P7-CPU设计文档

#CPU #Architecture

---

1 Notation

> 命名规则参考:
> CO教程要求
> Digital Design and Computer Architecture

1.1 DataBus

- Instr: 32 位指令信号，将被分为
  - opcode
  - rs
  - rt
  - rd
  - shamt
  - funct
  - imm16
  - imm26
- pc: 32 位程序计数器信号
- npc: 32位下一时钟刻程序计数器信号
- pc4：始终为PC + 4

- ExtImm：EXT输出完成扩展后的立即数
- RegAddr: GRF 5 位写入地址
- RegData: GRF 32 位写入数据
- srcA：ALU写入数据源A
- srcB：ALU写入数据源B
- outC：ALU输出数据源C
- ALUsig：ALU特殊计算标志位，处理Zero或其余标志信号
- MemAddr: DM 32 位写入地址
- MemData: DM 32 位写入数据
- ReadData：DM 32位输出数据
- **流水级相关**：使用@F @D @E @M @W表示某一阶段的数据，例如某级的指令编码为Instr@M

1.2 ControlBus

- RegWrite: GRF 写入控制信号
- MemWrite: DM 写入控制信号
- ALUOp：ALU操作信号
- NPCOp：NPC操作信号
- EXTOp：EXT操作信号
- RegDst：GRF.A3选择数据源
- RegSrc：GRF选择数据源
- ALUSrc：ALU选择数据源

1.3 MoveForward

规定所有经过转发的RD1/RD2数据源使用T作为前缀，标注为经历转发的真实的数据，包括：

- T_RD1_D：D级经过转发后的真实GRF[rs]
- T_RD2_D：D级经过转发后的真实GRF[rt]
- T_RD1_E：E级经过转发后的真实GRF[rs]
- T_RD2_E：E级经过转发后的真实GRF[rt]
- T_RD2_M：M级经过转发后的真实GRF[rt]

---

2 Requirements

2.1 支持指令集

```
nop, 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
```

注意：

- `eret` 具有跳转的功能但是没有延迟槽，你的设计应该保证 `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。

- 补充说明：
  - 分支跳转指令无论跳转与否，延迟槽指令为受害指令时 `BD` 均需要置位。
  - 发生取指异常或 `RI` 异常后视为 `nop` 直至提交到 CP0。
  - 跳转到不对齐的地址时，受害指令是 PC 值不正确的指令（即**需要向 EPC 写入不对齐的地址**）。
  - 对于未知指令的判断仅需考虑 opcode（和 R 型指令的 funct），且仅需判断是否出现在 P7 要求的指令集中，同时保证未知指令的测试用例中 opcode 和 funct 码的组合一定没有在 MARS 的基本指令集中出现。

---

3 Heirarchy Designation

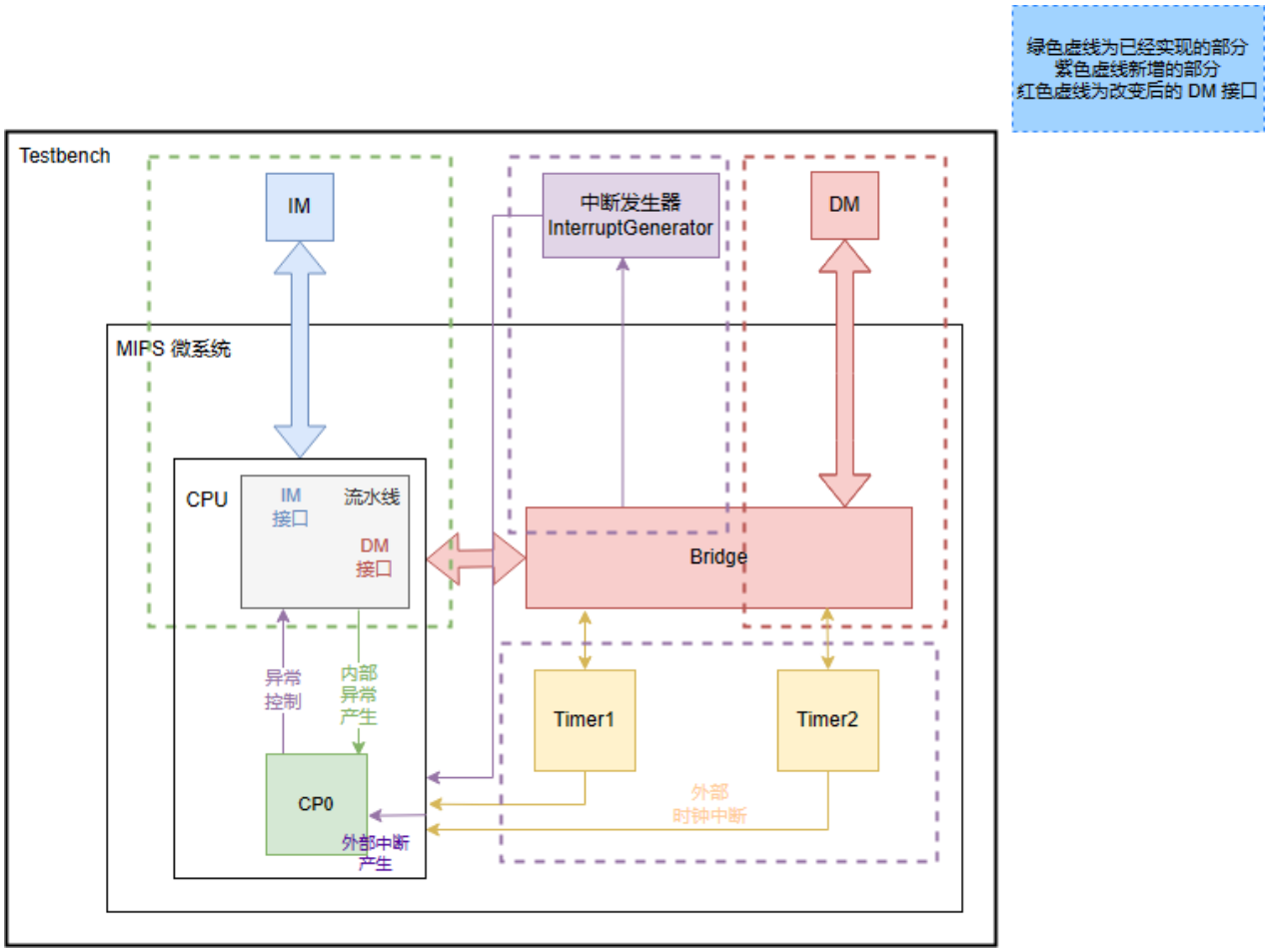
实现流程

1. 实现CP0
2. 完成元件内部，异常信号生成
3. 修改Controller
4. 完善转发阻塞逻辑
5. 扩展实现中断功能

架构

- mips
  - **Bridge**
  - Timer1/2（given）
  - `cpu`

- PC_REG
- NPC
- D_REG
- D_CU (CU)
- GRFs
- EXT
- CMP
- E_REG
- E_CU (CU)
- ALU
- MultDiv
- M_REG
- M_CU (CU)
- BE
- DP
- **CP0**
- W_REG
- W_CU (CU)
- CU
- MFSCU



设计构思

反思

P7首挂后再思考

首先，仅仅对于P7扩展的CPU而言，我们其实CPU内部唯一的扩展内容是

1. 加入CP0元件
2. 加入以前元件对于异常的扩展
3. 扩展指令集，加入全新的译码

挂掉的P7设计的缺陷：

1. 单独设立EXC元件完全没有必要，完全可以直接融入各级流水寄存器
2. 对于流水寄存器的命名规范不好，Req就是Req，不要搞flush弄得自己也不明白
3. 对于异常码的生成处理的不好，最后的Encoder内部还需要实现指令译码，这件事违背了"高内聚，低耦合"的代码设计原则
4. 在设计过程中有些事情没有想好构思好久开始弄

改进思路：

1. 重新处理异常码问题，改为每一流水级都要进行译码，考虑到需要对Opcode和funct进行处理，将EXCode生成融入Controller
2. 每一个流水级仅生成对应的指令异常码，严格遵循示意图
   - F -- PC异常
   - D -- 指令异常 -- Syscall or RI
   - E -- 算数异常 -- 包含指令存取地址 + 算数溢出
3. 重新思考关于部分指令的处理

四条指令：

`mfc0`

CP0 → GRFs

从CP0接受 + 写入，和DM一样，从M级取出，流入W级流水

在转发阻塞层面上与`lw`相似

处理办法：

1. **沿用之前的转发路径**
2. 对其使用A-T法分析

扩展需求：Tnew/Tuse in CU

`mtc0`

GRFs → CP0

从对于GRFs的转发和阻塞角度而言，并没有任何影响，只需要**沿用之前的转发路径**

但是从CP0的转发和阻塞角度而言，我们必须考虑到出现写入CP0前使用CP0寄存器内容的情况：

1. 用到`mfc0`，这个无需担心，因为只要程序员视角下指令序列为 `mtc0` ... `mfc0` ，指令就可以正常执行，因为对于CP0的操作都是在M级
2. 对于`eret`，这是个大问题，需要建立从E、M级向EPC的转发通路(W级不用，因为W级时EPC已经被写入)

扩展需求：EPC转发，Tnew/Tuse/CPWrite in CU

`syscall`
一个特殊的异常

对于`syscall`指令而言，处理方法应该为**如果没有F级PC异常**，则当前指令在**D级生成Syscall异常。**

扩展需求：EXCode in CU

`eret`
当出现该指令的时候(或者说在**D级译码出该指令**)

需要考虑以下几件事：

1. 由于`eret`无延时槽，因此需要将下一周期流水进入D级的指令清空，这里的**清空D级流水**需要注意一个新的问题，宏观PC到底跟谁走？

   > 答案是跟着返回的坐标走，我们考虑eret在W级，此时nop在M级，如果**这个时候突然来一个中断**，if跟着eret的PC，则**EPC变成了ERET** -- 死循环！

2. `eret`清空CP0内核态，必须等到`eret`流水到M级才能清空`EXClr`，记住我们已经将流水线CPU封装为单周期CPU！

3. 延时槽标记问题：只需要置0即可，原因为一方面**eret指令不存在延时槽**，一方面对于EPC而言，由于延时槽标记的存

在，**我们的EPC绝对不会跳转到一条延时槽指令**

扩展需求：ERET in CU / EXClr in CU / flush in D

其余指令的新内容

> 引入了中断与异常

优先级：中断优先于异常

异常

F级： PC异常 -- 地址 对齐 & 范围
元器件：NPC对PC_F进行判断，**直接生成异常码**
EXCode_F流入D级

D级：
元器件：

1. Controller先接受PC异常信号**EXCode_D**，考虑是否译码，如果出现了PC异常问题，则不进行译码，直接当成nop处理
2. D级需要处理`Syscall`和`RI`的译码的问题，出现该问题则置异常码
   T_EXCode_D流入E级

E级：
元器件：

1. ALU：给出算数溢出Overflow异常，返回CU
2. 由CU根据**EXCode_E**判断，并根据Overflow信息生成**T_EXCode_E**
   T_EXCode_E流入M级

> 对于控制器CU而言，当出现了**接受到的EXCode**（非输出的T_EXCode）非5'd0的情况，直接处理为`nop`指令，并将异常码流水即可

M级：
M级不存在再生成EXCode的问题，直接提交EXCode_M值CP0即可

W级：
W级无需考虑异常的问题

异常相关处理

处理关于：

1. 宏观PC = PC_M
2. ISDB 延时槽
3. Req

需要在**D/E/M级流水寄存器**扩展信号位

注意信号优先级： reset > Req > stall

宏观PC和ISDB的注意点主要在于当出现阻塞的空泡`nop`，需要将宏观PC和ISDB标记**与被阻塞的指令同步**

在D级判断**下一周期进入D级的指令是否为延时槽指令**

特别的，对于`Req`信号
各级流水寄存器、NPC、BE、MultDiv均需要接受，但GRFs不用（W级永远是"已经被执行"的指令）

Implement

Bridge

端口信号/功能表

信号名	方向	描述
CPUAddr[31:0]	I	cpu来的读写外设地址
CPUByteen[3:0]	I	cpu输出的字节使能信号
DMData[31:0]	I	DM的读来的数据
TCData0[31:0]	I	Timer0读来的数据
TCData1[31:0]	I	Timer1读来的数据
Byteen[3:0]	O	真正的DM字节使能信号
WriteTC0	O	是否写Timer0
WriteTC1	O	是否写Timer1
ReadData[31:0]	O	真正从外设读取到的数据

> CPU对外界的**数据交互**必须通过系统桥
> 注意，课程组规格要求可知，我们无需通过系统桥处理中断相应程序，**系统桥仅管辖DM TC0 TC1**
> 对于中断相应信号，直接连接cpu输出的MemAddr，MemData，Byteen即可

事实上，写入和读出的数据地址只需要直接连接即可，Bridge的意义在于通过**写入地址判断到底和谁进行交互**，从而生成正确的**写入控制信号**和**读外设数据**

> 异常流水处理
> 设置每一级的异常流水寄存器，将异常情况进行记录，并在最后给到M_CU统一执行处理，生成异常编码

CP0

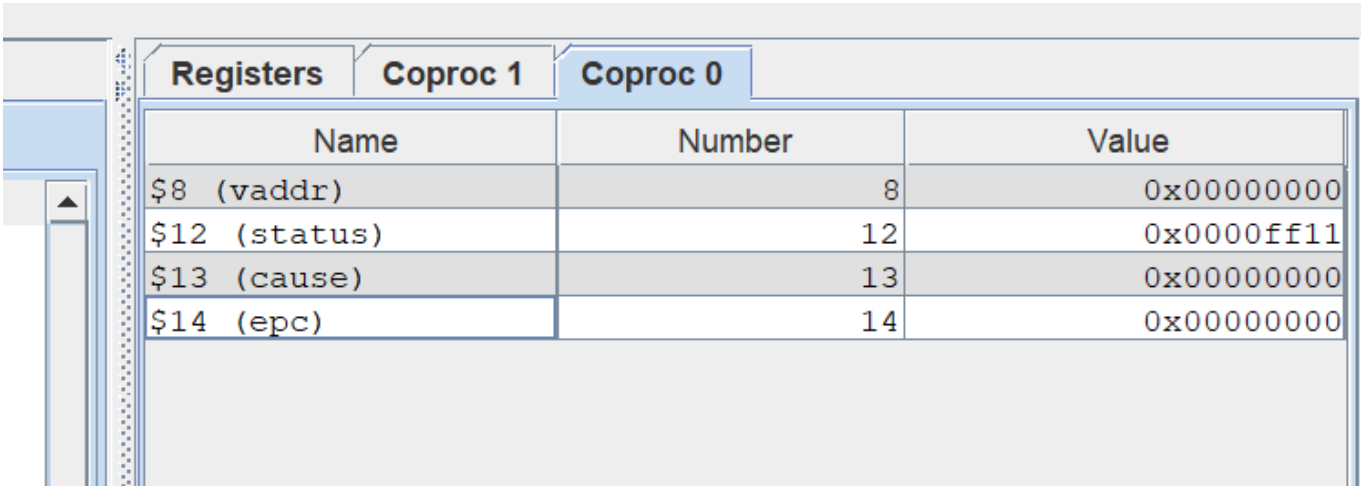
端口信号/功能表

信号名	方向	描述
clk	I	时钟
reset	I	复位
CPWrite	I	CP寄存器写使能
VPC[31:0]	I	受害PC
CPAddr[4:0]	I	CP寄存器地址
CPIn[31:0]	I	CP寄存器写入值
ISDB	I	延时槽标记

信号名	方向	描述
EXCode[4:0]	I	异常码，无异常置0即可
HWInt[5:0]	I	中断信号
EXClr	I	EPC复位
EPCout[31:0]	O	输出EPC寄存器
CPOut[31:0]	O	输出需求的CP寄存器值
Req	O	进入处理程序标志

内部实现

三个寄存器



名称	编号	功能
SR	12	配置异常功能
Cause	13	记录异常发生原因
EPC	14	记录受害PC

寄存器端口域	功能
SR[15:10]	对应6个外部中断，仅能通过`mtc0`修改，1则允许中断，0则禁止中断

寄存器端口域	功能
SR[1]	异常发生的时候置为高位，强制进入异常处理，禁止中断
SR[0]	全局中断使能，如果1则允许中断，0则禁止中断
Cause[31]	（延时槽造就）若为1，则EPC指向延时槽前跳转指令，若0，则当前指令
Cause[15:10]	6位待决中断位，对应6个外部中断，每周期来自Timer和其余外部中断，1表示有，0表示无
Cause[6:2]	异常码
EPC	记录异常处理后需要返回的PC

异常类型分析

ExcCode 的编码必须遵守规范，不然在评测的时候可能会出现问题。

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。

异常与中断码	助记符与名称	指令与指令类型	描述
PC 地址超过 `0x3000 ~ 0x6ffc`。			
`AdEL`（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`（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		

异常与中断码	助记符与名称	指令与指令类型	描述
	值。		
store 型指令	存数地址超出 DM、Timer0、Timer1、中断发生器的范围。		
8	`Syscall`（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。

实现分工：

- NPC -- 负责ADEL类异常（PC地址范围+对齐）
- CU -- 负责译码Syscall异常和RI异常
- ALU -- 负责Overflow异常报告

> 注意处理地址异常的时候，有一个点是地址算数溢出需要报告为AdES或AdEL而非Overflow这个点需要注意

其余元件扩展思路

首先扩展各级流水的思路：

1. 直接流水异常码和ISDB信号

2. 每条指令需要判断是否为延时槽
3. 所有时序相关元件需要接收Req信号，主要功能包含**flush**寄存器功能/**禁用**当前功能/判断是否**启动**乘除信号
4. NPC需要根据该信号执行跳转到异常处理程序地址
5. 对于清空延时槽和阻塞造成的气泡设置，注意宏观PC含义以及ISDB标记（**气泡的PC和ISDB应该与D级被阻塞指令一致**）

需要进行修改扩展的元件：

- D级
  - D_REG：扩展flush信号，扩展延时槽判断流水
  - NPC：扩展跳转至异常程序处理口，扩展pc异常
  - D_CU：给出解码异常问题，给出需要延时槽标记
  - D_EXC：记录当前级元件异常报告
- E级
  - E_REG：扩展flush信号，扩展延时槽判断流水
  - ALU：扩展算数溢出异常
  - MultDiv：扩展Req条件启动乘除法运算问题
  - E_EXC：记录当前级元件异常报告
- M级
  - M_REG：扩展flush信号，扩展延时槽判断流水
  - BE：扩展Req条件下，关闭所有
  - 扩展元件Judge：接受各级异常流水，生成对应异常码
  - M_EXC：记录当前级元件异常报告
- W级

    - W_REG：扩展flush信号，在Req指令下冲刷流水（不然异常指令会正常流入W级），扩展延时槽判断流水
- CU：D级CU扩展判断是否下一个为延时槽指令，并将信号接入到D_REG
- MFSCU：扩展数据通路转发控制信号

4.2 PC_REG

端口信号/功能表

信号	方向	描述
clk	I	时钟信号
reset	I	**同步**复位
stall	I	冻结信号
npc[31:0]	I	下一个指令的地址
pc[31:0]	O	当前指令地址

4.3 D_REG

端口信号/功能表

信号	方向	描述
clk	I	时钟信号
reset	I	**同步**复位
Req	I	异常信号
F_Instr[31:0]	I	新指令

信号	方向	描述
F_pc[31:0]	I	新指令pc值
Cond	I	条件跳转信号
flush	I	流水冲刷信号
ISDB_F	I	判断记录F级是否为延时槽指令
D_Instr[31:0]	O	当前指令Instr
D_pc[31:0]	O	当前指令pc值
D_Imm16[15:0]	O	Instr分线Imm16@D
D_Imm26[25:0]	O	Instr分线Imm26@D
ISDB_D	O	判断记录D级是否为延时槽指令

4.4 NPC

端口信号/功能表

信号	方向	描述
NPCOp[1:0]	I	操作控制码
Req	I	当前是否需要进入异常处理状态
pc_F[31:0]	I	当前F级的pc值
pc_D[31:0]	I	当前D级的pc值
Imm16[15:0]	I	16位立即数
Imm26[25:0]	I	j指令26位伪地址
Addr[31:0]	I	寄存器相关直接寻址
npc[31:0]	O	下一个pc
pc4[31:0]	O	常态输出PC+4的值

信号	方向	描述
PC_EXC	O	给出PC是否出现异常信号

NPCOp	功能	描述
2'b00	PC+4	npc = 顺序执行F_pc + 4
2'b01	PC相对寻址	npc = D_pc + 4 + SignExt(Imm16 | 00)
2'b10	伪直接寻址	npc = D_pc[31:28] | Imm26 | 00
2'b11	寄存器寻址	npc = Addr
Req	进入异常处理入口	npc = 0×0000_4080

注意这里，需要流水错误的PC值

4.5 GRF

端口信号/功能表

信号	方向	描述
clk	I	时钟信号
reset	I	**同步**复位信号
RegWrite	I	写入使能信号
A1[4:0]	I	Src of RD1
A2[4:0]	I	Src of RD2
RegAddr[4:0]	I	写入寄存器地址
RegData[31:0]	I	写入寄存器堆的数据

信号	方向	描述
pc[31:0]	*I*	*当前程序计数器值(辅助评测)*
RD1[31:0]	O	输出1
RD2[31:0]	O	输出2

序号	功能	描述
1	写入	当**RegWrite为1**且时钟**上升沿**时，更新GRF[A3]的数据为WD
2	输出	持续输出本上升沿记录的A1/A2数据，从RD1/RD2输出

4.6 EXT

端口信号/功能表

信号	方向	描述
EXTOp[1:0]	I	2'b00则为ZeroEXT，2'b01则为SignEXT，2'10为LUI
Imm16[15:0]	I	16位立即数
ExtImm[31:0]	O	扩展后的Imm32

4.7 CMP

端口信号/功能表

信号	方向	描述
C1[31:0]	I	比较数据源1
C2[31:0]	I	比较数据源2
CMPOp[2:0]	I	选择比较类型
Cond	O	比较结果

CMPOp	功能	描述
3'b000	BEQ	(C1 == C2) ? 1'b1 : 1'b0
3'b001	BNE	(C1 != C2) ? 1'b1 : 1'b0

4.8 E_REG

端口信号/功能表

信号	方向	描述
clk	I	时钟信号
reset	I	**同步**复位
flush	I	流水冲刷信号
Req	I	异常信号
stall	I	阻塞信号
ISDB_D	I	D级延时槽标记
D_Instr[31:0]	I	新指令

信号	方向	描述
D_pc[31:0]	I	新指令pc值
RD1[31:0]	I	GRF.RD1
RD2[31:0]	I	GRF.RD2
pc4[31:0]	I	jal中F_pc+4
ExtImm[31:0]	I	EXT.ExtImm
E_Instr[31:0]	O	当前指令Instr
E_pc[31:0]	O	当前指令pc值
E_pc4[31:0]	O	jal中F_pc+4
E_RD1[31:0]	O	GRF.RD1@E
E_RD2[31:0]	O	GRF.RD2@E
E_ExtImm[31:0]	O	EXT.ExtImm@E
ISDB_E	O	E级延时槽标记

ALU

端口信号/功能表

信号	方向	描述
srcA[31:0]	I	数据源SrcA
srcB[31:0]	I	数据源SrcB
ALUOp[4:0]	I	选取ALU操作
outC[31:0]	O	计算结果
Overflow	O	有符号数算数溢出标记

ALUOp	功能	描述
5'h00	ADD加法	C = A + B
5'h01	SUB减法	C = A - B
5'h02	OR或	C = A | B
5'h03	LUI高位	C = B，由EXT执行扩展，仅仅借用数据通路
5'h04	AND与	C = A & B
5'h05	SLT	C = (A < B) ? 1'b1 : 1'b0　处理为有符号数
5'h06	SLTU	C = (A < B) ? 1'b1 : 1'b0　处理为有符号数

MultDiv

端口信号/功能表

信号	方向	描述
Start	I	乘除法启动信号
MDOp[2:0]	I	操作信号
MDWrite	I	MD元件写使能
Req	I	全局异常信号，出现时不启动乘除运算
MDSel	I	输出选择信号,1'b0为lo,1'b1为hi
MD1[31:0]	I	乘法源1/被除数

信号	方向	描述
MD2[31:0]	I	乘法源2/除数
Busy	O	模拟乘除法运算周期，置为高位说明正在工作
MDout[31:0]	O	乘法高位/除法余数

MDOp	功能	描述
3'b000	mtlo	移动MD1(reg[rs])至寄存器lo
3'b001	mthi	移动MD1(reg[rs])至寄存器hi
3'b010	mult	执行乘法操作 (5周期)
3'b011	multu	执行无符号乘法操作 (5周期)
3'b100	div	执行除法操作 (10周期)
3'b101	divu	执行无符号除法操作 (10周期)

MDSel	功能	描述
0	mflo	移动lo值到reg[rd]
1	mfhi	移动hi值到reg[rd]

4.10 M_REG

端口信号/功能表

信号	方向	描述
clk	I	时钟信号

信号	方向	描述
reset	I	**同步**复位
flush	I	冲刷流水
ISDB_E	I	延时槽标记
E_Instr[31:0]	I	新指令
E_pc[31:0]	I	新指令pc值
E_pc4[31:0]	I	pc + 4数值@E
E_RD2[31:0]	I	GRF.RD2@E
E_outC[31:0]	I	ALU.outC@E
E_MDout[31:0]	I	MultDiv元件输出
E_Tnew[3:0]	I	E级的Tnew
M_Instr[31:0]	O	当前指令Instr
M_pc[31:0]	O	当前指令pc值
M_pc4[31:0]	O	pc + 4数值@M
M_RD2[31:0]	O	GRF.RD2@M
M_outC[31:0]	O	ALU.outC@M
M_MDout[31:0]	O	MultDiv元件输出流水
M_Tnew[3:0]	O	M级的Tnew
ISDB_M	O	延时槽标记

BE (Bit-Enable)

端口信号/功能表

信号	方向	描述
MemWrite	I	DM写使能信号
Req	I	全局异常信号，出现时禁用写使能
MemAddr[31:0]	I	32位DM地址
MemData[31:0]	I	DM中取出的数据
BEOp[1:0]	I	当前哪种类型存储sw/sh/sb
StoreData[31:0]	O	要写入DM的32位数据
ByteWrite[3:0]	O	4-bit，1代表对应位置写使能

BEOP	功能	描述
2'b00	SW	4'b1111 & {4{MemWrite}}
2'b01	SH	看Addr[1]，决定到底是4'b1100 & {4{MemWrite}} or 4'b1100 & {4{MemWrite}}
2'b10	SB	看Addr[1:0]，确定ByteWrite输出

DP (Data-Processor)

端口信号/功能表

信号	方向	描述
DPOp[2:0]	I	选择数据扩展方式
ReadData[31:0]	I	DM的32位输出数据

信号	方向	描述
MemAddr[31:0]	I	只关心MemAddr[1:0]
LoadData[31:0]	O	向寄存器中load的数据值

信号	功能	描述
3'b000	不扩展	
3'b001	符号扩展	对8位数据进行符号扩展
3'b010	符号扩展	对16位数据进行符号扩展

Encoder

端口信号/功能表

信号	方向	描述
Instr_M[31:0]	I	M级指令
PC_EXC_M	I	PC异常
Overflow_M	I	算数溢出标记
outC_M[31:0]	I	ALU结果
EXCode[4:0]	O	异常码

W_REG

端口信号/功能表

信号	方向	描述
clk	I	时钟信号

信号	方向	描述
reset	I	**同步**复位
flush	I	清空流水级寄存器信号
M_Instr[31:0]	I	新指令
M_pc[31:0]	I	新指令pc值
M_pc4[31:0]	I	pc + 4数值@M
M_outC[31:0]	I	ALU.outC@M
M_MDout[31:0]	I	MultDiv.MDout@M
M_LoadData[31:0]	I	DP.LoadData@M
M_Tnew[3:0]	I	M级的Tnew
W_Instr[31:0]	O	当前指令Instr
W_pc[31:0]	O	当前指令pc值
W_pc4[31:0]	O	pc + 4数值@W
W_LoadData[31:0]	O	DP.LoadData@M
W_outC[31:0]	O	ALU.outC@W
W_MDout[31:0]	O	MultDiv.MDout@W
W_Tnew[3:0]	O	W级的Tnew

EXC

其实仅仅需要流水`Overflow` `PC_EXC`信号

信号	方向	描述
clk	I	
flush	I	

信号	方向	描述
stall	I	
PC_EXC_in	I	
Overflow_in	I	
PC_EXC_out	O	
Overflow_out	O	

Controller

端口信号/功能表

信号	方向	描述
Instr[31:0]	I	需要译码的指令
Cond	I	条件跳转，判断是否条件成立
NPCOp[1:0]	O	NPC操作信号
EXTOp[1:0]	O	EXT操作信号
CMPOp[2:0]	O	CMP控制信号
RegWrite	O	GRF写入控制信号
RegDst[1:0]	O	GRF选择写入寄存器
RegSrc[1:0]	O	GRF选择RegData数据源
ALUSrc	O	ALU选择数据源
ALUOp[2:0]	O	ALU操作信号
MDOp[2:0]	O	MultDiv元件操作
MemWrite	O	DM 写入控制信号
BEOp[1:0]	O	BE存储指令类型

信号	方向	描述
DPOp[2:0]	O	DP对内存值处理类型
Tnew[3:0]	O	E_Tnew值
Tuse_RS[3:0]	O	Tuse_RS值
Tuse_RT[3:0]	O	Tuse_RT值
Start	O	开始进行乘除法运算
ISDB_F	O	D级CU判断下一条指令是否为延时槽
flush_D	O	特别的为eret指令冲刷延时槽
ERET	O	特别跳转

序号	信号	功能
1	RegWrite	0禁止，1允许写入
2	MemWrite	0禁止，1允许写入
3	ALUOp[2:0]	ALU操作信号
4	NPCOp[1:0]	NPC操作信号
5	EXTOp[1:0]	00零扩展，01符号扩展，10LUI
6	CMPOp[2:0]	CMP控制信号，3'b000代表beq
7	RegDst[1:0]	2'b00选择rd，2'b01选择rt，2'b10选择0x1f($ra)
8	RegSrc[1:0]	2'b00选择ALU.C，2'b01选择DM.ReadData，2'b10选择PC+4
9	ALUSrc	0选择RD2，1选择ExtImm

其中Tuse_RS与Tuse_RT会在D级流水中的CU使用
而Tnew会生成E级流水的T_new并且随着流水线逐级寄存器传递

内部实现基于指令分类（对应**相同数据通路**，因此同类指令可能会出现Op不同，但是**MUX的选择信号一定相同**）

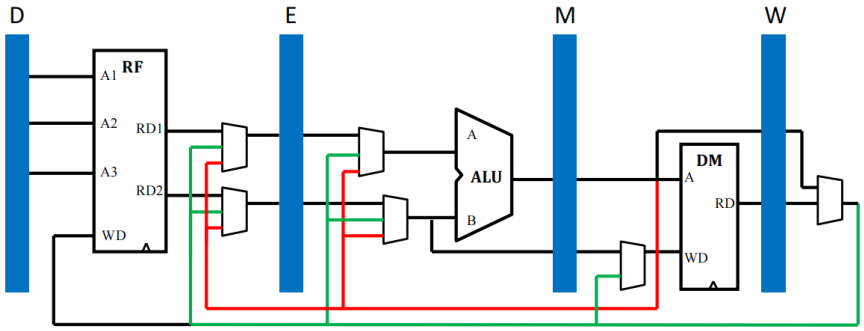
```
// Cal_reg_reg
add, sub, and, or, slt, sltu,

// Cal_Imm
lui, addi, andi, ori

// Load
lb, lh, lw

// Store
sb, sh, sw

// MultDiv
mult, multu, div, divu, mfhi, mflo, mthi,
mtlo

// B-type
beq, bne

// J_Link
jal

// J_reg
jr
```

4.14 MFSCU

转发

- 最新结果：可能出现在M和W
- 使用结果：D、E、M各级可能的位置
  - D级：RS寄存器值、RT寄存器值
  - E级：ALU的A、B；RT寄存器值
  - M级：DM的WD

	M级ALU计算结果	W级回写结果
D级被转发	RS寄存器值 RT寄存器值	RS寄存器值 RT寄存器值
E级被转发	ALU的A ALU的B RT寄存器值	ALU的A ALU的B RT寄存器值
M级被转发		DM的WD



67

遍历所有转发情况：

转发源	接收者
E_REG	D_GRF(RD1/RD2)
M_REG	D_GRF(RD1/RD2)
M_REG	E_ALU(srcA/srcB)
W_REG	D_GRF(RD1/RD2)
W_REG	E_ALU(srcA/srcB)
W_REG	M_DM(WD)

其中，W_REG向D_GRF的转发可以依靠GRF内部转发实现，降低转发模块复杂度

> 为什么引入E to D的转发?
> 保证全速流水，对于LUI指令，E级就已经算出来了LUI结果

因此，我们的转发模块需要实现的功能为：

- E to D
- M to D
- M to E
- W to E
- W to M

转发条件：

1. 转发源RegWrite信号有效
2. 转发源RegAddr与rs/rt一致
3. RegAddr非0号寄存器

转发设计构思：
暴力转发法，先假设所有数据冒险情况都可以通过转发解决，先设计转发数据通路，并且不考虑是否Tnew = 0。

> 这样的方法是合理的，因为错误的转发值在执行前一定会被正确的转发值覆盖
> 就算加入阻塞后，由于其实插入的值为nop，转发对冒险并不存在影响

转发层次：

1. 从源出发，选择需要转发什么(SrcMUX)
2. 到接受者，确定需要接受什么(DstMUX)
3. 考虑数据优先级(MFCU)

转发源SRC_MUX

考虑接受者真正需要的是什么，应该转发什么
例如对于M级：流水寄存器有ALU.outC@M，pc4@M，需要根据指令类型，确定我需要转发重写什么。

这里选择转发源的操作其实与本级指令CU译码出来的控制信号一致

我们不关心SRC_MUX的输出到底是哪一种信号，
我们只知道：其输出应该是**一个“新鲜”的rs/rt的数据**

接受源DST_MUX

考虑从当前接受的多个信号源中，用真正的新鲜的数据替代老旧的数据，这个控制信号需要依赖MFCU生成

MFSCU

流水级越低，数据越“新鲜”，优先使用低流水级数据

寄存器阻塞

使用A-T分析法易知，当且仅当出现Tuse < Tnew时才需要阻塞
列表分析指令的Tuse和Tnew，

并基于此给出策略矩阵

并且需要满足前提条件：

- 指令对应寄存器相等
- 需要进行写寄存器
- 寄存器非0

```
96      wire STALL_RS_E = (Tuse_RS < Tnew_E) && (RegAddr_E != 5'd0) && (RegAddr_E == rs_D) && (RegWrite_E);
97      wire STALL_RS_M = (Tuse_RS < Tnew_M) && (RegAddr_M != 5'd0) && (RegAddr_M == rs_D) && (RegWrite_M);
98      wire STALL_RS_W = (Tuse_RS < Tnew_W) && (RegAddr_W != 5'd0) && (RegAddr_W == rs_D) && (RegWrite_W);
99
100     wire STALL_RT_E = (Tuse_RT < Tnew_E) && (RegAddr_E != 5'd0) && (RegAddr_E == rt_D) && (RegWrite_E);
101     wire STALL_RT_M = (Tuse_RT < Tnew_M) && (RegAddr_M != 5'd0) && (RegAddr_M == rt_D) && (RegWrite_M);
102     wire STALL_RT_W = (Tuse_RT < Tnew_W) && (RegAddr_W != 5'd0) && (RegAddr_W == rt_D) && (RegWrite_W);
103
104     wire STALL_RS = STALL_RS_E | STALL_RS_M | STALL_RS_W;
105     wire STALL_RT = STALL_RT_E | STALL_RT_M | STALL_RT_W;
106
107     assign stall = STALL_RS | STALL_RT;
108
109 endmodule
110
```

乘除单元阻塞

约定： 当 Busy 信号或 Start 信号为 1 时，`mult, multu, div, divu, mfhi, mflo, mthi, mtlo` 等乘除法相关的指令均被阻塞在 D 流水级。

端口信号/功能表

信号	方向	描述
Start	I	乘除单元启动信号
Busy	I	乘除单元工作信号
Instr_D[31:0]	I	
Instr_E[31:0]	I	
Instr_M[31:0]	I	
RegAddr_E[4:0]	I	当前E级指令回写寄存器地址/无需回写则置0
RegWrite_E	I	E级指令写寄存器信号
RegAddr_M[4:0]	I	当前M级指令回写寄存器地址/无需回写则置0
RegWrite_M	I	M级指令写寄存器信号
RegAddr_W[4:0]	I	当前W级指令回写寄存器地址/无需回写则置0
RegWrite_W	I	W级指令写寄存器信号
Tuse_RS[3:0]	I	D级Tuse_RS
Tuse_RT[3:0]	I	D级Tuse_RT
Tnew_E[3:0]	I	E级流水的Tnew
Tnew_M[3:0]	I	M级流水的Tnew
Tnew_W[3:0]	I	W级流水的Tnew
D_RS_SEL[1:0]	O	D级RS的接受者MUX选择信号
D_RT_SEL[1:0]	O	D级RT的接受者MUX选择信号
E_RS_SEL[1:0]	O	E级RS的接受者MUX选择信号

信号	方向	描述
E_RT_SEL[1:0]	O	E级RT的接受者MUX选择信号
M_RT_SEL[1:0]	O	M级RT的接受者MUX选择信号
stall	O	阻塞信号

思考题

1

> 说明鼠标和键盘的输入信号是如何被 CPU 知晓的?

通过中断请求进行IO操作，当键盘上按下一个按键后，其实会发出一个请求信号，相当于一个外部中断器

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）

统一化中断异常处理一方面是统一化的处理会比较便于编码，降低CPU复杂度，相应的牺牲了CPU功能的灵活性

可以，而且我认为对于某些外部中断，他们可能对于特殊的外设需要特殊的中断处理程序，这些可能就需要单独进行分配处理程序空间，但是为了加入这个灵活变动的入口势必要扩展我们的硬件实现，会增加控制复杂度和硬件元件消耗

3

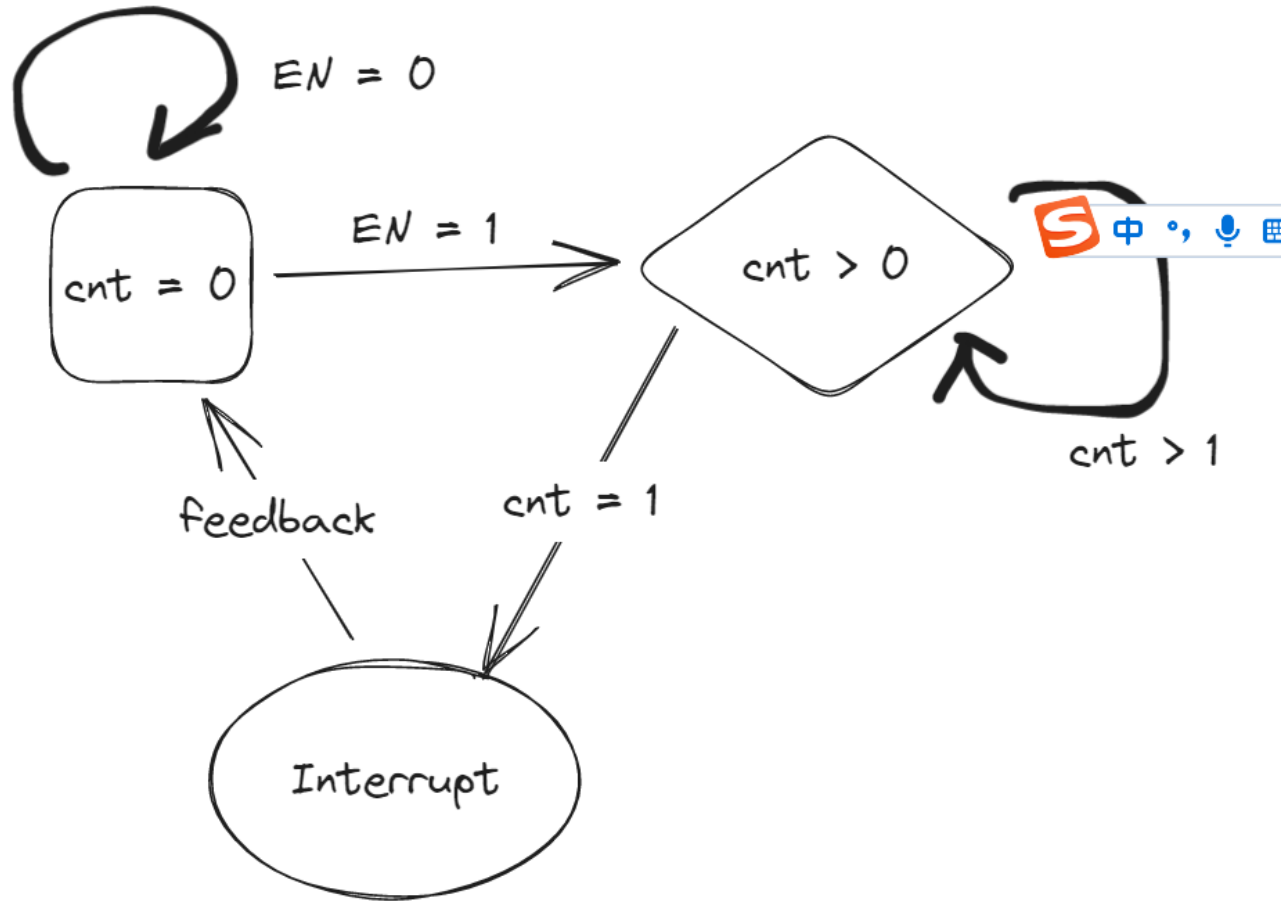
> 为何与外设通信需要 Bridge?

系统桥主要的作用其实是对于外设读写的分配，当我们需要有多种外设的时候，我们需要一个“中介”帮助我们分配存取任务，而且有的时候，由于对应的内存空间不同，我们可能对应的使能信号不同，比如本实验中对于Timer的读写其实是仅允许store类且WE为1-bit，但是CPU输出的都是4-bit的Byteen，就需要一个中间交换机来做一下信号处理

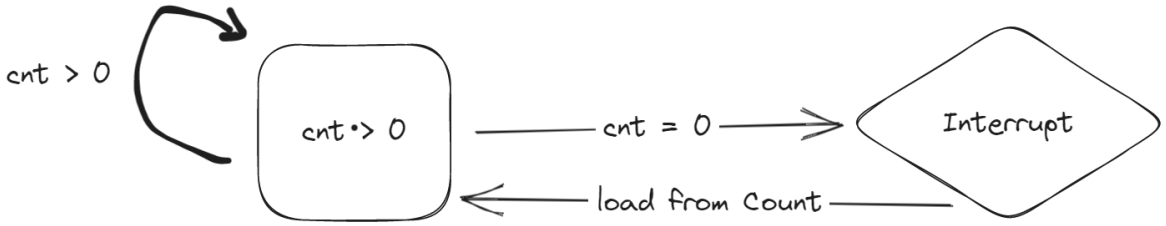
4

> 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

Mode0 是一个EN信号之后，倒数一段时间然后持续产生一个中断信号，得到响应后停止，等待下一个EN



Mode1 一个是每隔一段时间给出一个周期的中断信号



> 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？

会发生一个可能的问题在于，当异常处理程序结束之后，回到哪一条指令存在问题

因此，我们插入的空泡应该需要额外携带一些信息，比如对于以下指令序列：

```
lw $1, 0($0)   // Stage-M
nop            // Stage-E
add $2, $1, $1 // Stage-D
```

nop的宏观PC和ISDB标记应该与当前D级指令一样，这样当流水到

```
nop // M
add // E
```

nop在M级产生中断后，处理程序完成后我们应该回到ADD指令进行执行，所以真正的受害PC应该是ADD的宏观PC

6

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

这个问题是引入了中断和异常后出现的，对于该指令的延时槽如果发生了中断或异常

```
jalr $31, $31 // W
sw $1, 1($0)  // M
```

首先，M级的异常并不会影响W级的写入，那么在执行异常处理程序之前，$31已经被更新

之后，在执行完异常处理程序后，由于sw为延时槽，则重新会到跳转指令jalr，但是会跳转到更新后的$31，与无异常下的指令行为不符

7

> 请详细描述你的测试方案及测试数据构造策略。

手动 + 自动测试

使用了自己之前搭建的评测机，并进行了扩展

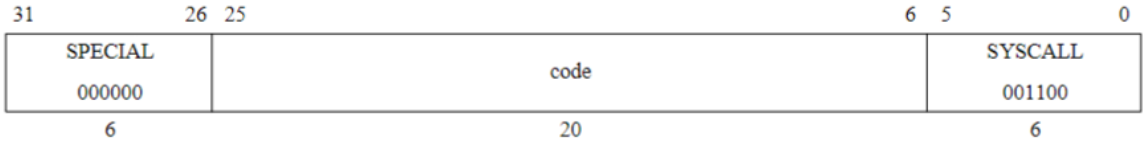
单独设计了一个handle.asm

之后，在保证指令相互会出现数据冲突的前提下，保证所有能出现异常的指令尽量出现异常

此外手搓了一些测试数据

Reference

System Call **SYSCALL**



Format: SYSCALL **MIPS32**

Purpose:

To cause a System Call exception

Description:

A system call exception occurs, immediately and unconditionally transferring control to the exception handler.

The *code* field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

Restrictions:

None

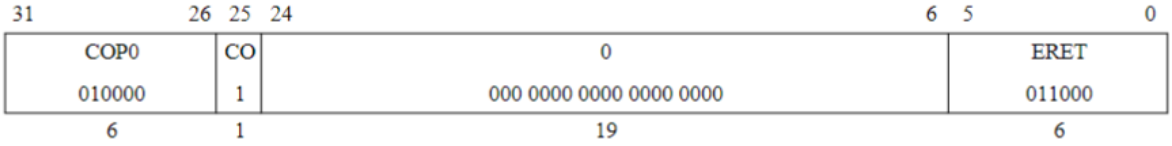
Operation:

```
SignalException(SystemCall)
```

Exceptions:

System Call

Exception Return **ERET**



Format: ERET **MIPS32**

Purpose:

To return from interrupt, exception, or error trap.

Description:

ERET clears execution and instruction hazards, conditionally restores $SRSCtl_{CSS}$ from $SRSCtl_{PSS}$ in a Release 2 implementation, and returns to the interrupted instruction at the completion of interrupt, exception, or error processing. ERET does not execute the next instruction (i.e., it has no delay slot).

Restrictions:

The operation of the processor is **UNDEFINED** if an ERET is executed in the delay slot of a branch or jump instruction.

An ERET placed between an LL and SC instruction will always cause the SC to fail.

ERET implements a software barrier that resolves all execution and instruction hazards created by Coprocessor 0 state changes (for Release 2 implementations, refer to the SYNCI instruction for additional information on resolving instruction hazards created by writing the instruction stream). The effects of this barrier are seen starting with the instruction fetch and decode of the instruction at the PC to which the ERET returns.

In a Release 2 implementation, ERET does not restore $SRSCtl_{CSS}$ from $SRSCtl_{PSS}$ if $Status_{BEV}$ = 1, or if $Status_{ERL}$ = 1 because any exception that sets $Status_{ERL}$ to 1 (Reset, Soft Reset, NMI, or cache error) does not save $SRSCtl_{CSS}$ in $SRSCtl_{PSS}$. If software sets $Status_{ERL}$ to 1, it must be aware of the operation of an ERET that may be subsequently executed.

27. MFC0：读 CP0 寄存器

编码	31 26	25 21	20 16	15 11	10 0
	COP0 010000	mfc0 00000	rt	rd	0 0 0000 0000
	6	5	5	5	11
格式	`mfc0 rt, rd`				
描述	`GPR[rt] ← CP0[rd]`				
操作	`GPR[rt] ← CP0[rd]`				
示例	`mfc0 $s1, $1`				
其他					

编码	31 26	25 21	20 16	15 11	10 0
	COP0 010000	mtc0 00100	rt	rd	0 0 0000 0000
	6	5	5	5	11
格式	`mtc0 rt, rd`				
描述	`CP0[rd] ← GPR[rt]`				
操作	`CP0[rd] ← GPR[rt]`				
示例	`mtc0 $s1, $1`				
其他					