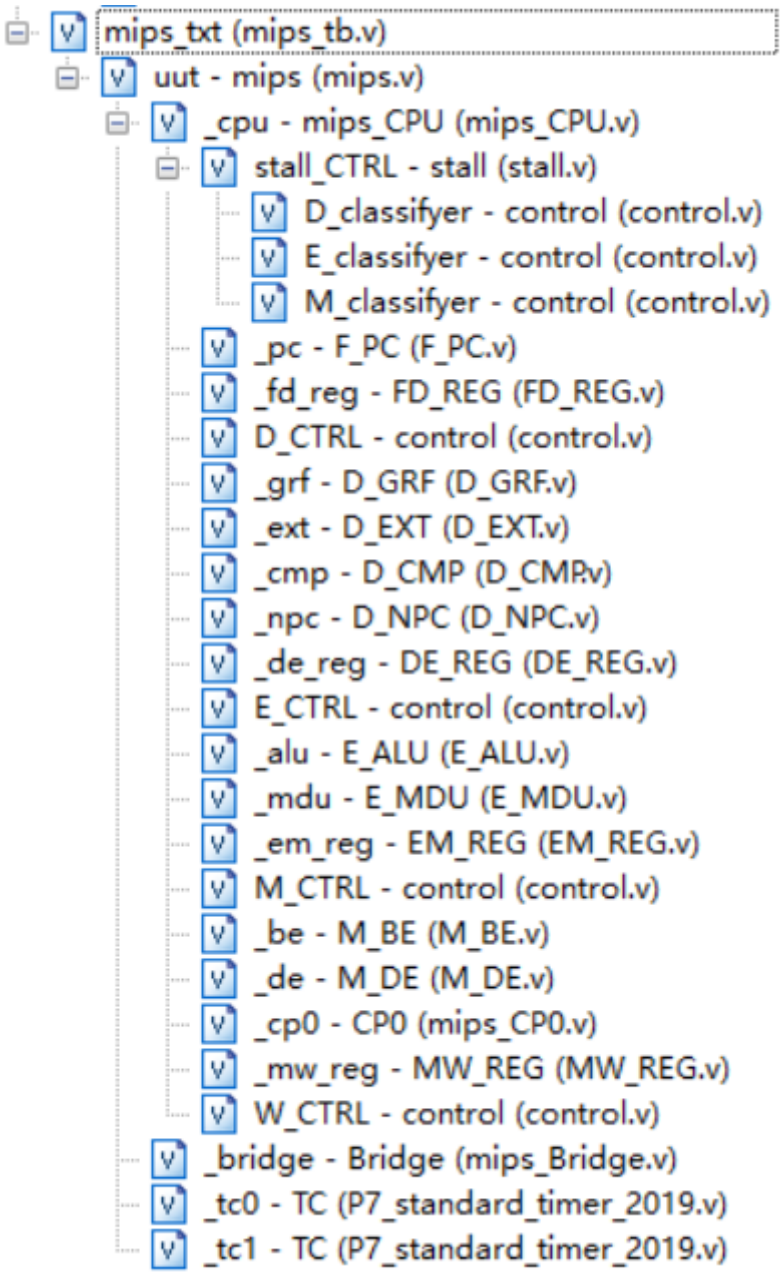
计算机组成原理实验报告—MIPS微系统设计

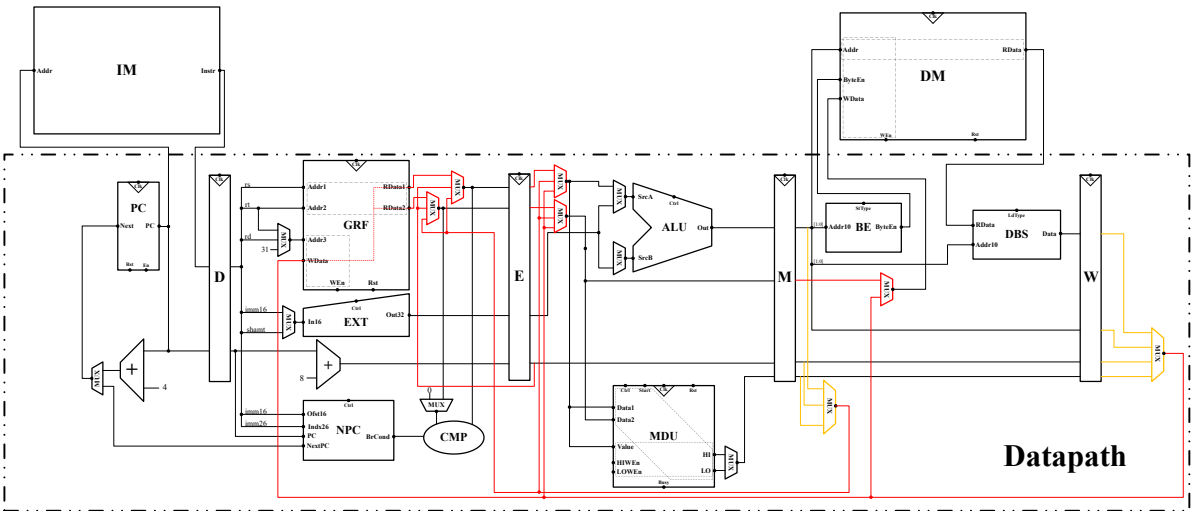
总体设计概述

要求实现的指令集为`MIPS-C4`，即**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、MFC0、MTC0、ERET**，在P6的基础上新增加了MFC0、MTC0、ERET

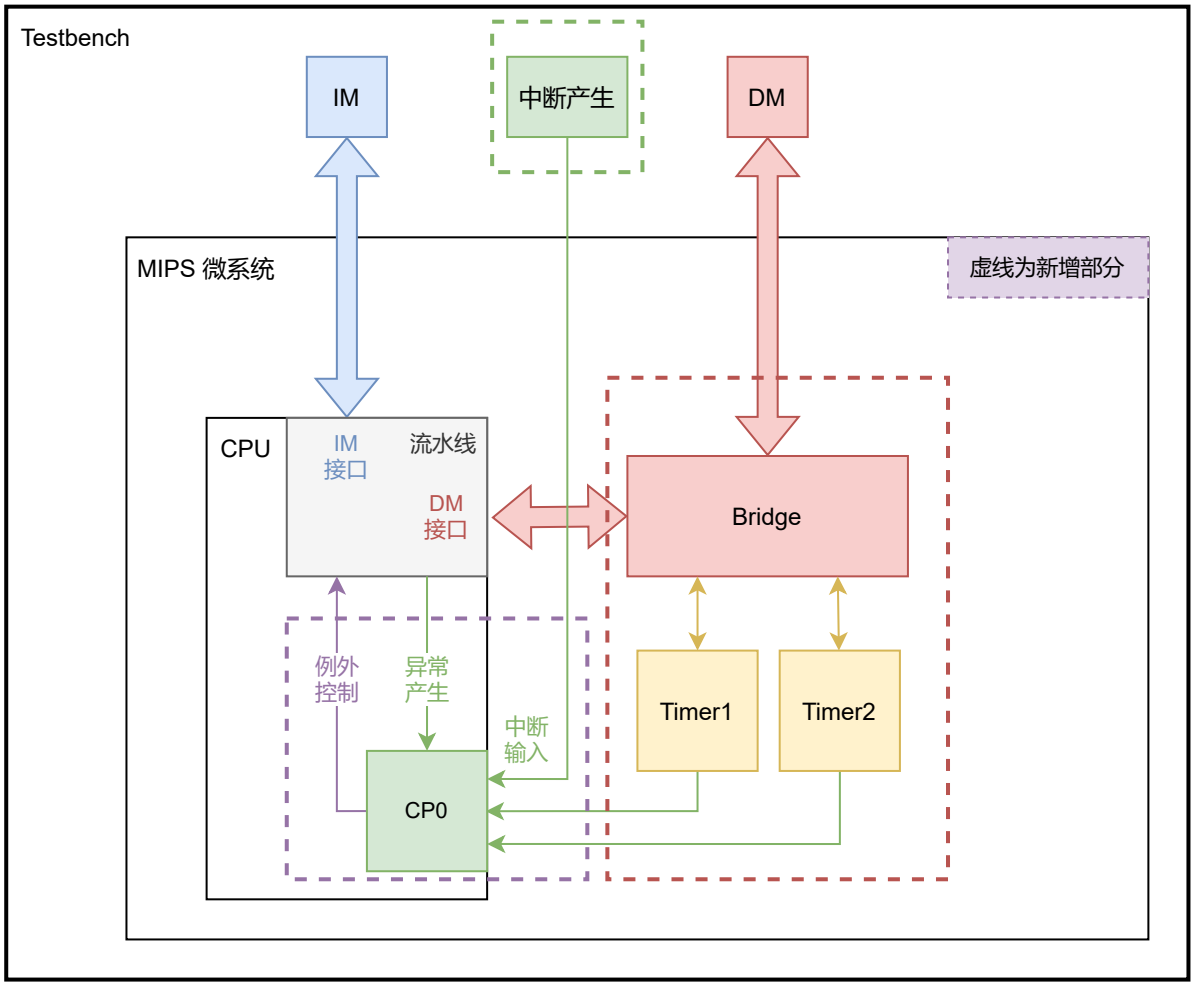
整体结构目录树如下：



CPU内部通路设计如下图：



MIPS微系统整体设计如下图：



下面将分别从CPU、系统桥Bridge与I/O来介绍微系统的主要组成

CPU

CPU部分与P6相比添加了重要模块CP0协处理器，其余的端口定义和转发、阻塞规则与P6相同，详见附带的P6设计文档，在此处不再赘述

考虑到宏观PC的处理，我把CP0协处理器放置在了M级

CP0协处理器

介绍

协处理器 0，包含 4 个 32 位寄存器，用于支持中断和异常。

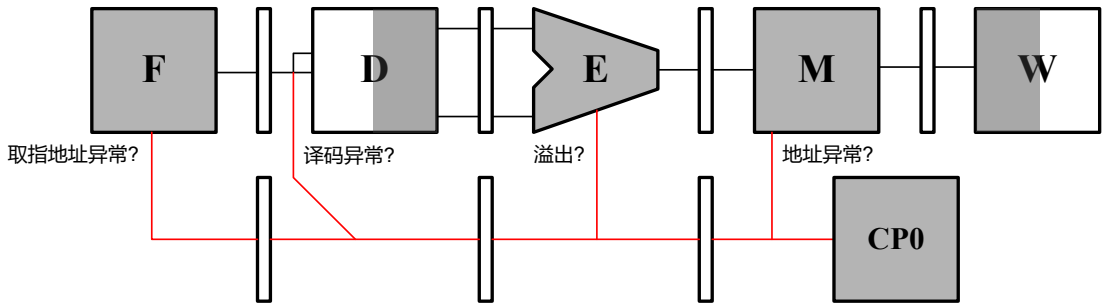
端口定义

端口	输入/输出	位宽	描述
A1	I	5	指定 4 个寄存器中的一个，将其存储的数据读出到 RD
A2	I	5	指定 4 个寄存器中的一个，作为写入的目标寄存器
Din	I	32	写入寄存器的数据信号
PC	I	32	目前传入的下一个 EPC 值
ExcCodeIn	I	5	目前传入的下一个 ExcCode 值
isInDelaySlot	I	32	目前传入的下一个 BD 值
HWInt	I	6	外部硬件中断信号
WE	I	1	写使能信号，高电平有效
EXLClr	I	1	传入 eret 指令时将 SR 的 EXL 位置 0 ，高电平有效
clk	I	1	时钟信号
reset	I	1	同步复位信号
Req	O	1	输出当前的中断请求
EPCOut	O	32	输出当前 EPC 寄存器中的值
Dout	O	32	输出 A 指定的寄存器中的数据
TestIntResponse	O	1	检测CPU是否对外部中断产生响应，从而决定是否去写 `0x0000_7f20`

功能定义

序号	功能名称	功能描述
1	同步复位	当时钟上升沿到来且同步复位信号有效时，将所有寄存器的值设置为0x00000000。
2	读数据	读出 A1 地址对应寄存器中存储的数据到 RD；当 WE 有效时会将 WD 的值会实时反馈到对应的 RD，当 ERET 有效时会将 EXL 置 0，即内部转发。
3	写数据	当 WE 有效且时钟上升沿到来时，将 WD 的数据写入 A2 对应的寄存器中。
4	中断处理	根据各种传入信号和寄存器的值判断当前是否要进行中断，将结果输出到 IntReq。

处理异常的流程如下图：



将异常码ExcCode、是否处于延迟槽中的判断信号isInDelaySlot和当前PC（如果时取指地址异常则传递错误的PC值）一直跟着流水线到达M级直至提交至CP0，由CP0综合判断分析是否响应该异常

如果需要响应该异常，则CP0输出Req信号置为1，此时FD、DE、DM、MW寄存器响应Req信号，清空Instr，将PC值设为0x4180，然后输入F级的NPC也被置为0x4180，下一条指令从0x4180开始执行

当外设和系统外部输入中断信号时，CP0同样也会确认是否响应该中断，然后把Req置为1，执行相同的操作

当系统外部输入中断信号时，CP0还会输出一个TestIntResponse信号指示是否响应外部中断信号，如果响应则系统会相应去写0x7f20地址，从而时外部中断信号停止

下面是P6的实验报告的部分内容：

所有的流水线寄存器均需要添加isInDelaySlot, ExcCode的传递

命名规范

- 对于元件的文件命名，均为`流水线层级_元件英文简称`，例如`D_GRF.v`，`E_ALU.v`等，实例化时命名为`_小写英文名`，例如`_alu`，`_grf`等
- 对于流水线寄存器文件命名为`两边的流水线层级_REG`，例如`FD_REG.v`，`DE_REG.v`，实例化时命名为`_小写英文名`，例如`_fd_reg`
- 每一级的控制信号和临时的`wire`均以本级的名称开头，如`E_ALUOp`，`M_DMOp`等
- 在流水线中参与流水的信息遵从以下约定（以D级为例）
  - `PC`和`Instr`命名以流水线层级开头，如`D_PC`，`D_Instr`
  - 寄存器地址分别为`D_rs_addr`，`D_rt_addr`，读出数据为`D_rs_data`，`D_rt_data`
  - 转发得到的寄存器数据（直接读取也视为一种转发）记作`D_FWD_rs_data`，`D_FWD_rt_data`
  - 即将写入的寄存器地址为`E_GRFA3`，即将写入的数据记作`E_GRFWD`，选择信号为`E_GRFWDSel`

下面将按照流水线层级逐一分析各个单元

F级(Fetch/取指令)

- 本级没有转发，阻塞时需要取消`PC`写使能
- 本级的输入有来自D级的`NPC`，本级的输出是`F_PC`和`F_Instr`，两者需要参与流水线流水

PC（程序计数器）

信号名称	方向	功能描述
NPC[31:0]	输入	待写入PC的指令地址
clk	输入	时钟信号
reset	输入	同步复位信号
PC_WrEn	输入	PC的写使能
PC	输出	当前指令地址
Req	输入	是否有中断请求，如果有中断请求，则把PC置为0x4180
eret	输入	是否为eret指令，如果是，把PC设为EPC
EPC[31:0]	输入	输入的EPC的值

然后与`mips_txt.v`交互获得当前指令

```
F_PC _pc(
    .clk(clk),
    .reset(reset),

    .PC_WrEn(PC_WrEn),
    .NPC(NPC),

    .PC(F_PC)
);

assign i_inst_addr = F_PC;
assign F_Instr = i_inst_rdata;
```

D级(Decode/译码)

- 本级需要处理来自E, M, W级的转发，其中W级为寄存器内部转发，另外两个分别是`D_FWD_rs`,`D_FWD_rt`，在`CMP`和`NPC`中需要用
- 本级的输入是来自F级的`PC`和`Instr`，输出是`D_rs_data`，`D_rt_data`，`D_ext32`，`D_PC`和`D_Instr`，还有输出到F级的`NPC`
- 本级元件较多，比较复杂

FD_REG（F/D级流水线寄存器）

```
module FD_REG(
    input clk,
    input reset,
    input flush,
    input WrEn,
    input Req,

    input [31:0] F_PC,
    input [31:0] F_Instr,
    input F_DelaySlot,
    input [4:0] F_EXCCode,

    output reg [31:0] D_PC,
    output reg [31:0] D_Instr,
    output reg D_DelaySlot,
    output reg [4:0] D_EXCCode
    );
```

D_GRF（寄存器堆）

端口说明

信号名称	方向	功能描述
A1[4:0]	输入	5位地址输入信号，将其储存的数据读出到RD1
A2[4:0]	输入	5位地址输入信号，将其储存的数据读出到RD2
A3[4:0]	输入	5位地址输入信号，将其作为写入数据的目标寄存器
RD1[31:0]	输出	输出A1指定的寄存器中的32位数据
RD2[31:0]	输出	输出A2指定的寄存器中的32位数据
WD[31:0]	输入	32位数据输入信号
clk	输入	时钟信号
reset	输入	异步复位信号，将32个寄存器中的数据清零；1：复位；0：无效

- 这次删去了`GRFWrEn`写使能信号，因为如果我们不写寄存器，可以把`GRFA3`设为0，就相当于不写寄存器了

控制信号说明

1. `D_GRFA3`

直接给出待写入寄存器的地址，弃用了在P4中利用`GRFA3Sel`进行选择的设计，这是因为P5采用分布式译码，每一级都需要`GRFA3`的信息，因此在`CTRL`里面直接集成了

2. `D_GRFWDSel`

控制信号值	功能
`WDSel_dmrd`	选择写入寄存器的数据来自DM
`WDSel_aluans`	选择写入寄存器的数据来自ALU运算结果
`WDSel_pc8`	选择写入寄存器的数据为**当前流水线层级中的PC+8**

D_EXT（位扩展）

将16位二进制数进行零扩展或符号扩展到32位

控制信号说明

控制信号值	功能
`EXT_unsign`	零扩展
`EXT_sign`	符号扩展

D_CMP（比较器）

把原来ALU中比较值是否相等的运算移到了CMP里面，去指导`beq`这一类型的指令是否跳转

控制信号目前只有`CMP_beq`，未来可以扩展

端口说明

信号名称	方向	功能描述
rs[31:0]	输入	**处理完转发后** `$rs` 寄存器的值
rt[31:0]	输入	**处理完转发后** `$rt` 寄存器的值
CMPOp[2:0]	输入	控制信号
b_jump	输出	指示是否跳转，输入 `NPC`，未来可以添加 `bltzal` 指令

D_NPC（次地址计算单元）

把 `beq` 是否执行的判断交给了 `CMP`，直接根据输入信号 `jump` 判断是否跳转

其实 `NPC` 横跨了F级和D级两级，因为同时会输入 `F_PC` 和 `D_PC`，前者正常跳转 `F_PC+4` 用，后者则用于流水 `PC` 值，后面转发 `PC+8` 的时候用

这次我们弃用了P4中直接输出 `PC+4` 的设计，转而让 `PC` 信号参与流水，在需要转发时计算 `PC+8`

如果输入的是eret指令，那么NPC值应当为EPC+4

端口说明

信号名称	方向	功能描述
F_PC[31:0]	输入	32位输入当前F级地址
D_PC[31:0]	输入	32位输入当前D级地址
b_jump	输入	指示b类型指令是否跳转
NPCOp[1:0]	输入	控制信号
RSS[31:0]	输入	**处理完转发后** `$rs` 寄存器保存的32位地址
NPC[31:0]	输出	32位输出次地址
EPC[31:0]	输入	32位EPC的值

控制信号说明

控制信号值	功能
`NPC_pc4`	`NPC=PC+4`
`NPC_b`	执行 `beq` 等b类指令
`NPC_j_jal`	执行 `j`，`jal` 指令
`NPC_jalr_jr`	执行 `jalr`，`jr` 指令

E级(Execute/执行)

DE_REG（D/E级流水线寄存器）

- 输入`D_PC,D_Instr,D_ext32`，此外上一级的`$rs`和`$rt`的值也要参与流水，即`D_FWD_rs,D_FWD_rt`需要参与流水，**这是由于指令序列`sw, nop, addu`的存在，`sw`在M级需要使用`$rt`的数据，但是在E级不会再进行转发（因为在D级已经转发过了），因此需要让正确的`$rt`值参与流水**
- 输出`E_PC,E_Instr,E_ext32,E_rs_data,E_rt_data`，`ALU`需要这些信息

```
module DE_REG(
    input clk,
    input reset,
    input flush,
    input WrEn,

    input Req,
    input Stall,

    input [31:0] D_PC,
    input [31:0] D_Instr,
    input [31:0] D_ext32,
    input [31:0] D_rs_data,
    input [31:0] D_rt_data,
    input D_b_jump,

    input D_DelaySlot,
    input [4:0] D_EXCCode,

    output reg [31:0] E_PC,
    output reg [31:0] E_Instr,
    output reg [31:0] E_ext32,
    output reg [31:0] E_rs_data,
    output reg [31:0] E_rt_data,
    output reg E_b_jump,

    output reg E_DelaySlot,
    output reg [4:0] E_EXCCode
    );
```

E_ALU（算术逻辑单元）

- 相比于P4，ALU做了很大的变动，添加了`ALUASel`信号选择A运算数的来源，这是为了便于扩展`sll`和`sllv`类指令的原因取消了`shamt`信号，`shamt`信号从`ALUBSel`中选择进入`ALU`中

端口说明

信号名称	方向	功能描述
A[31:0]	输入	32位输入运算数A
B[31:0]	输入	32位输入运算数B
ALUOp[4:0]	输入	控制信号
C[31:0]	输出	32位输出运算结果

控制信号说明

1. ALUOp

现在支持了所有运算指令功能，详见`def.v`文件

2. ALUASel

控制信号值	功能
`ASel_rt`	对于`sll`和`sllv`等移位指令，选择`$rt`的值
`ASel_rs`	对于其他大部分运算指令，采用`$rs`的值

3. ALUBSel

控制信号值	功能
`BSel_rt`	选择**处理完转发后**`$rt`寄存器中的值进行运算
`BSel_imm`	选择立即数进行运算
`BSel_shamt`	使用`{27'b0, E_ALUshamt}`得到32为扩展移位数
`BSel_rs`	考虑到`sllv`指令要求可变的位移数，这里可以选择`{27'b0, E_FWD_rs_data[4:0]}`，即`$rs`寄存器中的数据作为移位数

E_MDU（乘除槽）

端口说明

信号名称	方向	功能描述
clk	输入	时钟信号
reset	输入	复位信号
MDUOp[2:0]	输入	控制信号
D1[31:0]	输入	32位输入运算数A
D1[31:0]	输入	32位输入运算数B
Start	输入	开始运算的指示信号
Busy	输出	是否处于运算过程中
HI[31:0]	输出	32位HI寄存器值结果
LO[31:0]	输出	32位LO寄存器值结果

控制信号说明

1. MDUOp

控制信号值	功能
`MDU_mult`	乘法运算
`MDU_div`	除法运算
`MDU_multu`	无符号乘法运算
`MDU_divu`	无符号除法运算
`MDU_mfhi`	`mfhi`指令
`MDU_mflo`	`mflo`指令
`MDU_mthi`	`mthi`指令，把D1的值赋给HI寄存器中
`MDU_mtlo`	`mtlo`指令，把D1的值赋给LO寄存器中

M级(Memory/储存)

- 输入`E_PC`,`E_Instr`，此外上一级的ALUAns参与流水，即`E_ALUAns`,`E_ext32`需要参与流水，**这是因为`ALUAns`是待写入或读取的内存地址，另外，上一级的rt值需要参与流水**，因此还需要输入`E_FWD_rt`，**这是因为`sw`指令会向内存中写入`$rt`的数据**
- 输出`M_PC`,`M_Instr`,`M_ALUAns`,`M_DMRD`

EM_REG（E/M级流水线寄存器）

```
module EM_REG(
    input clk,
    input reset,
    input flush,
    input WrEn,

    input Req,

    input [31:0] E_PC,
    input [31:0] E_Instr,
    input [31:0] E_ALUAns,
    input [31:0] E_rt_data,
    input [31:0] E_ext32,
    input E_b_jump,
    input [31:0] E_MDUAns,

    input E_DelaySlot,
    input [4:0] E_EXCCode,
    input E_EXC_DMOv,

    output reg [31:0] M_PC,
    output reg [31:0] M_Instr,
    output reg [31:0] M_ALUAns,
    output reg [31:0] M_rt_data,
    output reg [31:0] M_ext32,
    output reg M_b_jump,
    output reg [31:0] M_MDUAns,

    output reg M_DelaySlot,
    output reg [4:0] M_EXCCode,
    output reg M_EXC_DMOv
    );
```

M_DM（数据储存器）

- `DM`已经不需要自行实现，调用`mips_txt.v`中的接口即可
- 利用BE模块处理待写入数据，使其支持按半字、字节、字储存
- 利用DE模块处理DM返回的数据，使其可以按照不同要求存入寄存器

M_BE

信号名称	方向	功能描述
BEOp[1:0]	输入	控制信号
Addr[31:0]	输入	地址信息，用于处理半字、字节
rt_data[31:0]	输入	读取的寄存器数据，待处理
DMWrEn	输入	写使能
m_data_byteen[3:0]	输出	控制写入半字、字节的位置位置
m_data_wdata[31:0]	输出	待写入数据

M_DE

信号名称	方向	功能描述
DEOp[1:0]	输入	控制信号
Addr[31:0]	输入	地址信息，用于处理半字、字节
m_data_rdata[31:0]	输入	`mips_txt.v`返回的DM中的数据
DMRD[31:0]	输出	处理之后的正确的读取数据

与接口进行交互

```
// 与DM交互
M_BE _be(
    .BEOp(M_BEOp),
    .Addr(M_ALUAns),
    .rt_data(M_FWD_rt_data),
    .m_data_byteen(m_data_byteen),
    .m_data_wdata(m_data_wdata)
);

assign m_inst_addr = M_PC;
assign m_data_addr = M_ALUAns;

wire [31:0] M_DMRD;
M_DE _de(
    .DEOp(M_DEOp),
    .Addr(M_ALUAns),
    .m_data_rdata(m_data_rdata),
    .DMRD(M_DMRD)
);

// 正确输出GRF读写信息
assign w_grf_addr = W_GRFA3;
assign w_grf_wdata = W_GRFWD;
assign w_grf_we = W_GRFWrEn;
assign w_inst_addr = W_PC;
```

此外CP0放在了M级

W级(Write/回写)

- W级事实上与D级重合了，但是仍然需要处理向E,M级的转发

MW_REG（M/W级流水线寄存器）

```
module MW_REG(
    input clk,
    input reset,
    input flush,
    input WrEn,
    input Req,

    input [31:0] M_PC,
    input [31:0] M_Instr,
    input [31:0] M_ALUAns,
    input [31:0] M_DMRD,
    input M_b_jump,
    input [31:0] M_MDUAns,
    input [31:0] M_CP0Out,

    output reg [31:0] W_PC,
    output reg [31:0] W_Instr,
    output reg [31:0] W_ALUAns,
    output reg [31:0] W_DMRD,
    output reg W_b_jump,
    output reg [31:0] W_MDUAns,
    output reg [31:0] W_CP0Out
    );
```

数据通路分析

由于指令很多，采用分类的方法，同一类型的指令共享相似的数据通路

CTRL在每一级都会进行译码，即采用分布式译码

D级(Decode/译码)

```
// D级Decode
control D_CTRL(
    .Instr(D_Instr),

    .rs(D_rs_addr),
    .rt(D_rt_addr),
    .imm16(D_imm16),
    .imm26(D_imm26),
    .CMPOp(D_CMPOp),
    .NPCOp(D_NPCOp),
    .EXTOp(D_EXTOp)
);
```

E级(Execute/执行)

```
// E级Execute
control E_CTRL(
    .Instr(E_Instr),

    .rs(E_rs_addr),
    .rt(E_rt_addr),
    .shamt(E_ALUshamt),

    .ALUOp(E_ALUOp),
    .ALUASel(E_ALUASel),
    .ALUBSel(E_ALUBSel),
    .GRFA3(E_GRFA3),
    .GRFWDSel(E_GRFWDSel)
);
```

M级(Memory/储存)

```
control M_CTRL(
    .Instr(M_Instr),

    .rt(M_rt_addr),
    .DMOp(M_DMOp),
    .DMWrEn(M_DMWrEn),
    .GRFA3(M_GRFA3),
    .GRFWDSel(M_GRFWDSel)
);
```

W级(Write/回写)

```
control W_CTRL(
    .Instr(W_Instr),

    .GRFA3(W_GRFA3),
    .GRFWDSel(W_GRFWDSel)
);
```

冲突处理方法

转发

对于转发，我采用的是暴力转发的方法，如果不阻塞就意味着一定能够在使用该寄存器的值之前获得最新的且正确的值，那么之前转发的错误的值不用去管它，最后总能得到一个正确的值去覆盖原先错误的值

W-D级转发在寄存器内部实现

```
assign D_FWD_rs_data = (D_rs_addr == 0) ? 0 :
                       (D_rs_addr == E_GRFA3) ? E_GRFWD :
                       (D_rs_addr == M_GRFA3) ? M_GRFWD :
                       D_rs_data;

assign D_FWD_rt_data = (D_rt_addr == 0) ? 0 :
                       (D_rt_addr == E_GRFA3) ? E_GRFWD :
                       (D_rt_addr == M_GRFA3) ? M_GRFWD :
                       D_rt_data;
```

```
assign E_FWD_rs_data = (E_rs_addr == 0) ? 0 :
                       (E_rs_addr == M_GRFA3) ? M_GRFWD :
                       (E_rs_addr == W_GRFA3) ? W_GRFWD :
                       E_rs_data;

assign E_FWD_rt_data = (E_rt_addr == 0) ? 0 :
                       (E_rt_addr == M_GRFA3) ? M_GRFWD :
                       (E_rt_addr == W_GRFA3) ? W_GRFWD :
                       E_rt_data;
```

```
assign M_FWD_rt_data = (M_rt_addr == 0) ? 0 :
                       (M_rt_addr == W_GRFA3) ? W_GRFWD :
                       M_rt_data;
```

阻塞

阻塞需要添加一行，关于`eret`的阻塞

对于阻塞的处理，我直接采用教程中$Tuse$和$Tnew$进行判断的方法，设计了`stall_CTRL`模块，专门负责处理阻塞时流水线寄存器的`flush`和`WrEn`信号

我的设计只在D级进行阻塞，阻塞控制器接受D，E，M级的指令输入，处理分析指令类别，并给他们赋上不同的$Tuse$和$Tnew$的值，然后用组合逻辑判断，如果$Tuse < Tnew$就直接阻塞D级，知道$Tuse = Tnew$时再继续执行

```
wire E_stall_rs = (E_GRFA3 == D_rs_addr && D_rs_addr != 0) && (E_Tnew > D_Tuse_rs);
wire E_stall_rt = (E_GRFA3 == D_rt_addr && D_rt_addr != 0) && (E_Tnew > D_Tuse_rt);

wire M_stall_rs = (M_GRFA3 == D_rs_addr && D_rs_addr != 0) && (M_Tnew > D_Tuse_rs);
wire M_stall_rt = (M_GRFA3 == D_rt_addr && D_rt_addr != 0) && (M_Tnew > D_Tuse_rt);

assign Stall = E_stall_rs | E_stall_rt | M_stall_rs | M_stall_rt;

assign FD_REG_WrEn = !Stall;
assign DE_REG_WrEn = 1'b1;
assign EM_REG_WrEn = 1'b1;
assign MW_REG_WrEn = 1'b1;

assign PC_WrEn = !Stall;

assign FD_REG_Flush = 1'b0;
assign DE_REG_Flush = Stall;
assign EM_REG_Flush = 1'b0;
assign MW_REG_Flush = 1'b0;
```

Bridge

系统桥是处理CPU与外设（两个计时器）之间信息交互的通道

CPU中store类指令需要储存的数据经过BE处理后会通过m_data_addr, m_data_byteen, m_data_wdata三个信号输出到桥中，桥会根据写使能m_data_byteen和地址m_data_addr来判断到底写的是内存还是外设，然后给出正确的写使能

load类指令则是全部把地址传递给每个外设和DM中，然后桥根据地址选择从应该反馈给CPU从哪里读出来的数据，然后DE在处理读出的数据，反馈正确的结果

Bridge的端口列表如下：

```
module Bridge(
    output [31:0] m_data_addr,
    output [31:0] m_data_wdata,
    output [3:0] m_data_byteen,
    input [31:0] m_data_rdata,

    input [31:0] tmp_m_data_addr,
    input [31:0] tmp_m_data_wdata,
    input [3:0] tmp_m_data_byteen,
    output [31:0] tmp_m_data_rdata,

    output [31:0] TC0_Addr,
    output TC0_WE,
    output [31:0] TC0_Din,
    input [31:0] TC0_Dout,

    output [31:0] TC1_Addr,
    output TC1_WE,
    output [31:0] TC1_Din,
    input [31:0] TC1_Dout
    );
```

顶层Mips微系统

最后mips.v中实例化CPU，Bridge，TC0和TC1三个模块相互交互

利用这样两句实现写0x7f20，停止中断使能

```
assign m_data_addr = (TestIntResponse && interrupt) ? 32'h0000_7f20 : bridge_m_data_addr;
assign m_data_byteen = (TestIntResponse && interrupt) ? 1 : bridge_m_data_byteen;
```

测试方法和工具

没有自动评测工具，通过对特定的异常和中断编写程序进行测试

取指异常

```
.text

li $28, 0
li $29, 0

# jr PC mod 4 not 0
la $1, label1
la $2, label1
addiu $1, $1, 1
jr $1
nop
label1:
```

```
# jr PC < 0x3000
li $1, 0x2996
la $2, label2
jr $1
nop
label2:

# jr PC > 0x4ffc
li $1, 0x4fff
la $2, label3
jr $1
nop
label3:

end:j end

.ktext 0x4180
mfc0 $12, $12
mfc0 $13, $13
mfc0 $14, $14
mtc0 $2, $14
eret
ori $1, $0, 0
```

其它异常

```
.text

li $28, 0
li $29, 0

lw $1, 1($0)
lh $1, 1($0)
lhu $1, 1($0)

lh $1, 0x7f00($0)
lhu $1, 0x7f04($0)
lb $1, 0x7f08($0)
lbu $1, 0x7f10($0)
lb $1, 0x7f14($0)
lb $1, 0x7f18($0)

li $2, 0x7fffffff
lw $1, 1($2)
lh $1, 1($2)
lhu $1, 1($2)
lb $1, 1($2)
lbu $1, 1($2)

lw $1, 0x3000($0)
lh $1, 0x4000($0)
lhu $1, 0x6000($0)
lb $1, 0x7f0c($0)
lbu $1, 0x7f1c($0)

sw $1, 1($0)
```

```
sh $1, 1($0)

sh $1, 0x7f00($0)
sb $1, 0x7f04($0)
sh $1, 0x7f08($0)
sb $1, 0x7f10($0)
sh $1, 0x7f14($0)
sb $1, 0x7f18($0)

li $2, 0x7fffffff
sw $1, 1($2)
sh $1, 1($2)
sb $1, 1($2)

sw $1, 0x7f08($0)
sh $1, 0x7f08($0)
sb $1, 0x7f08($0)
sw $1, 0x7f18($0)
sh $1, 0x7f18($0)
sb $1, 0x7f18($0)

sw $1, 0x3000($0)
sh $1, 0x4000($0)
sh $1, 0x6000($0)
sb $1, 0x7f0c($0)
sb $1, 0x7f1c($0)

msub $1, $2

li $1, 0x7fffffff
add $1, $1, $1
addi $1, $1, 1
li $1, 0x80000000
add $1, $1, $1
addi $1, $1, -1
sub $1, $1, $2
sub $1, $2, $1

end:j end


.ktext 0x4180
mfc0 $12, $12
mfc0 $13, $13
mfc0 $14, $14
addi $14, $14, 4
mtc0 $14, $14
eret
ori $1, $0, 0
```

计时器功能测试

```
.text
li $12, 0x0c01
mtc0 $12, $12

li $1, 500
```

```
li $2, 9

sw $1, 0x7f04($0)
sw $2, 0x7f00($0)
li $1, 1000
sw $1, 0x7f14($0)
sw $2, 0x7f10($0)

lw $1, 0x7f00($0)
lw $1, 0x7f04($0)
lw $1, 0x7f10($0)
lw $1, 0x7f14($0)

li $1, 0
li $2, 0

for:
ori $3, $3, 0
beq $1, $0, for
nop
beq $2, $0, for
nop

lw $1, 0x7f00($0)
lw $1, 0x7f04($0)
lw $1, 0x7f10($0)
lw $1, 0x7f14($0)

end:j end

.ktext 0x4180
mfc0 $13, $13
li $15, 0x7fffffff
and $13, $13, $15
li $14, 1024
beq $13, $14, timer0
nop
li $14, 2048
beq $13, $14, timer1
nop
eret

timer0:
li $1, 1
sw $0, 0x7f00($0)
eret

timer1:
li $2, 2
sw $0, 0x7f10($0)
eret
```

延迟槽异常测试

```
.text

li $28, 0
li $29, 0

li $1, 1
bne $0, $0, end
lw $1, 1($0)
li $1, 1
bne $0, $0, end
lh $1, 1($0)
li $1, 1
bne $0, $0, end
lhu $1, 1($0)

li $1, 1
bne $0, $0, end
lh $1, 0x7f00($0)
li $1, 1
bne $0, $0, end
lhu $1, 0x7f04($0)
li $1, 1
bne $0, $0, end
lb $1, 0x7f08($0)
li $1, 1
bne $0, $0, end
lbu $1, 0x7f10($0)
li $1, 1
bne $0, $0, end
lb $1, 0x7f14($0)
li $1, 1
bne $0, $0, end
lb $1, 0x7f18($0)

li $2, 0x7fffffff
li $1, 1
bne $0, $0, end
lw $1, 1($2)
li $1, 1
bne $0, $0, end
lh $1, 1($2)
li $1, 1
bne $0, $0, end
lhu $1, 1($2)
li $1, 1
bne $0, $0, end
lb $1, 1($2)
li $1, 1
bne $0, $0, end
lbu $1, 1($2)

li $1, 1
bne $0, $0, end
lw $1, 0x3000($0)
li $1, 1
```

```
bne $0, $0, end
lh $1, 0x4000($0)
li $1, 1
bne $0, $0, end
lhu $1, 0x6000($0)
li $1, 1
bne $0, $0, end
lb $1, 0x7f0c($0)
li $1, 1
bne $0, $0, end
lbu $1, 0x7f1c($0)

li $1, 1
bne $0, $0, end
sw $1, 1($0)
li $1, 1
bne $0, $0, end
sh $1, 1($0)

li $1, 1
bne $0, $0, end
sh $1, 0x7f00($0)
li $1, 1
bne $0, $0, end
sb $1, 0x7f04($0)
li $1, 1
bne $0, $0, end
sh $1, 0x7f08($0)
li $1, 1
bne $0, $0, end
sb $1, 0x7f10($0)
li $1, 1
bne $0, $0, end
sh $1, 0x7f14($0)
li $1, 1
bne $0, $0, end
sb $1, 0x7f18($0)

li $2, 0x7fffffff
li $1, 1
bne $0, $0, end
sw $1, 1($2)
li $1, 1
bne $0, $0, end
sh $1, 1($2)
li $1, 1
bne $0, $0, end
sb $1, 1($2)

li $1, 1
bne $0, $0, end
sw $1, 0x7f08($0)
li $1, 1
bne $0, $0, end
sh $1, 0x7f08($0)
li $1, 1
bne $0, $0, end
sb $1, 0x7f08($0)
```

```
li $1, 1
bne $0, $0, end
sw $1, 0x7f18($0)
li $1, 1
bne $0, $0, end
sh $1, 0x7f18($0)
li $1, 1
bne $0, $0, end
sb $1, 0x7f18($0)

li $1, 1
bne $0, $0, end
sw $1, 0x3000($0)
li $1, 1
bne $0, $0, end
sh $1, 0x4000($0)
li $1, 1
bne $0, $0, end
sh $1, 0x6000($0)
li $1, 1
bne $0, $0, end
sb $1, 0x7f0c($0)
li $1, 1
bne $0, $0, end
sb $1, 0x7f1c($0)

li $1, 1
bne $0, $0, end
msub $1, $2



li $1, 0x7fffffff
li $11, 1
bne $0, $0, end
add $1, $1, $1
li $11, 1
bne $0, $0, end
addi $1, $1, 1
li $1, 0x80000000
li $11, 1
bne $0, $0, end
add $1, $1, $1
li $11, 1
bne $0, $0, end
addi $1, $1, -1
li $11, 1
bne $0, $0, end
sub $1, $1, $2
li $11, 1
bne $0, $0, end
sub $1, $2, $1

end:j end


.ktext 0x4180
mfc0 $12, $12
```

```
mfc0 $13, $13
mfc0 $14, $14
addi $14, $14, 8
mtc0 $14, $14
eret
ori $1, $0, 0
```

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/软件接口”是指令（机器码）。硬件实现了一些功能，并按照规约可以被相应的指令所操控。软件通过规约使用相应的指令操控硬件完成相应的功能，从而达到软件所期望的效果。指令在这个过程中实现了硬件软件的对接，因此是“硬件/软件接口”。

2. **BE 部件对所有的外设都是必要的吗？**

   不是，只有对**字节/半字**有存取需求的才有必要。

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。**

   见计时器说明文档。

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
   li $12, 0x0401
   mtc0 $12, $12
   li $1, 100
   li $2, 9
   sw $1, 0x7f04($0)
   sw $2, 0x7f00($0)

   dead_loop:
   j dead_loop
   nop

   .ktext 0x4180
   li $1, 100
   li $2, 9
   sw $1, 0x7f04($0)
   sw $2, 0x7f00($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？**

   鼠标和键盘产生中断信号，进入中断处理区的对应位置，将输入信号从鼠标和键盘中读入寄存器。