

2021年北航计组P6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、SRV、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}。为了实现这些功能，CPU主要包含了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	$A \oplus B$
0101	$A \ll B$
0110	$A \gg B$
0111	$A \ggg B$
1000	$\sim(A B)$
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 $D_pc_{31-28} || index || 00$
- 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

        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
                                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
end

```

```

        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 `1b`、`1bu`、`1h`、`1hu` 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单元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

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
```

(三) 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
shftv	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 = 1bu? 3'b001:
                1b? 3'b010:
                1hu? 3'b011:
                1h? 3'b100:
                3'b000;

```

(三)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
shftv	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
shftv	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_shftv,  
    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 ),
.shiftn(D_shiftn),
.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_shiftn |
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_shiftn | 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_shiftn ,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 ),
.shiftn(E_shiftn),
.mf(E_mf)
);

assign Tnew_E = E_load? 2'b10:
                ( E_calc_r | E_calc_i | E_lui | E_shifts | E_shiftn | 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

```

二 测试代码

shifts 、shiftv类型后与branch类指令的冲突

```

ori    $0,$21,76
sra    $0,$0,2
bne    $0,$0,TAG_16
addiu  $0,$0,1
addiu  $0,$0,1
TAG_16:
mflo   $1
mfhi   $2
addi   $1,$0,31
addi   $21,$0,61
mfhi   $13
slti   $13,$13,-6
sll    $13,$13,2
blez   $13,TAG_17
addiu  $13,$13,1
addiu  $13,$13,1
TAG_17:
mflo   $1
mfhi   $2
addi   $1,$0,38

```

```

mflo      $7
sltui     $7,$7,5
srl       $8,$7,2
bgtz      $7,TAG_18
addiu     $7,$7,1
addiu     $7,$7,1
TAG_18:
mflo      $1
mfhi      $2
addi      $1,$0,0
addi      $8,$0,29
mfhi      $0
xori      $0,$8,112
sra       $8,$0,2
bgez      $0,TAG_19
addiu     $0,$0,1
addiu     $0,$0,1
TAG_19:
mflo      $1
mfhi      $2
addi      $1,$0,38
addi      $8,$0,147
mflo      $14
addi      $14,$14,-211
sll       $14,$14,1
blez      $14,TAG_20
addiu     $14,$14,1
addiu     $14,$14,1
TAG_20:
mflo      $1
mfhi      $2
addi      $1,$0,232
mfhi      $7
addiu     $7,$7,-193
srl       $7,$9,1
bgtz      $7,TAG_21
addiu     $7,$7,1
addiu     $7,$7,1
TAG_21:
mflo      $1
mfhi      $2
addi      $1,$0,101
mflo      $7
andi      $7,$7,166
sra       $7,$7,1

```

E级load类指令与D级shiftv的冲突

```

ori $1, $0, 0x01020304
ori $2, $0, 0x02030407
ori $3, $0, 0x08051117
ori $4, $0, 0x15131519
ori $5, $0, 2
ori $10, $0, 6
sw $4, 0($0)
sw $3, 4($0)
sw $1, 8($0)
sw $1, 12($0)

```

```

sw $3, 100($0)
sw $2, 104($0)
sw $2, 108($0)
sw $4, 112($0)
sw $3, 116($0)
lh $7, 110($0)
sllv $11, $7, $5
lh $6, 102($0)
sllv $11, $3, $6

```

E级load类指令与D级md、mt类的冲突

```

ori $1, $0, 1
ori $2, $0, 6
ori $3, $0, 4
ori $4, $0, 4
sw $4, 0($0)
sw $3, 4($0)
sw $1, 8($0)
sw $1, 12($0)
sw $3, 100($0)
sw $2, 104($0)
sw $2, 108($0)
sw $4, 112($0)
sw $3, 116($0)
lbu $1, 0($2)
mthi $1
lw $8, 100($0)
lw $9, 112($0)
mult $8, $9
mfhi $11
mflo $12

```

load类与branch类的冲突

calc_R类与branch类的冲突

```

ori $t1, $t1, 125
ori $t2, $t2, 200
ori $t3, $t3, 25
ori $t4, $t4, 50
subu $t2, $t2, $t4
beq $t1, $t2, label1
sw $t1, 300($0)
sw $t2, 400($0)
sw $t3, 500($0)
label1:
lui $s1, 0x3000
lui $s2, 0x2000
subu $t2, $t2, $t3
beq $t1, $t2, label2
nop
sw $s1, 700($0)
sw $s2, 800($0)
label2:
sw $s1, 100($0)
sw $s2, 200($0)

```

```

ori $t1, $t1, 300

```

```

ori $t2, $t2, 100
sw $t2, 120($0)
beq $t1, $t2, label3
nop
ori $s1, $s1, 1
ori $s2, $s2, 2
label3:
lw $t1, 120($0)
beq $t1, $t2, label4
nop
ori $s3, $s3, 3
ori $s4, $s4, 4
label4:
ori $5, $5, 5
ori $6, $6, 6

ori $t1, $t1, 400
ori $t2, $t2, 600
ori $t3, $t3, 100
ori $t4, $t4, 50
sw $t3, 100($0)
lw $t4, 100($0)
subu $t1, $t1, $t4
addu $t2, $t2, $t4

```

三 思考题

1. 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？

ALU模块为组合逻辑，乘除法部件的设计涉及到时序逻辑，运用单独的乘除法部件可以避免逻辑的混乱。乘除法运算周期较大，在程序中的模拟分别为5个周期、10周期，如果将乘除法部件整合进ALU，会影响CPU的执行效率。将乘除法部件独立，可以使后续的涉及乘除部件的指令正常流水下去，在乘除法运算的周期，可以继续执行其他指令，提高CPU整体效率。

2. 参照你对延迟槽的理解，试解释“乘除槽”。

“乘除槽”，我的理解是，如果乘除指令的后续指令与乘除法部件有关，那么都将阻塞在D级，与延时槽类似。

3. 举例说明并分析何时按字节访问内存相对于按字访问内存性能上更有优势。（Hint：考虑 C 语言中字符串的情况）

比如在C程序中，有很多需要访问字节的操作，针对char类型的操作，那按字节访问，会比按字访问更节省时间。

冲突列表如下:

		Tnew_M	calc_R 0	calc_i 0	lui 0	load 1	store 0	beq bne 0	j jr 0	jal jalr 0	shifts 0	shiftv 0	md 0	mt 0	mf 0	bgt blz 0
	Tuse_rt															
calc_R	1															
calc_I	3															
lui	3															
load	3															
store	2															
beq bne	0					S										
j jal	3															
jr jalr	3															
shifts	1															
shiftv	1															
md	1															
mf	3															
mt	1															
BLEZ- BGIZ, BLTZ- BGEZ	3															

5.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？

运用了指令分类，主要将所有指令分成了 calc_R、calc_i、shiftr、shifts、md、mf、mt、branch（根据Tuse时间不同，细分成了branch_S,branch_D），j_link, j_imm, j_rs, 所属同一类型的指令控制信号大致相同，Tuse 和 Tnew 的时间也大致相同，所以用分类后的识别信号来驱动 Tuse Tnew以及各种模块的控制信号，能减少复杂性。

