# P7 流水线CPU设计文档

## 设计概述

- 设计的处理器为32位五级流水线处理器
- 处理器支持的指令集为

```
add, sub, and, or, slt, sltu, lui
addi, andi, ori
lb, lh, lw, sb, sh, sw
mult, multu, div, divu, mfhi, mflo, mthi, mtlo
beq, bne, jal, jr,
mfc0, mtc0, eret, syscall
```

- 在 P6 基础上新增了 `mfc0, mtc0, eret, syscall` 四条新指令
- `eret` 具有跳转的功能但是没有延迟槽，保证 `eret` 的后续指令不被执行
- `syscall` 指令行为与 MARS 不同，无需实现特定的输入输出功能，只需直接产生异常并进入内核态

CPU整体架构参考了《数字设计与计算机体系结构》图7-58。

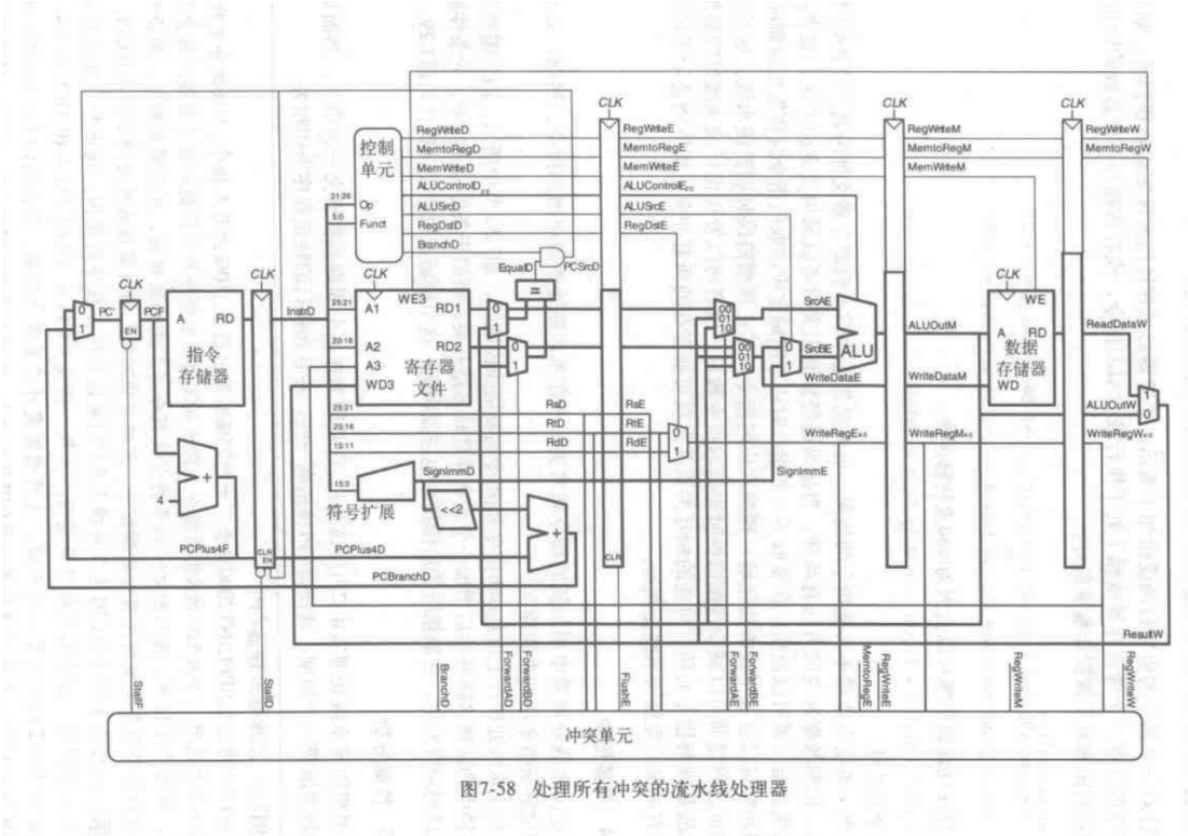
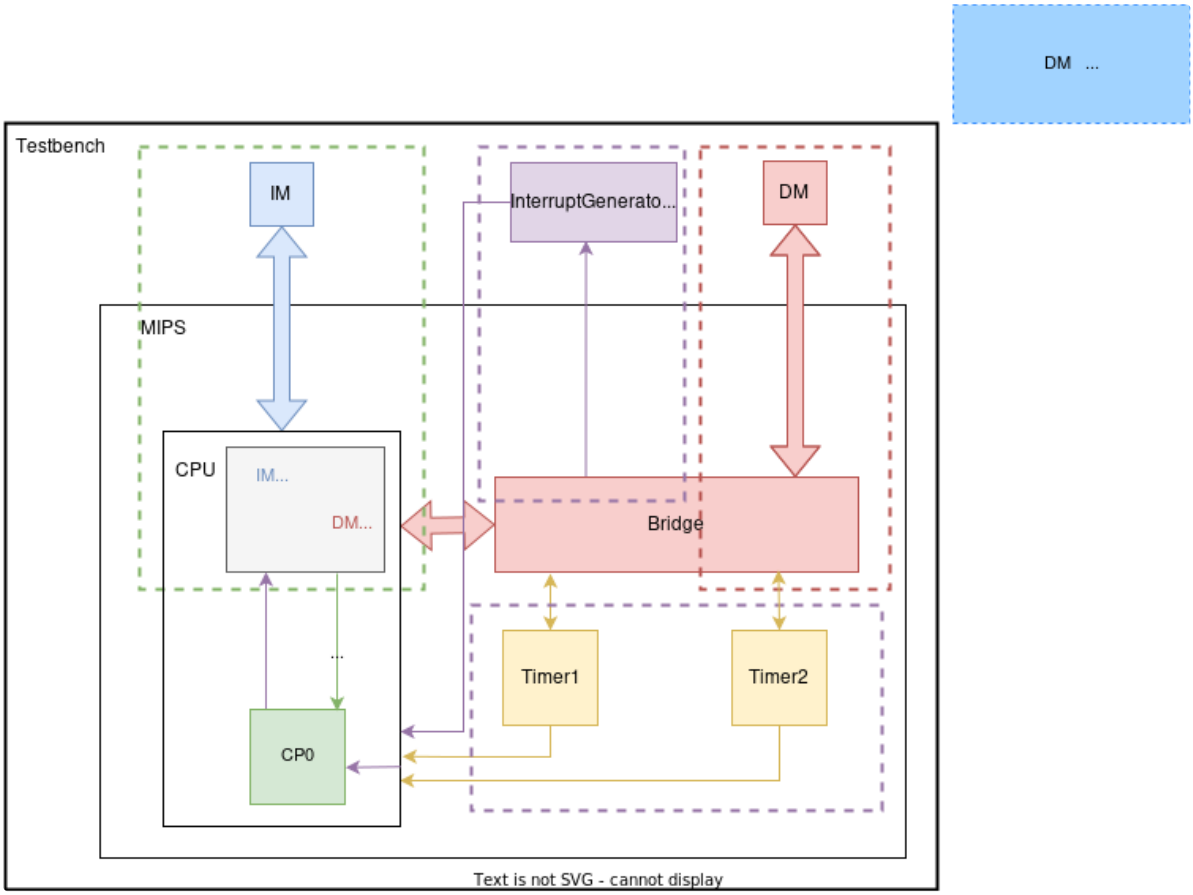



图7-58 处理所有冲突的流水线处理器

mips微系统的架构图如下图所示：




# F级：取指令（Fetch）

本级的输入为来自D级的`next_pc`，用于更新下一个PC的值。

本级的输出为`F_PC`和`F_Instr`，分别对应从F级指令的PC和F级指令的内容，均需要参与流水。

## F_IFU

只负责PC的存储与更新，F_instr来自`mips_txt.v`的交互。

| 信号名 | 方向 | 描述 |
| --- | --- | --- |
| clk | I | 时钟信号 |
| reset | I | **同步**复位信号，高电平有效 |
| enable | I | PC写使能信号，高电平有效 |
| next_pc[31:0] | I | 待更新的指令地址 |
| pc[31:0] | O | 当前指令地址 |

### 与`mips_txt.v`交互

```
ifu F_IFU(
    .clk(clk),
    .reset(reset),
    .enable(IFU_WE),
    .next_pc(next_PC),
    .pc(F_PC)
);

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

### F/D级流水线寄存器

| 信号名 | 方向 | 功能描述 |
| --- | --- | --- |
| clk | I | 时钟信号 |
| reset | I | **同步**复位信号，高电平有效 |
| flush | I | 寄存器刷新信号，高电平有效，发生阻塞时使用 |
| enable | I | 写使能信号，高电平有效 |
| F_pc[31:0] | I | F级PC |
| F_instr[31:0] | I | 时钟信号 |
| D_pc[31:0] | O | D级PC |
| D_instr[31:0] | O | 32位的指令值 |

## D级：译码（Decode）

本级的输入为来自F级的`F_PC`和`F_Instr`。

本级的输出为`D_gpr_rs`，`D_gpr_rt`，`D_extres`，`D_PC`，`D_Instr`和`next_pc`。

本级涉及到来自E级、M级、W级的转发，其中来自W级的转发通过**GRF内部转发**的方式实现。

`$rs`和`$rt`的值在本级转发成`D_fwd_gprrs`和`D_fwd_gprrt`，和`D_extres`，`D_PC`，`D_Instr`参与流水。

本级需要对**此级指令**的$Tuse$和此时**E级**指令与**M级指令**的$Tnew$进行比较，从而确定是否执行阻塞。

> $Tuse$和$Tnew$：
>
> - $Tuse$表示这条指令位于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。
>   - 每个指令的$Tuse$是固定不变的
>   - 一个指令可以有两个$Tuse$值
> - $Tnew$表示位于**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。
>   - $Tnew$是一个动态值，每个指令处于流水线不同阶段有不同的$Tnew$值
>   - 一个指令在一个时刻至多有一个$Tnew$值（一个指令至多写一个寄存器）
> - 当$Tuse \geqslant Tnew$，说明需要的数据可以及时算出，可以通过**转发**来解决

> 当$Tuse < Tnew$，说明需要的数据不能及时算出，必须**阻塞**流水线解决

## D_GRF

| 信号名 | 方向 | 描述 |
| --- | --- | --- |
| clk | I | 时钟信号 |
| reset | I | **同步**复位信号，高电平有效 |
| A1[4:0] | I | 5位地址输入信号，指定32个寄存器中的一个，将其中存储的数据读出到RD1 |
| A2[4:0] | I | 5位地址输入信号，指定32个寄存器中的一个，将其中存储的数据读出到RD2 |
| A3[4:0] | I | 5位地址输入信号，指定32个寄存器中的一个将WD中的数据写入 |
| WD[31:0] | I | 32位数据输入信号 |
| WPC[31:0] | I | 写入寄存器时对应的指令PC值 |
| RD1[31:0] | O | 输出A1指定的寄存器中的32位数据 |
| RD2[31:0] | O | 输出A2指定的寄存器中的32位数据 |

- 如果不需要写寄存器，只需要把A3Sel设为0即可。
- 此处**WPC**和**WD**均来自**W级**。

### 控制信号

| WDSel | 操作 |
| --- | --- |
| WDSel_aluans | WD来自ALU的运算结果 |
| WDSel_dmrd | WD来自DM的输出RD |
| WDSel_PCa8 | WD为**当前流水线层级的PC** + 8 |

## D_NPC

| 信号名 | 方向 | 描述 |
| --- | --- | --- |
| F_pc[31:0] | I | 当前F级PC的值 |
| D_pc[31:0] | I | 当前D级PC的值 |
| PCSel[1:0] | I | 指定更新PC的方式 |
| branch | I | branch类型指令**是否达到跳转条件**，高电平有效 |
| imm[25:0] | I | j指令和jal指令中的立即数，即D_Instr[25:0] |
| offset[15:0] | I | branch类型指令的偏移量，即D_Instr[15:0] |
| ra[31:0] | I | **完成转发后** $rs 寄存器保存的地址值 |
| next_pc[31:0] | O | 下一指令的PC |

### 控制信号

| PCSel | 操作 |
| --- | --- |
| PCSel_PCa4 | PC ← PC + 4 |
| PCSel_branch | PC ← PC + 4 + sign_extend(offset\|\|$0^2$>) |
| PCSel_j | PC ← PC[31:28] \|\| imm \|\| $0^2$ |
| PCSel_jr | PC ← GPR[rs] |

## D_CMP

| 信号名 | 方向 | 描述 |
| --- | --- | --- |
| gpr_rs[31:0] | I | **完成转发后** $rs 寄存器中的值 |
| gpr_rt[31:0] | I | **完成转发后** $rt 寄存器中的值 |
| CMPOp[1:0] | I | 指定比较数据的方式 |
| flag | O | 是否满足所设条件，高电平有效 |

### 控制信号

| CMPOp | 操作 |
| --- | --- |
| CMP_beq | beq指令：若 GPR[rs] == GPR[rt]，则flag置1，否则置0 |

## D_EXT

| 信号名 | 方向 | 描述 |
| --- | --- | --- |
| num[15:0] | I | 16位需要扩展的立即数 |
| EXTOp | I | 指定进行扩展的方式 |
| result[31:0] | O | 32位完成扩展的立即数 |

### 控制信号

| EXTOp | 操作 |
| --- | --- |
| EXT_zero | result = zero_extend(num) |
| EXT_signed | result = sign_extend(num) |

### D/E级流水线寄存器

| 信号名 | 方向 | 功能描述 |
| --- | --- | --- |
| clk | I | 时钟信号 |
| reset | I | **同步**复位信号，高电平有效 |
| flush | I | 寄存器刷新信号，高电平有效，发生阻塞时使用 |
| enable | I | 写使能信号，高电平有效 |
| D_pc[31:0] | I | D级PC |
| D_instr[31:0] | I | 32位指令值 |
| D_extres[31:0] | I | 16位立即数扩展的结果 |
| D_gpr_rs[31:0] | I | GPR[rs] |
| D_gpr_rt[31:0] | I | GPR[rt] |
| E_pc[31:0] | O | E级PC |
| E_instr[31:0] | O | 32位指令值 |
| E_gpr_rs[31:0] | O | GPR[rs] |
| E_gpr_rt[31:0] | O | GPR[rt] |

## E级：执行（Execute）

本级的输入为`D_PC`，`D_Instr`，`D_extres`，`D_gpr_rs`，`D_gpr_rt`。

本级的输出为`E_PC`，`E_Instr`，`E_aluans`，`E_gpr_rt`，`E_grfWD`。

本级涉及到来自M级、W级的转发，`$rt`的值在本级转发得`D_fwd_gprrt`，和`E_PC`，`E_Instr`，`E_aluans`参与流水。

### E_ALU

| 信号名 | 方向 | 描述 |
| --- | --- | --- |
| A[31:0] | I | 32位运算数 |
| B[31:0] | I | 32位运算数 |
| ALUOp[2:0] | I | 指定ALU进行的计算 |
| result[31:0] | O | 32位运算结果 |

#### 控制信号

详见`def.v`文件中的定义。

## E_MDU（乘除槽）

| 信号名 | 方向 | 功能描述 |
| --- | --- | --- |
| clk | I | 时钟信号 |
| reset | I | **同步**复位信号，高电平有效 |
| MDUOp[2:0] | I | 指定乘除槽进行的操作 |
| gpr_rs[31:0] | I | GPR[rs] |
| gpr_rt[31:0] | I | GPR[rt] |
| start | I | 指定乘除槽是否开始计算，高电平有效 |
| busy | O | 乘除槽是否处于运算过程中 |
| HI[31:0] | O | 32位HI寄存器值结果 |
| LO[31:0] | O | 32位LO寄存器值结果 |

### 控制信号

| MDUOp | 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gpr_rs的值赋给HI寄存器中 |
| MDU_mtlo | mtlo指令，把gpr_rs的值赋给LO寄存器中 |

## E/M级流水线寄存器

| 信号名 | 方向 | 功能描述 |
| --- | --- | --- |
| clk | I | 时钟信号 |
| reset | I | **同步**复位信号，高电平有效 |
| flush | I | 寄存器刷新信号，高电平有效，发生阻塞时使用 |
| enable | I | 写使能信号，高电平有效 |
| E_pc[31:0] | I | E级PC |
| E_instr[31:0] | I | 32位指令值 |
| E_aluans[31:0] | I | ALU的运算结果 |
| E_gpr_rt[31:0] | I | GPR[rt] |
| M_pc[31:0] | O | M级PC |
| M_instr[31:0] | O | 32位指令值 |
| M_aluans[31:0] | O | ALU的运算结果 |
| M_gpr_rt[31:0] | O | GPR[rt] |

# M级：存储器（Memory）

本级的输入为`E_PC`，`E_Instr`，`E_aluans`，`E_gpr_rt`。

本级的输出为`M_PC`，`M_Instr`，`M_aluans`，`M_dmrd`，`M_grfWD`。

本级涉及到来自M级、W级的转发，`$rt`的值在本级转发得`M_fwd_gprrt`。

本机参与流水的有`M_PC`，`M_Instr`，`M_aluans`，`M_dmrd`。

## M_DM

本次实验只需要调用调用`mips_txt.v`中的接口即可，无需自行实现DM。

使用`fromRAM`模块处理DM返回的数据，使其符合写入寄存器的要求。

使用`toRAM`模块处理写入DM的数据，支持按照字、半字、字节的模式储存进DM。

### M_fromRAM

| 信号名 | 方向 | 功能描述 |
| --- | --- | --- |
| A[31:0] | I | 进行写操作的地址 |
| DMOp[1:0] | I | 指定模块进行的操作 |
| m_data_rdata[31:0] | I | 从`mips_txt.v`中的DM读出的数据 |
| RD | O | 处理后的正确的读取数据 |

### M_toRAM

| 信号名 | 方向 | 功能描述 |
| --- | --- | --- |
| A[31:0] | I | 进行读操作的地址 |
| gpr_rt[31:0] | I | 读取的待处理的寄存器数据 |
| DMOp[1:0] | I | 指定模块进行的操作 |
| MemWrite | I | 写使能信号，高电平有效 |
| m_data_byteen[3:0] | O | 控制写入数据在DM中的位置 |
| m_data_wdata[31:0] | O | 处理后的正确的待写入数据 |

### 控制信号

| DMOp | 操作 |
| --- | --- |
| DM_word | 读/写整个字，对应lw和sw指令 |
| DM_halfword | 读/写半个字，对应lh和sh指令 |
| DM_byte | 读/写一个字节，对应lb和sb指令 |

### 与mips_txt.v交互

```
wire [31:0] M_dmrd;
assign m_inst_addr = M_PC;
assign m_data_addr = M_aluans;

toRAM M_Store(
    .A(M_aluans),
    .gpr_rt(M_fwd_gprrt),
    .DMOp(M_DMOp),
    .MemWrite(M_MemWrite),
    .m_data_byteen(m_data_byteen),
    .m_data_wdata(m_data_wdata)
);

fromRAM M_Load(
    .A(M_aluans),
    .DMOp(M_DMOp),
    .m_data_rdata(m_data_rdata),
    .RD(M_dmrd)
);

assign w_grf_wdata = W_grfWD;
assign w_inst_addr = W_PC;
assign w_grf_addr = W_A3Sel;
```

## CP0协处理器

考虑到宏观PC的处理，将CP0置于M级较为合理。

将**异常码 ExcCode 、是否处于延迟槽中的判断信号 BDIn 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 0x00004180 ，然后向F级NPC的NPC也被置为 0x00004180 ，下一条指令从 0x00004180 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 IntResponse 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 0x00007f20 地址，从而时外部中断信号停止。

> **宏观PC**表示整个 CPU “宏观”运行指令所对应的PC地址。
>
> 所谓“宏观”指令，表示该指令之前的所有指令序列对CPU的更新已完成，该指令及其之后的指令序列对 CPU 的更新未完成。

| 信号名 | 方向 | 功能描述 |
| --- | --- | --- |
| clk | I | 时钟信号 |
| reset | I | **同步**复位信号，高电平有效 |
| enable | I | 写使能信号，高电平有效 |
| CP0Add[4:0] | I | CP0中寄存器的地址 |
| CP0In[31:0] | I | 写入寄存器的32位数据 |
| VPC[31:0] | I | 受害PC |
| BDIn | I | 是否为延迟槽内指令，高电平有效 |
| ExcCodeIn[4:0] | I | 异常码 |
| HWInt[5:0] | I | 外部硬件中断信号 |
| EXLClr | I | 将SR的Exl置0，高电平有效 |
| CP0Out[31:0] | O | 输出地址为 CP0Add 的寄存器的值 |
| EPCOut[31:0] | O | 输出当前 EPC 的值 |
| Req | O | 是否响应中断请求 |
| IntResponse | O | 是否响应 中断发生器 的中断请求 |

### 寄存器

| 寄存器 | 编号 | 功能 |
| --- | --- | --- |
| SR | 12 | 配置异常的功能 |
| Cause | 13 | 记录异常发生的原因和情况 |
| EPC | 14 | 记录异常处理结束后需要返回的 PC |

### 功能域

| 寄存器 | 功能域 | 位域 | 解释 |
| --- | --- | --- | --- |
| SR (State Register) | IM (Interrupt Mask) | 15:10 | 分别对应六个外部中断，相应位置 1 表示允许中断，置 0 表示禁止中断。这是一个被动的功能，只能通过 mtc0 这个指令修改，通过修改这个功能域，我们可以屏蔽一些中断。 |
| SR (State Register) | EXL (Exception Level) | 1 | 任何异常发生时置位，这会强制进入核心态（也就是进入异常处理程序）并禁止中断。 |
| SR (State Register) | IE (Interrupt Enable) | 0 | 全局中断使能，该位置 1 表示允许中断，置 0 表示禁止中断。 |
| Cause | BD (Branch Delay) | 31 | 当该位置 1 的时候，EPC 指向当前指令的前一条指令（一定为跳转），否则指向当前指令。 |
| Cause | IP (Interrupt Pending) | 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，将会每个周期被修改一次，修改的内容来自计时器和外部中断。 |
| Cause | ExcCode | 6:2 | 异常编码，记录当前发生的是什么异常。 |
| EPC | - | - | 记录异常处理结束后需要返回的 PC。 |

### 异常码

| 异常与中断码 | 助记符与名称 | 指令与指令类型 | 描述 |
| --- | --- | --- | --- |
| 0 | Int<br>(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<br>(取指异常) | 所有指令 | PC 地址未字对齐。 |
| | | | PC 地址超过 0x3000 ~ 0x6ffc。 |
| | AdEL<br>(取数异常) | lw | 取数地址未与 4 字节对齐。 |
| | | lh | 取数地址未与 2 字节对齐。 |
| | | lh, lb | 取 Timer 寄存器的值。 |
| | | load 型指令 | 计算地址时加法溢出。 |
| | | load 型指令 | 取数地址超出 DM、Timer0、Timer1、中断发生器的范围。 |
| 5 | AdES<br>(存数异常) | sw | 存数地址未 4 字节对齐。 |
| | | sh | 存数地址未 2 字节对齐。 |
| | | sh, sb | 存 Timer 寄存器的值。 |
| | | store 型指令 | 计算地址加法溢出。 |
| | | store 型指令 | 向计时器的 Count 寄存器存值。 |
| | | store 型指令 | 存数地址超出 DM、Timer0、Timer1、中断发生器的范围。 |
| 8 | Syscall<br>(系统调用) | syscall | 系统调用。 |
| 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。 |

## M/W级流水线寄存器

| 信号名 | 方向 | 功能描述 |
| --- | --- | --- |
| clk | I | 时钟信号 |
| reset | I | **同步**复位信号，高电平有效 |
| flush | I | 寄存器刷新信号，高电平有效，发生阻塞时使用 |
| enable | I | 写使能信号，高电平有效 |
| M_pc[31:0] | I | M级PC |
| M_instr[31:0] | I | 32位指令值 |
| M_aluans[31:0] | I | ALU的运算结果 |
| M_dmrd[31:0] | I | 从DM中读取的值 |
| W_pc[31:0] | O | W级PC |
| W_instr[31:0] | O | 32位指令值 |
| W_aluans[31:0] | O | ALU的运算结果 |
| W_dmrd[31:0] | O | 从DM中读取的值 |

# W级：写回（Writeback）

本级与D级是重合的，需要处理向E级和M级的转发。

# 转发

采用**暴力转发**的方式。由AT法的分析，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。因此采用暴力转发总能得到一个正确的值去覆盖原先错误的值。

```
// D级转发
assign D_fwd_gprrs = (D_rs == 5'd0) ? (32'd0) :
                     (D_rs == E_A3Sel) ? E_grfWD :
                     (D_rs == M_A3Sel) ? M_grfWD : D_gpr_rs;

assign D_fwd_gprrt = (D_rt == 5'd0) ? (32'd0) :
                     (D_rt == E_A3Sel) ? E_grfWD :
                     (D_rt == M_A3Sel) ? M_grfWD : D_gpr_rt;


// E级转发
assign E_grfWD = (E_WDSel == `WDSel_PCa8) ? (E_PC + 8) : 32'd0; // 不能对功能部件输出进行转发
assign E_fwd_gprrs = (E_rs == 5'd0) ? 32'd0 :
                     (E_rs == M_A3Sel) ? M_grfWD :
                     (E_rs == W_A3Sel) ? W_grfWD :
                     E_gpr_rs;

assign E_fwd_gprrt = (E_rt == 5'd0) ? 32'd0 :
                     (E_rt == M_A3Sel) ? M_grfWD :
                     (E_rt == W_A3Sel) ? W_grfWD :
                     E_gpr_rt;
```

```
22
23
24  // M级转发
25  assign M_grfWD = (M_WDSel == `WDSel_aluans) ? M_aluans :
26                   (M_WDSel == `WDSel_mduans) ? M_mduans :
27                   (M_WDSel == `WDSel_PCa8) ? (M_PC + 8) : 32'd0;
28  assign M_fwd_gprrt = (M_rt == 5'd0) ? (5'd0) :
29                       (M_rt == W_A3Sel) ? W_grfWD :
30                       M_gpr_rt;
31
32
33  // W级转发
34  assign W_grfWD = (W_WDSel == `WDSel_aluans) ? W_aluans :
35                   (W_WDSel == `WDSel_dmrd) ? W_dmrd :
36                   (W_WDSel == `WDSel_mduans) ? W_mduans :
37                   (W_WDSel == `WDSel_PCa8) ? (W_PC + 8) : 32'd0;
```

# 阻塞

使用组合逻辑，判断每一级中指令的$Tuse$和$Tnew$。

如果有$Tuse$ < $Tnew$，就执行阻塞。**只可能在D级进行阻塞。**

```
1   // stall_handle.v
2   assign D_Tuse_rs = (D_branch | D_j2r) ? 3'd0 :
3                      (D_ic | D_rc | D_load | (D_store && !D_shift_s) | D_mt |
    D_md) ? 3'd1 : 3'd3;
4
5   assign D_Tuse_rt = (D_branch) ? 3'd0 :
6                      (D_rc | D_md) ? 3'd1 :
7                      (D_store | D_mtc0) ? 3'd2 : 3'd3;
8
9   assign E_Tnew = (E_rc | E_ic | E_mf) ? 3'd1 :
10                  (E_load | E_mfc0) ? 3'd2 :
11                  3'd0;
12
13  assign E_stall_rs = (E_A3Sel == D_rs && D_rs != 5'd0) && (E_Tnew >
    D_Tuse_rs);
14  assign E_stall_rt = (E_A3Sel == D_rt && D_rt != 5'd0) && (E_Tnew >
    D_Tuse_rt);
15
16  assign M_Tnew = (M_load | M_mfc0) ? 3'd1 : 3'd0;
17
18  assign M_stall_rs = (M_A3Sel == D_rs && D_rs != 5'd0) && (M_Tnew >
    D_Tuse_rs);
19  assign M_stall_rt = (M_A3Sel == D_rt && D_rt != 5'd0) && (M_Tnew >
    D_Tuse_rt);
20  assign M_stall_eret = D_eret && ((E_mtc0 && E_rd == 5'd14) || (M_mtc0 &&
    M_rd == 5'd14));
21
22  assign E_stall_mdu = ((D_mf | D_mt | D_md) && (E_MDU_busy | E_MDU_start));
23
24
25  // mips.v
```

```
wire stall;
assign FD_WE = !stall; //冻结FD寄存器
assign IFU_WE = !stall; //冻结PC
assign DE_flush = stall; //清空DE寄存器

assign DE_WE = 1'b1;
assign EM_WE = 1'b1;
assign MW_WE = 1'b1;

assign FD_flush = 1'b0;
assign EM_flush = 1'b0;
assign MW_flush = 1'b0;
```

# 系统桥

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。

| 条目 | 地址或地址范围 | 备注 |
| --- | --- | --- |
| 数据存储器 | 0x0000_0000~0x0000_2FFF | |
| 指令存储器 | 0x0000_3000~0x0000_6FFF | |
| PC 初始值 | 0x0000_3000 | |
| 异常处理程序入口地址 | 0x0000_4180 | |
| 计时器 0 寄存器地址 | 0x0000_7F00~0x0000_7F0B | 计时器 0 的 3 个寄存器 |
| 计时器 1 寄存器地址 | 0x0000_7F10~0x0000_7F1B | 计时器 1 的 3 个寄存器 |
| 中断发生器响应地址 | 0x0000_7F20~0x0000_7F23 | |

Bridge的端口列表如下：

```
module bridge(
    input [31:0] m_data_rdata,
    input [31:0] m_temp_data_addr,
    input [31:0] m_temp_data_wdata,
    input [3:0] m_temp_data_byteen,
    input [31:0] TC0_out,
    input [31:0] TC1_out,
    output [31:0] m_data_addr,
    output [31:0] m_data_wdata,
    output [3:0] m_data_byteen,
    output [31:0] m_temp_data_rdata,
    output [31:0] TC0_addr,
    output TC0_WE,
    output [31:0] TC0_in,
    output [31:0] TC1_addr,
```

```
    output TC1_WE,
    output [31:0] TC1_in
    );
```

## MIPS微系统实现

```
// mips.v

module mips(
    input clk,                      // 时钟信号
    input reset,                    // 同步复位信号
    input interrupt,                // 外部中断信号
    output [31:0] macroscopic_pc,  // 宏观 PC

    output [31:0] i_inst_addr,     // IM 读取地址（取指 PC）
    input  [31:0] i_inst_rdata,    // IM 读取数据

    output [31:0] m_data_addr,     // DM 读写地址
    input  [31:0] m_data_rdata,    // DM 读取数据
    output [31:0] m_data_wdata,    // DM 待写入数据
    output [3 :0] m_data_byteen,   // DM 字节使能信号

    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 写使能信号
    output [4 :0] w_grf_addr,      // GRF 待写入寄存器编号
    output [31:0] w_grf_wdata,     // GRF 待写入数据

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

wire [31:0] TC0_addr;
wire TC0_WE;
wire [31:0] TC0_in;
wire [31:0] TC0_out;
wire TC0_IRQ;

wire [31:0] TC1_addr;
wire TC1_WE;
wire [31:0] TC1_in;
wire [31:0] TC1_out;
wire TC1_IRQ;

TC TC0(
    .clk(clk),
    .reset(reset),
    .Addr(TC0_addr[31:2]),
    .WE(TC0_WE),
    .Din(TC0_in),
    .Dout(TC0_out),
```

```
    .IRQ(TC0_IRQ)
);

TC TC1(
    .clk(clk),
    .reset(reset),
    .Addr(TC1_addr[31:2]),
    .WE(TC1_WE),
    .Din(TC1_in),
    .Dout(TC1_out),
    .IRQ(TC1_IRQ)
);

wire [31:0] m_temp_data_addr;
wire [31:0] m_temp_data_rdata;
wire [31:0] m_temp_data_wdata;
wire [3:0] m_temp_data_byteen;

wire [5:0] HWInt;
wire HWIntResponse;
assign HWInt = {3'b000, interrupt, TC1_IRQ, TC0_IRQ};

cpu CPU(
    .clk(clk),
    .reset(reset),
    .HWInt(HWInt),
    .macroscopic_pc(macroscopic_pc),
    .i_inst_rdata(i_inst_rdata),
    .m_data_rdata(m_temp_data_rdata),
    .i_inst_addr(i_inst_addr),
    .m_data_addr(m_temp_data_addr),
    .m_data_wdata(m_temp_data_wdata),
    .m_data_byteen(m_temp_data_byteen),
    .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),
    .HWIntResponse(HWIntResponse)
);

wire [31:0] bridge_m_data_addr;
wire [3:0] bridge_m_data_byteen;

bridge Bridge(
    .m_data_rdata(m_data_rdata),
    .m_temp_data_addr(m_temp_data_addr),
    .m_temp_data_wdata(m_temp_data_wdata),
    .m_temp_data_byteen(m_temp_data_byteen),
    .TC0_out(TC0_out),
    .TC1_out(TC1_out),
    .m_data_addr(bridge_m_data_addr),
    .m_data_wdata(m_data_wdata),
    .m_data_byteen(bridge_m_data_byteen),
    .m_temp_data_rdata(m_temp_data_rdata),
```

```
    .TC0_addr(TC0_addr),
    .TC0_WE(TC0_WE),
    .TC0_in(TC0_in),
    .TC1_addr(TC1_addr),
    .TC1_WE(TC1_WE),
    .TC1_in(TC1_in)
);

assign m_data_addr = (HWIntResponse && interrupt) ? 32'h0000_7f20 : bridge_m_data_addr;
assign m_data_byteen = (HWIntResponse && interrupt) ? 4'b0001 : bridge_m_data_byteen;

assign m_int_addr = (HWIntResponse && interrupt) ? 32'h0000_7f20 : bridge_m_data_addr;
assign m_int_byteen = (HWIntResponse && interrupt) ? 4'b0001 : bridge_m_data_byteen;

endmodule
```

# 测试方案

对特定的异常和中断编写程序进行测试。

## 取值异常

```
.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
```

## 存取地址异常

```
.text
    ori $28, $0, 0x0000
    ori $29, $0, 0x0f00
    mtc0    $0, $12

    lui $8, 0x7fff
    ori $8, $8, 0xffff

    lui $9, 0x8000
    ori $9, $9, 0x0000

    lw  $10, 1($8)      # 测试对 lw 地址上界溢出的处理
    lh  $10, 1($8)      # 测试对 lh 地址上界溢出的处理
    lb  $10, 1($8)      # 测试对 lb 地址上界溢出的处理
    lw  $10,-1($9)      # 测试对 lw 地址下界溢出的处理
    lh  $10,-1($9)      # 测试对 lh 地址下界溢出的处理
    lb  $10,-1($9)      # 测试对 lb 地址下界溢出的处理

    sw  $10, 1($8)      # 测试对 sw 地址上界溢出的处理
    sh  $10, 1($8)      # 测试对 sh 地址上界溢出的处理
    sb  $10, 1($8)      # 测试对 sb 地址上界溢出的处理
    sw  $10,-1($9)      # 测试对 sw 地址下界溢出的处理
    sh  $10,-1($9)      # 测试对 sh 地址下界溢出的处理
    sb  $10,-1($9)      # 测试对 sb 地址下界溢出的处理

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
```

## 计算溢出

```
.text
    ori $28, $0, 0x0000
    ori $29, $0, 0x0f00
    mtc0    $0, $12

    lui $8, 0x7fff
    ori $8, $8, 0xffff

    lui $9, 0x8000
    ori $9, $9, 0x0000

    ori $10, 0x0001
    lui $11, 0xffff
    ori $11, $11, 0xffff

    add $12, $10, $8        # 测试 add 上界溢出的情况
    add $12, $11, $9        # 测试 add 下界溢出的情况
    addi    $12, $8, 1      # 测试 addi 上界溢出的情况
    addi    $12, $9, -1     # 测试 addi 下界溢出的情况
    sub $12, $8, $11        # 测试 sub 上界溢出的情况
    sub $12, $9, $10        # 测试 sub 下界溢出的情况
```

## 计时器功能

```
.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

```
.text
    ori $28, $0, 0x0000
    ori $29, $0, 0x0f00
    mtc0    $0, $12

    j   nxt1
    lw  $0, 1($0)           # 测试延迟槽内 lw 地址不对齐异常
nxt1:
    j   nxt2
    sw  $0, 1($0)           # 测试延迟槽内 sw 地址不对齐异常
nxt2:
    lui $8, 0x7fff
    ori $8, $8, 0xffff
    j   end
    addi    $10, $8, 1      # 测试延迟槽内 addi 溢出异常
    end:j end
    nop
```

## 未知指令/系统调用

```
    lui      $s0,0x8000
    lui      $s1,0x7fff
    ori      $s1,$s1,0xffff
    syscall
    add     $10,$s0,$s0
    sub     $10,$s0,$s1
    addi    $10,$s1,10
    sw      $10,0x1002($0)
    sh      $10,0x1001($0)
    mult    $10,$10
    lw      $10,0x1002($0)
    lh      $10,0x1001($0)
    mult    $10,$10
    lh      $10,0x1001($0)
    sub     $10,$s0,$s1
    addi    $10,$s1,10
    sw      $10,0x1002($0)
    sh      $10,0x1001($0)
    mult    $10,$10
    sw      $10,0x1002($0)
    sh      $10,0x1001($0)
    lw      $10,0x1002($0)
    lh      $10,0x1001($0)
    lhu     $10,0x1001($0) # 未知指令
    mult    $10,$10
    sh      $10,0x1001($0)
    add     $10,$s0,$s0
    sub     $10,$s0,$s1
    mult    $10,$10
    add     $10,$s0,$s0
    sub     $10,$s0,$s1
    j label_1
    add     $10,$s0,$s0
    sub     $10,$s0,$s1
```

## 综合测试

```
ori $1,$1,0x7001
mtc0 $1,$12

#pc地址未对齐
ori $2,$2,0x300a
#jr $2
#add $2,$2,$2#顺便延迟槽
#pc地址超范围
#jr $2
#ori $3,$3,0x0003

#lw、lh没有字对齐
lw $2,0($3)
lh $2,1($0)
#lh、lb取Timer寄存器的值
```

```
ori $4,$4,0x7f00
lw $5,0($4)#应该没错
lh $5,0($4)
lb $5,20($4)
#计算地址加法溢出
lui $6,65535
ori $6,$6,65535
lw $7,1($6)
#取数地址超出范围
ori $7,0x7f0c
lw $7,0($7)

#sw、sh没有字对齐
sw $2,0($3)
sh $2,1($0)
#sh、sb取Timer寄存器的值
sw $5,0($4)#应该没错
sh $5,0($4)
sb $5,20($4)
#计算地址加法溢出
lui $6,65535
ori $6,$6,65535
sw $7,1($6)
#向计时器Count寄存器存值
sw $7,-4($7)
#存数地址超出范围
sw $7,100($7)

#syscall
syscall

#RI
nor $2,$3,$4

#算术溢出
addi $1,$0,1
sub $8,$0,$1
add $9,$8,$6#不应溢出
sub $9,$6,$8#溢出
add $9,$6,$7#溢出
sub $9,$0,$6
addi $9,$9,-100#溢出


end:
beq $0,$0,end#死循环
nop

#异常处理程序
.ktext 0x4180
mfc0 $k0,$12
mfc0 $k0,$13
mfc0 $k0,$14
addi $k0,$k0,4
mtc0 $k0,$14
```

```
71 eret
72 add $2,$2,$2#应当没有延迟槽
```

# 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?

   鼠标和键盘的输入信号都会转化为不同的系统中断信号，CPU根据中断信号的值可以执行对应的汇编指令。

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）

   依旧可以实现，无非是需要更改一下CPU中当出现异常或中断时要跳转到的异常处理程序地址，之后由用户提供的程序依旧可以对中断和异常进行处理。但入口常常变动会导致该CPU的适用性降低，换个执行指令段需要换个入口。

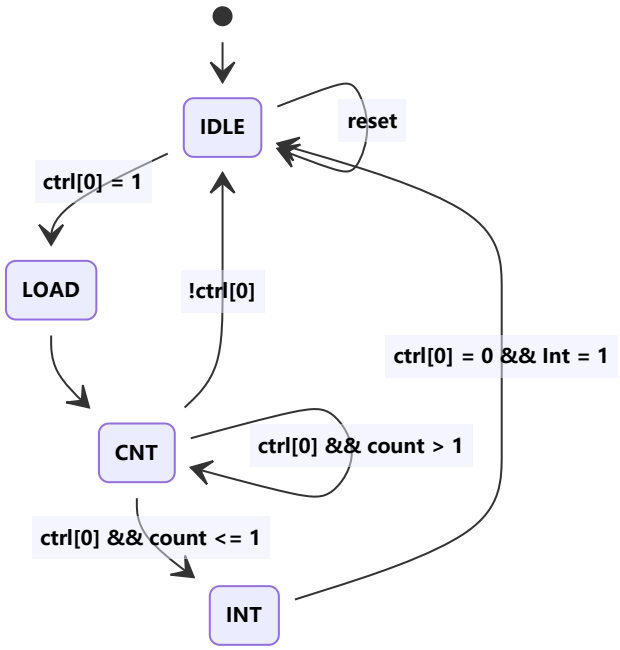
3. 为何与外设通信需要 Bridge?

   使得CPU不需要关心具体的数据从何而来，只需要知道地址即可。假如每个外设都要针对CPU做单独处理，那么时间与经济成本实在是过于昂贵且没必要了。

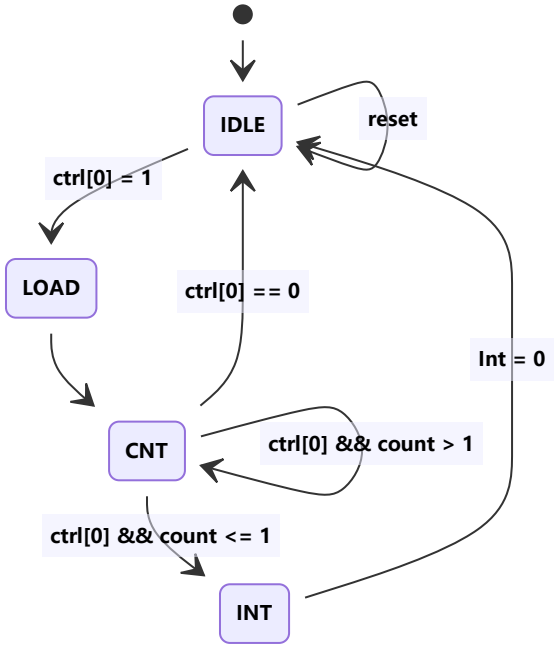
4. 请阅读官方提供的定时器源代码，阐述两种中断模式的异同，并分别针对每一种模式绘制状态移图。

   相同之处：在允许计数的情况下，都是从初值寄存器中获取初数值到计数值寄存器中开始计数，两种模式都受控制寄存器的控制
   区别之处：模式0在计数结束后，会一直提供中断信号，直到IM或者EN被修改使其禁止中断或停止计数，模式1在计数结束后，只会提供一周期的中断信号，然后自动再次赋初值开始计数，知道IM或者EN被修改行为才会被改变

   模式0的状态转移图如下：




   模式1的状态转移图如下：




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？

   会导致宏观PC突然为0，这显然是不合理的。在清空流水线的时候，应该保留PC信息。

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

   如果 `jalr $31 $31`的延迟槽内发生异常或需要响应中断，由于此时 `$31` 寄存器的值已经被 `jalr` 改变，但是处理异常结束后，会再次执行 `jalr` 指令，从而就会跳转到不正确的 `PC` 地址。