A1或A3=A2时，且写入信号为1时，直接将WD3端的数据输出到RD1或者RD2
3	写数据	当WE有效且时钟上升沿到来时，将WD3存入到A3所对应的寄存器，并且输出对应的地址指令以及相应的寄存器信息

内部转发代码

```
assign RD1 = (A1 == 0)? 32'b0:
             (A1 == A3 && en == 1'b1 )? WD:
                                        grf[A1];
assign RD2 = (A2 == 0)? 32'b0:
             (A2 == A3 && en == 1'b1 )? WD:
                                        grf[A2];
```

2.ALU

ALU 端口简述

端口名	方向	描述
DataA [31:0]	I	数据A
DataB [31:0]	I	数据B
ALUctrl [3:0]	I	运算控制选择信号
result [31:0]	O	输出结果
ALUzero	O	当result结果为0，ALUzero输出为1

ALU 功能介绍

ALUctrl	运算描述
0000	A&B
0001	A|B
0010	A + B
0011	A - B
0100	A⊕B
0101	A<<B
0110	A>>B
0111	A>>>B
1000	~(A|B)
1001	$signed(A)<$signed(B)?
1010	A<B?

3. NPC

端口名	方向	描述
[31:0] D_pc	I	位于D级的当前指令地址
[31:0]F_pc	I	位于F级的下一指令地址
[25:0] imm	I	指令立即数
[31: 0] FW_rs_D	I	当指令为jr时，经过转发后的需转跳的rs寄存器所存储的地址
branch	I	beq指令信号
cmp_result	I	beq指令判断信号
jump	I	jump指令信号
jr	I	jr指令判断信号
[31:0] npc	O	下一指令

功能介绍

1.jr信号为1时，npc为经过转发后的D级 rs寄存器的值。

2.jump信号为1时，npc的值为 D_pc31-28 || index || 00

3 .branch 和 cmp_result信号同时为1时，npc为 D_PC +4+ sign_ext(imm)||00

4.上述情况都不满足时，npc为Fpc+4

4.IFU

端口名	方向	描述
clk	I	时钟信号
reset	I	清零信号
[31:0] npc	I	下一地址
en	I	pc寄存器使能端
F_pc	O	F所需取指的地址

功能介绍

1. 根据复位信号，将F_pc地址信号同步复位至0x0000_3000
2. 根据当前npc的输入，在时钟上升沿，更新F_pc的输出
3. 冻结寄存器，使能端 en = ！ stall，当流水线需要暂停时，寄存器使能端为0，无法写入新的F_pc地址，寄存器被冻结。

5. EXT

端口	方向	描述
[25:0] imm	I	待扩展的立即数
[1:0] EXT	I	扩展控制信号
D_imm	O	扩展后的输出信号

控制实现代码

```
assign D_imm = ( EXT == 2'b00 )? {{16{imm[15]}},imm[15:0]}  :
               ( EXT == 2'b01 )? {{16{1'b0}},imm[15:0]}  :
               ( EXT == 2'b10 )? { imm[15:0],{16{1'b0}} } :
                                                    32'b0 ;
```

6. MDU乘除模块

端口	方向	描述
clk	I	时钟信号
reset	I	清零信号
[31:0] dataA	I	RS输入数据
[31:0] dataB	I	RT输入数据
[3:0] MDUtype	I	由控制器CU输入的乘除相关指令识别信号
busy	O	乘除运算执行时输出busy信号
start	O	乘除运算开始前一周期输出start信号
[31:0] dataout	O	输出HI、LO寄存器数据

乘除模块行为约定如下：

- 当mul、mulu、div、divu指令进入E级，start信号置1持续1周期，自 Start 信号有效后的第 1 个 clock 上升沿开始，乘除部件开始执行运算，同时 Busy 置位为 1。
- 在运算结果保存到 HI 和 LO 后，Busy 位清除为 0。
- 当 Busy 或 Start 信号为 1 时，位于D级的指令mfhi、mflo、mthi、mtlo、mult、multu、div、divu 均被阻塞在D级。
- 数据写入 HI 或 LO，均只需 1 个 cycle。

实现代码

```
module MDU(
    input clk,
    input reset,
    input [31:0] dataA,
    input [31:0] dataB,
    input [3:0] MDUtype,
     output reg  busy,
    output start,
    output [31:0] dataout
    );

     wire mult =  ( MDUtype == 4'd1 )? 1 : 0;
     wire multu = ( MDUtype == 4'd2 )? 1 : 0;
     wire div =   ( MDUtype == 4'd3 )? 1 : 0;
     wire divu =  ( MDUtype == 4'd4 )? 1 : 0;
     wire mfhi =  ( MDUtype == 4'd5 )? 1 : 0;
     wire mflo =  ( MDUtype == 4'd6 )? 1 : 0;
     wire mthi =  ( MDUtype == 4'd7 )? 1 : 0;
     wire mtlo =  ( MDUtype == 4'd8 )? 1 : 0;

     reg [31:0] HI, LO;
     reg [31:0 ] temp_hi, temp_lo;
     reg [3:0] counter ;


     assign start = mult | multu | div |divu ;
```

```
    assign dataout = ( mfhi == 1'b1 ) ? HI :
                     ( mflo == 1'b1 ) ? LO :
                                                32'b0;


    initial begin
       counter <= 4'd0;
        HI <= 32'd0;
        LO <= 32'd0;
    end


    always@( posedge clk ) begin
         if ( reset == 1'b1 ) begin
                 busy <= 1'b0;
                 counter <= 4'd0;
                HI <= 32'd0;
                LO <= 32'd0;
           end
           else begin
                 if ( counter == 4'd0 ) begin
                         if ( mult == 1'b1 ) begin
                                busy <= 1'b1;
                                 counter <= 4'd5;
                                 { temp_hi ,temp_lo } <= $signed(dataA) *
$signed(dataB);
                           end
                           else if ( multu == 1'b1 ) begin
                                busy <= 1'b1;
                                 counter <= 4'd5;
                                 { temp_hi ,temp_lo } <= dataA * dataB ;
                           end
                           else if ( div == 1'b1 ) begin
                               busy <= 1'b1;
                                counter <= 4'd10;
                               temp_lo <= $signed(dataA) / $signed(dataB);
                                temp_hi <= $signed(dataA) % $signed(dataB);
                           end
                           else if ( divu == 1'b1 ) begin
                               busy <= 1'b1;
                                counter <= 4'd10;
                               temp_lo <= dataA / dataB;
                                temp_hi <= dataA % dataB;
                           end
                           else if ( mthi == 1'b1 ) begin
                                HI <= dataA ;
                                 counter <= 4'd0;
                           end
                           else if ( mtlo == 1'b1 ) begin
                                LO <= dataA ;
                                 counter <= 4'd0;
                           end

                   end
                   else if  ( counter == 4'd1 ) begin
                              busy <= 1'b0;
                                 counter <= 4'd0;
                                 HI <= temp_hi;
```

```
                                LO <= temp_lo;
                    end
                    else begin
                           counter <= counter -1 ;
                    end
             end
      end


endmodule
```

7.BE扩展模块

根据存储器外置的要求，我们需要输出字节写使能，以及对应的存储数据，所以需要特定的模块来计算字节写使能以及对原存入数据进行移位，确保写入对齐。

端口	方向	描述
[31:0] indata	I	原需写入数据存储器的数据
[31:0] A	I	需存入的数据寄存器地址
[1:0] s_type	I	store指令的类型
[3:0] BE	O	4位字节使能
[31:0] outdata	O	移位后对齐的数据

实现代码

```
module M_BE(
    input [31:0] indata,
    input [1:0] s_type,
     input [31:0] A,
    output reg [3:0] BE,
    output reg [31:0] outdata

    );

    always@(*) begin
          if ( s_type == 2'b01 ) begin
                 BE = 4'b1111;
                  outdata =  indata;
         end
         else if ( s_type == 2'b10 ) begin
                if ( A[1] == 1'b0 ) begin
                       BE = 4'b0011;
                         outdata = { {16{1'b0}}, indata[15:0]}  ;
                end
                else if ( A[1] == 1'b1 ) begin
                       BE = 4'b1100;
                         outdata = { indata[15:0],{16{1'b0}} }  ;
```

```
                end
        end
        else if ( s_type == 2'b11 ) begin
                if ( A[1:0] == 2'b00 ) begin
                      BE = 4'b0001;
                        outdata = { {24{1'b0}}, indata[7:0] };
                end
                else if ( A[1:0] == 2'b01 ) begin
                      BE = 4'b0010;
                        outdata = { {16{1'b0}}, indata[7:0], {8{1'b0}} };
                end
                else if ( A[1:0] == 2'b10 ) begin
                      BE = 4'b0100;
                        outdata = { {8{1'b0}}, indata[7:0], {16{1'b0}} };
                end
                else if ( A[1:0] == 2'b11 ) begin
                      BE = 4'b1000;
                        outdata = { indata[7:0], {24{1'b0}} };
                end
           end
           else if ( s_type == 2'b00 ) begin
                   BE = 4'b0000;
                     outdata = indata;
           end

     end

endmodule
```

关于字节使能的描述：

引自教程网站

m_data_byteen[3:0]是字节使能信号，其最高位到最低位分别与m_data_wdata的[31:24]、[23:16]、[15:8]及[7:0]对应（即一位对应一个字节）。例如，若m_data_byteen[3]为 1，则m_data_wdata[31:24]会被写入到m_data_addr所指向 word 的[31:24]，依此类推。若m_data_byteen的任意一位为 1，则代表当前需要写入内存，也就是说可以用|m_data_byteen 代替数据存储器的写使能信号。

m_data_byteen[3:0]主要用于支持sb、sh这两条指令。当处理器执行sb、sh指令时，对 EX/MEM 保存的 ALU 计算结果 32 位地址的低两位进行解读，产生相应的m_data_byteen`信号，就可以“通知”Testbench 中的 DM 该写入哪些字节。

8.数据拓展模块

对于lb、lbu、lh、lhu来说，需要额外增加一个数据扩展模块。这个模块把从数据存储器读出的数据做符号或无符号扩展。

端口	方向	描述
[31:0] A	I	访问数据存储器的地址
[31:0] Din	I	访问数据存储器的输出数据
[2:0] Op	I	扩展类型指令信号
[31:0] Dout	O	扩展后的数据

实现代码

```
module M_data_ext(
    input [31:0] A,
    input [31:0] Din,
    input [2:0] Op,
    output reg [31:0] Dout
    );


     always@(*) begin
          if ( Op == 3'b000 ) begin
                 Dout = Din;
            end
            else if ( Op == 3'b001 ) begin
                       if ( A[1:0] == 2'b00 ) begin
                             Dout = { {24{1'b0}}, Din[7:0]};
                        end
                        else if ( A[1:0] == 2'b01 ) begin
                             Dout = { {24{1'b0}}, Din[15:8]};
                        end
                        else if ( A[1:0] == 2'b10 ) begin
                             Dout = { {24{1'b0}}, Din[23:16]};
                        end
                        else if ( A[1:0] == 2'b11 ) begin
                             Dout = { {24{1'b0}}, Din[31:24]};
                        end
            end
            else if ( Op == 3'b010 ) begin
                       if ( A[1:0] == 2'b00 ) begin
                             Dout = { {24{Din[7]}}, Din[7:0]};
                        end
                        else if ( A[1:0] == 2'b01 ) begin
                             Dout = { {24{Din[15]}}, Din[15:8]};
                        end
                        else if ( A[1:0] == 2'b10 ) begin
                             Dout = { {24{Din[23]}}, Din[23:16]};
                        end
                        else if ( A[1:0] == 2'b11 ) begin
                             Dout = { {24{Din[31]}}, Din[31:24]};
                        end
            end
            else if ( Op == 3'b011 ) begin
                      if ( A[1] == 1'b0 ) begin
                              Dout = { {16{1'b0}}, Din[15:0]};
                      end
                         else if (  A[1] == 1'b1) begin
```

```
                            Dout = { {16{1'b0}}, Din[31:16]};
                        end
            end
            else if ( Op == 3'b100 ) begin
                        if ( A[1] == 1'b0 ) begin
                            Dout = { {16{Din[15]}}, Din[15:0]};
                        end
                        else if (  A[1] == 1'b1 ) begin
                            Dout = { {16{Din[31]}}, Din[31:16]};
                        end
            end

     end


endmodule
```

9 .CMP模块

CMP模块位于D级，根据CU输出的不同branch类指令信号，比较对应 RS、RT寄存器数据是否符合条件并输出判断结果用于NPC计算转跳地址。

端口	方向	描述
[31:0] RS	I	输入RS寄存器的值
[31:0] RT	I	输入RT寄存器的值
[2:0] btype	I	branch指令类型
cmpresult	O	比较结果

实现代码

```
module D_CMP(
    input [31:0] rs,
    input [31:0] rt,
     input [2:0] btype,
    output cmp_result
    );

     assign cmp_result = ( ( btype == 3'b001 &&  rs == rt ) ||
                           ( btype == 3'b010 &&  rs != rt ) ||
                           ( btype == 3'b011 &&  ( rs[31] == 1'b1 || rs ==
32'b0) ) ||
                           ( btype == 3'b100 &&  ( rs[31] == 1'b0 && rs !=
32'b0) ) ||
                           ( btype == 3'b101 &&  ( rs[31] == 1'b1 && rs !=
32'b0) ) ||
                           ( btype == 3'b110 &&  ( rs[31] == 1'b0 || rs ==
32'b0) ) ||
                           ( btype == 3'b111 && $signed(rs) < $signed(32'b0))
                                   )? 1 : 0 ;
```

```
endmodule
```

（二）中断设计

CP0协处理器模块

本CPU将CP0协处理模块放在M级中。CP0 模块在 P7 实验中的主要作用是实现对异常、中断的控制，具体包括 SR、Cause、EPC、PRId 四个寄存器及与中断、异常相关的处理逻辑。

与中断异常相关的四个寄存器

SR 寄存器

- IM[7:2]，即 SR[15:10]，为 6 位中断屏蔽位，分别对应 6 个外部中断。

  相应位置 1 表示允许中断，置 0 表示禁止中断。

- IE，即 SR[0]，为全局中断使能。

  该位置 1 表示允许中断，置 0 表示禁止中断。

- EXL，即 SR[1]，为异常级。

  该位置 1 表示已进入异常，不再允许中断，置 0 表示允许中断。

Cause 寄存器

- IP[7:2]，即 Cause[15:10]，为 6 位待决的中断位，分别对应 6 个外部中断

  相应位置 1 表示有中断，置 0 表示无中断。

- ExcCode[6:2]，即 Cause[6:2]，异常编码，记录当前发生的是什么异常。

- BD，即 Cause[31]，记录发生中断的指令是否是延迟槽指令。当 BD 位被置高时, 就意味着 EPC 中存储的是发生中断异常的指令的上一指令的地址。

EPC 寄存器

EPC 寄存器负责保存中断/异常时的 PC 值。

PRId 寄存器

通常存入处理器 ID，可以用于实现个性的编码

CPU 在执行指令时，指令可能会在流水级的不同部位产生异常，如在 E 级可能发生算术指令异常、在 D 级可能发生跳转指令地址异常等。CPU 并不会直接处理这些异常，而是得到一个异常码 ExcCode，并将 `ExcCode` 继续向后流水直至将异常码提交给 CP0。对中断的处理同理，CPU 会将得到的外部中断信号 HWint，直接提交到 CP0。也就是说，CP0 模块负责接收 CPU 产生和接收到的各种中断请求。

CP0 在接收到中断请求后，首先会对请求的合法性进行检验。对于异常产生的中断请求，CP0 在确定当前未响应中断后，响应中断请求；对于接收到的外设中断请求，CP0 在确定 SR 寄存器相应位无中断屏蔽、当前无中断且全局中断使能允许中断后，响应该中断请求。其他情况下，CP0 不会响应中断请求。

CP0 确定响应中断请求意味着 CPU 将跳转至异常处理地址 `0x00004180`，CP0 会向 NPC 模块传递一个跳转信号。在 CPU 进入异常处理程序前，CP0 需要记录当前已进入中断（将 SR 寄存器的 EXL 置位），记录当前的异常码并将指令当前 PC 存储在 EPC 寄存器中（注意，若当前执行的指令为延迟槽指令，则需要存储 PC - 4）。CPU 执行 `eret` 指令从异常返回时，CP0 模块需负责记录当前已结束中断（将 SR 寄存器的 EXL 置 0)

检测的异常

异常与中断码	助记符与名称	指令与指令类型	描述
0	`Int`(外部中断)	所有指令	中断请求，来源于计时器与外部中断
4	`AdEL`(取指异常)	所有指令	PC地址未字对齐
4	`AdEL`(取指异常)	所有指令	PC地址超过 `0x3000 ~ 0x6ffc`
4	`AdEL`(取数异常)	`lw`	取数地址未与 4 字节对齐
4	`AdEL`(取数异常)	`lh`, `lhu`	取数地址未与 2 字节对齐
4	`AdEL`(取数异常)	`lh`, `lhu`, `lb`, `lbu`	取 Timer 寄存器的值
4	`AdEL`(取数异常)	load 型指令	计算地址时加法溢出
4	`AdEL`(取数异常)	load 型指令	取数地址超出 DM、Timer0、Timer1 的范围
5	`AdES`(存数异常)	`sw`	存数地址未 4 字节对齐
5	`AdES`(存数异常)	`sh`	存数地址未 2 字节对齐
5	`AdES`(存数异常)	`sh`, `sb`	存 Timer 寄存器的值
5	`AdES`(存数异常)	store 型指令	计算地址加法溢出
5	`AdES`(存数异常)	store 型指令	向计时器的 Count 寄存器存值
5	`AdES`(存数异常)	store 型指令	存数地址超出 DM、Timer0、Timer1 的范围
10	`RI`(未知指令)	-	未知的指令码
12	`Ov`(溢出异常)	`add`, `addi`, `sub`	算术溢出||

端口	方向	描述
[4:0] A1	I	取出CP0寄存器的地址
[4:0] A2	I	写入CP0寄存器的地址
[31:0] DIn	I	写入数据
[31:0] PC	I	受害指令的地址
[4:0] ExcCodeIn	I	异常类型
[5:0] HWInt	I	中断信号
We	I	CP0写使能
EXLClr	I	退出异常信号
clk	I	时钟
reset	I	清零信号
BDin	I	受害指令是否为延时槽指令
IntReq	O	中断信号
[31:0] EPCout	O	退出异常时，需回到原主程序的地址
[31:0] DOut	O	读出数据
interrupt_respond	O	响应中断的信号

```
module CP0(
    input [4:0] A1,
    input [4:0] A2,
    input [31:0] DIn,
    input [31:0] PC,
    input [6:2] ExcCodeIn,
    input [5:0] HWInt,
    input We,
    input EXLClr,
    input clk,
    input reset,
     input BDin,
    output IntReq,
    output [31:0] EPCout,
    output [31:0] DOut,
     output interrupt_respond
    );

     reg [31:0] SR, Cause,  PRId;
     reg [31:0] EPCReg;

    assign DOut = (  A1 == 5'd12 ) ? SR :
                  (  A1 == 5'd13 ) ? Cause :
                       (  A1 == 5'd14 ) ? EPCout  :
                       (  A1 == 5'd15 ) ? PRId :
                                          32'b0;
```

```
    assign interrupt_respond =     HWInt[2] & SR[12] & IE & !EXL  ;

    wire [7:2] IM;
    wire IE, EXL;

    assign IM = SR[15:10];
    assign IE = SR[0];
    assign EXL = SR[1];
    /////

    wire IReq , EReq;
    assign IReq = |( HWInt & IM ) & IE & !EXL ;
    assign Ereq = ( | ExcCodeIn  ) & !EXL;

    assign IntReq = IReq | Ereq;
    assign EPCout = ( IntReq & BDin )? PC-4:
                    ( IntReq & !BDin ) ? PC :
                                                EPCReg;



    initial begin
          SR <=32'b0;
            Cause <= 32'b0;
            EPCReg <= 32'b0;
            PRId <= 32'h1234_5678;
    end

    always@(posedge clk) begin
          if ( reset == 1'b1 ) begin
                SR <=32'b0;
                Cause <= 32'b0;
                EPCReg <= 32'b0;
                PRId <= 32'h1234_5678;
            end
            else begin
                if ( IntReq == 1'b1 ) begin
                      SR[1] <= 1'b1;  // EXL  SR[1] 赋为1
                        Cause[31] <=  BDin;                // BD = BDin;
                        EPCReg  <= EPCout;
                        if ( IReq == 1'b1 ) begin
                             Cause[6:2] <= 5'd0 ;
                    end
                    else begin
                             Cause[6:2] <= ExcCodeIn;
                    end
//ExcCode = Cause[6:2];
                end
                else if ( We == 1'b1 ) begin //排除了中断写入的情况
                       if (  A2 == 5'd12 ) begin
                               SR <= DIn;
                          end
                          else if (  A2 == 5'd14 ) begin
                               EPCReg <= DIn ;
                          end

                end
```

```
                    if ( EXLClr == 1'b1 ) begin
                         SR[1] <= 1'b0;
                    end     // EXLClr

                    Cause[15:10] <= HWInt;

               end // else if end


     end// alwasy end


endmodule
```

Bridge以及IO接口设计

Bridge与外置DM、Timer1、Timer0、CPU相连接，Bridge主要作用是作为CPU与外置DM模块以及Timer的数据传输中介。

功能表现

1.当需要写入数据时，根据CPU传入的Addr范围，选择为相应的外设写入使能端设置1，将CPU传入的待写入数据WD传输给外设的写入数据端。

2.当需要读取数据时，根据CPU传入的Addr范围，选择相应外设的输入数据，作为PRRD输出，传回CPU。

端口	方向	描述
[31:0] PrAddr	I	CPU传入地址
[31:0] PrWD	I	CPU传入数据
PrWE	I	CPU传入store指令类型写使能
[3:0] Prbeen	I	CPU传入的字节使能
[31:0] m_data_rdata	I	DM外设的输入数据
[31:0] TC0out	I	Timer0外设的输入数据
[31:0] TC1out	I	Timer1外设的输入数据
interrupt_respond	I	响应中断的写入信号
[31:0] PrRD	O	传回CPU的写入数据
[3:0] m_data_byteen	O	传给DM外设的字节使能
[31:0] m_data_addr	O	传给DM外设的地址
[31:0] m_data_wdata	O	传给DM外设的数据
TC0WE	O	Timer0的写使能
TC1WE	O	Timer1的写使能
TC0Addr	O	传给Timer0外设的地址
TC0WD	O	传给Timer0外设的数据
TC1Addr	O	传给Timer1外设的地址
TC1WD	O	传给Timer1外设的数据

代码实现

```
module Bridge(
    input [31:0] PrAddr,
    input [31:0] PrWD,
    input PrWE,
    input [3:0] Prbeen,
    input [31:0] m_data_rdata,
    input [31:0] TC0out,
    input [31:0] TC1out,
     input interrupt_respond,
     output [31:0] PrRD,
    output [3:0]  m_data_byteen,
    output [31:0] m_data_addr,
    output [31:0] m_data_wdata,
    output TC0WE,
    output TC1WE,
     output [31:0] TC0Addr,
     output [31:0] TC0WD,
     output [31:0] TC1Addr,
```

```
    output [31:0] TC1WD
   );
    wire  HitTC0,HitTC1 ,HitDM;

    assign  HitDM   =  ( PrAddr >= 32'h0000_0000  && PrAddr <= 32'h0000_2FFF )?
1:0;
    assign  HitTC0   = ( PrAddr >= 32'h0000_7F00  && PrAddr <= 32'h0000_7F0B )?
1:0; //0x0000_7F10 至 0x0000_7F1B
    assign  HitTC1   = ( PrAddr >= 32'h0000_7F10  && PrAddr <= 32'h0000_7F1B )?
1:0;


    assign  TC0WE =  HitTC0 & PrWE;
    assign  TC1WE =  HitTC1 & PrWE;

    assign m_data_byteen = interrupt_respond ? 4'b1111:
                           HitDM ? Prbeen:
                                        4'b0000;

   assign m_data_addr = interrupt_respond ?  32'h0000_7f20:
                                                        PrAddr;

   assign m_data_wdata = PrWD;
    assign TC0WD = PrWD;
    assign TC0Addr = PrAddr;
    assign TC1WD = PrWD;
    assign TC1Addr = PrAddr;

    assign PrRD = HitDM  ?  m_data_rdata:
                  HitTC0 ?   TC0out :
                 HitTC1 ?   TC1out :
                                         32'b0;

endmodule
```

(四) 控制器CU设计思路

端口	方向	描述
[31:0] instr	I	指令
[4:0] rs_addr	O	rs寄存器地址
[4:0] rt_addr	O	rt寄存器地址
[4:0] rd_addr	O	rd寄存器地址
[25:0] imm	O	立即数
[1:0] EXT	O	EXT模块控制信号
branch	O	beq指令信号
jump	O	jal、j指令信号
jrtype	O	jr指令信号
ALUdataAsrc	O	ALU dataA端的数据选择信号
ALUdataBsrc	O	ALU dataB端的数据选择信号
RFWE	O	寄存器写入信号
[3:0] ALUctrl	O	ALU控制信号
[1:0] RFWD_Type	O	寄存器写入数据来源类型判断
[4:0] RFaddr	O	寄存器写入地址
calc_r	O	指令类型为 R信号
calc_i	O	指令类型为 I信号
load	O	指令类型为 load信号
store	O	指令类型为 store信号
lui	O	指令为 lui信号
j_imm	O	指令为直接转跳立即数的信号
j_rs	O	指令为转跳寄存器的存储地址的信号
j_link	O	指令为转跳并链接地址到寄存器的信号
[2:0] btype,	O	branch类指令的分类信号
shifts	O	S型位移类指令的识别信号
shiftv	O	V型位移类指令的识别信号
md	O	乘除指令指令的识别信号
mf	O	读HI、LO寄存器指令的识别信号
mt	O	写HI、LO寄存器指令的识别信号
branch_t_s	O	branch类指令识别，但在D级只使用rs寄存器

端口	方向	描述
branch_t_d	O	branch类指令识别，但在D级使用rs、rt寄存器
[3:0] MDUop	O	与MDU模块相关的指令分类信号
[1:0] s_type	O	store类型指令，分类信号
[2:0] EXT_op	O	load类指令扩展信号

控制器CU主要实现3个功能：

1.输出指令所对应的rs、rt、rd寄存器地址以及立即数。

2.输出指令所对应的各类模块的控制信号。

3.输出指令所属的指令类型，该功能主要用于SU模块，根据输出的指令类型更新Tuse和Tnew的值，用于分析是否需要暂停

控制信号	calc_R	calc_i	shifts	shiftv	load	store	branch
RFWE	1	1	1	1	1	0	0
[3:0] ALUctrl	视具体指令	视具体指令	视具体指令	视具体指令	0010	0010	X
[1:0] EXT	XX	ori andi xori 01 lui 10 其他 00	XX	XX	00	00	XX
branch	0	0	0	0	0	0	1
jump	0	0	0	0	0	0	0
jrtype	0	0	0	0	0	0	0
ALUdataAsrc	0	0	1	1	0	0	X
[1:0] ALUdataBsrc	00	01	11	10	01	01	XX

控制信号	md	mf	mt	j	jal	jr	jalr
RFWE	0	1	0	0	1	0	1
[3:0] ALUctrl	XXXX	XXXX	XXXX	XXXX	XXXX	XXXX	XXXX
[1:0] EXT	XX	XX	XX	XX	XX	XX	XX
branch	0	0	0	0	0	0	0
jump	0	0	0	1	1	0	0
jrtype	0	0	0	0	0	1	1
ALUdataAsrc	X	X	X	X	X	X	X
[1:0] ALUdataBsrc	XX	XX	XX	XX	XX	XX	XX

MDUop控制信号

```
assign MDUop  = mult ?  4'd1:
                multu ? 4'd2:
                div?    4'd3:
                divu ?  4'd4:
                mfhi ?  4'd5:
                mflo ?  4'd6:
                mthi ?  4'd7:
                mtlo ?  4'd8:
                                4'd0;
```

store类型控制信号

```
assign  s_type = sw? 2'b01:
                 sh? 2'b10:
                 sb? 2'b11:
                  2'b00;
```

load类扩展控制信号

```
assign  EXT_op = lbu? 3'b001:
                 lb?  3'b010:
                 lhu? 3'b011:
                 lh?  3'b100:
                 3'b000;
```

(五)暂停转发设计

1. 转发设置

确定转发位置

转发设置较为简单，首先是要判断哪些模块的输入接口需要转发，根据课件的知道，需要寄存器值输入的模块都需要转发，所以确定的所需转发地方如下：

1.ID级CMP的RS、RT寄存器输入端

2.ID级NPC的RS输入端同样也需要转发，但可以与CMP输入端共用一个转发选择器

3.EX级 ALU的dataA、dataB输入端

4.DM的数据输入端，当s执行sw指令时，需要使用寄存器的值

确定转发来源

当确定所需转发的端口后，我们在进行分析，转发数据的来源，显然，每一级转发的来源都来自其后各级流水线寄存器内的值。

ID级转发来源：E级流水线寄存器中的PC

M级流水线寄存器中的PC和ALUresult

W级流水线寄存器中的PC 、ALUresult、DMresult、HILO寄存器读出的值

EX级转发来源：M级流水线寄存器中的PC和ALUresult

W级流水线寄存器中的PC 、ALUresult、DMresult、HILO寄存器读出的值

DM级转发来源：W级流水线寄存器中的PC 、ALUresult、DMresult、HILO寄存器读出的值

判断是否需要转发

当需求者需要使用的**寄存器地址**与转发来源的**写入寄存器地址**（即要求转发来源处**寄存器写入信号为1**） 相同且不为0时，转发。

代码实现

为了便于转发的选择器的书写以及结构清晰，我们可以将同一级的流水线寄存器中的输出做一个选择，只要在CU中加一个输入数据类型判断信号，根据输出类型信号选择转发的数据即可。比如，当前位于M级的指令为addu，类型为calc_R，所以M级的转发数据应该选择为 ALUresult。

选择示例如下：

```
assign W_WD = ( RFWDtype_W == 2'b00 )? W_ALUresult:
                  ( RFWDtype_W == 2'b10 )? W_pc+8:
                      ( RFWDtype_W == 2'b01 )? W_dm:
                  ( RFWDtype_W == 2'b11 )? W_HILO:
                                                32'b0;
```

转发控制代码如下：

```
    wire [31:0] FW_rs_D = ( D_rs == 5'd0 )? 5'd0:
                          ( D_rs == RFaddr_E &&  RFWR_E == 1'b1 )?  E_WD:
                          ( D_rs == RFaddr_M &&  RFWR_M == 1'b1 )?  M_WD:
                          ( D_rs == RFaddr_W &&  RFWR_W == 1'b1 )?  W_WD:

RD1;
    wire [31:0] FW_rt_D = ( D_rt == 5'd0 )? 5'd0:
                          ( D_rt == RFaddr_E &&  RFWR_E == 1'b1 )?  E_WD:
                          ( D_rt == RFaddr_M &&  RFWR_M == 1'b1 )?  M_WD:
                          ( D_rt == RFaddr_W &&  RFWR_W == 1'b1 )?  W_WD:

RD2;
```

2.暂停设置

根据教程AT法的要求，**当 D 级指令读取寄存器的地址与 E 级或 M 级的指令写入寄存器的地址相等且不为 0，且 D 级指令的 Tuse 小于对应 E 级或 M 级指令的 Tnew 时**，我们就需要在 D 级暂停指令。在其他情况下，数据冒险均可通过转发机制解决。

暂停控制信号以SU模块实现，在SU模块中，我们只需要根据位于D 、E、M级中的指令，计算D级的 **Tuse_rs, Tuse_rt** 和 **Tnew_E, Tnew_M, Tnew_W。** 在实际效果中，由于Tnew_M一直为0或者不存在，所以不会触发暂停。

Tuse表

指令类型	rs	rt
calc_R	1	1
calc_I	1	X
lui	X	X
load	1	X
store	1	2
beq|bne	0	0
j	X	X
jal	X	X
jr|Jalr	0	X
shifts	X	1
shiftv	1	1
md	1	1
mt	1	X
BLEZ、BGTZ、 BLTZ、BGEZ	0	X
mf	X	X

备注：由于转发机制是Tuse 小于对应 E 级或 M 级指令的 Tnew ，所以对于那些不使用寄存器的指令，我们需要将他们的Tuse X视为无穷大，代码中用一个比2大的数表示即可。

Tnew表

指令类型	E	M	W
calc_R	1	0	0
calc_I	1	0	0
lui	1	0	0
load	2	1	0
store	X	X	X
branch	X	X	X
j	X	X	X
jal |jalr	0	0	0
jr	X	X	X
shifts	1	0	0
shiftv	1	0	0
mf	1	0	0
mt	X	X	X
md	X	X	X

备注：由于转发机制是Tuse 小于对应 E 级或 M 级指令的 Tnew ，所以对于那些不产生寄存器新值的指令，我们需要将他们的Tnew X视为无穷小，代码中用一个小于等于0的数表示即可。

乘除模块增加的暂停判断：当 Busy 或 Start 信号为 1 时，位于D级的指令mfhi、mflo、mthi、mtlo、mult、multu、div、divu 均被阻塞在D级。

暂停判断代码实现

```
module SU(
    input [31:0] D_instr,
    input [31:0] E_instr,
    input [31:0] M_instr,
    input [31:0] W_instr,
     input D_cmpresult,
     input E_cmpresult,
     input M_cmpresult,
     input  HILO_busy,
    output  stall
    );


     wire [4:0] D_rs_addr , D_rt_addr;
     wire  D_calc_r, D_calc_i, D_load, D_store, D_j_rs, D_shifts, D_shiftv,
branch_link_D ;
```

```
     wire  D_md, D_mf, D_mt;
     wire [1:0] Tuse_rs , Tuse_rt;
     wire  branch_t_s, branch_t_d;

     CU D_CU (
    .instr(D_instr),
     .cmp_result(D_cmpresult),
    .rs_addr(D_rs_addr),
    .rt_addr(D_rt_addr),
    .calc_r(D_calc_r),
    .calc_i(D_calc_i),
    .load(D_load),
    .store(D_store),
    .j_rs(D_j_rs),
     .branch_link(branch_link_D),
     .shifts( D_shifts ),
    .shiftv( D_shiftv ),
     .md(D_md),
     .mf(D_mf),
    .mt(D_mt),
     .branch_t_s(branch_t_s),
    .branch_t_d(branch_t_d)
    );

     assign Tuse_rs = ( D_calc_r |  D_calc_i | D_load | D_store | D_shiftv |
D_md | D_mt) ? 2'b01:
                     ( branch_t_d | branch_t_s | D_j_rs | branch_link_D)? 2'b00:
                                                          2'b11;
    assign Tuse_rt =  D_store ? 2'b10 :
                      ( D_calc_r | D_shifts | D_shiftv | D_md ) ? 2'b01:
                      branch_t_d? 2'b00 :
                                                 2'b11;              // 课上的
Branch_link指令rt tuse可能是0


    wire [4:0] E_A3;
   wire E_calc_r , E_calc_i , E_load, E_lui, E_WE, E_shifts, E_shiftv ,E_mf;
    wire [1:0] Tnew_E;
    CU E_CU (
    .instr(E_instr),
     .cmp_result(E_cmpresult),
    .RFWE(E_WE),
    .RFaddr(E_A3),
    .calc_r(E_calc_r),
    .calc_i(E_calc_i),
    .load(E_load),
    .lui(E_lui),
     .shifts( E_shifts ),
    .shiftv( E_shiftv ),
     .mf(E_mf)
    );

     assign Tnew_E = E_load? 2'b10:
                     ( E_calc_r | E_calc_i | E_lui | E_shifts | E_shiftv | E_mf )
? 2'b01:
                                                                  2'b00;
    wire [4:0] M_A3;
   wire M_WE, M_load;
```

```
     CU  M_CU(
    .instr(M_instr),
     .cmp_result(M_cmpresult),
    .RFWE(M_WE),
    .RFaddr(M_A3),
    .load(M_load)
    );

    assign Tnew_M = M_load ? 2'b01:
                                    2'b00;

   wire stall_rs_E ,stall_rs_M , stall_rt_E ,stall_rt_M;
    assign stall_rs_E = ( Tuse_rs < Tnew_E ) && ( D_rs_addr == E_A3 ) && ( E_A3
!= 5'd0) && (E_WE == 1'b1);
   assign stall_rs_M = ( Tuse_rs < Tnew_M ) && ( D_rs_addr == M_A3 ) && ( M_A3
!= 5'd0) && (M_WE == 1'b1);

    assign stall_rt_E = ( Tuse_rt < Tnew_E ) && ( D_rt_addr == E_A3 ) && ( E_A3
!= 5'd0) && (E_WE == 1'b1);
   assign stall_rt_M = ( Tuse_rt < Tnew_M ) && ( D_rt_addr == M_A3 ) && ( M_A3
!= 5'd0) && (M_WE == 1'b1);

    wire  stall_rs = stall_rs_E | stall_rs_M;
    wire  stall_rt = stall_rt_E | stall_rt_M;
    wire  stall_HILO = HILO_busy & ( D_md | D_mf | D_mt );

    assign stall = stall_rs | stall_rt |  stall_HILO ;

endmodule
```

（三）思考题

1、我们计组课程一本参考书目标题中有“硬件/软件接口”接口字样，那么到底什么是“硬件/软件接口”？(Tips：什么是接口？和我们到现在为止所学的有什么联系？)

硬件接口为物理意义上真实存在的接口，一般是设备间用来进行数据交换的通道，比如usb接口。软件接口就是指程序中具体负责在不同模块之间传输或接受数据的并做处理的类或者函数，而不是指具体的数据。

2、BE 部件对所有的外设都是必要的吗？

并不是，BE部件是用于定义字节使能端，适用于可能会出现加载或存储字节单位的存储部件，而有些外设可能只会存储字的单位，比如Timer。

3、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，并分别针对每一种模式绘制状态转移图。

见Timer文档

4、请开发一个主程序以及定时器的 exception handler。整个系统完成如下功能：

（1）定时器在主程序中被初始化为模式 0；

（2）定时器倒计数至 0 产生中断；

（3）handler 设置使能 Enable 为 1 从而再次启动定时器的计数器。(2) 及 (3) 被无限重复。

（4）主程序在初始化时将定时器初始化为模式 0，设定初值寄存器的初值为某个值，如 100 或 1000。(注意，主程序可能需要涉及对 CP0.SR 的编程，推荐阅读过后文后再进行。)

```
.text
ori $t1, $0, 9
sw $t1, 0x7000($0)
ori $t2, $0, 20
sw $t2, 0x7004($0)
jump:
beq $s0, $s0,jump
nop
nop

.ktext 4180
ori $t1, $0,9
sw  $t1, 0x7100($0)
eret
```

5、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?

以Windows平台为例,键鼠等外设产生电子信号，触发中断。驱动程序响应中断并通知系统层面。

当前Windowstation进行响应(比如重新绘制光标)，产生对应的系统消息。