P7设计文档

22373321 张瀚文

内部异常的判断

根据指导书中对异常的分类，对于流水线的`FDEM`级，每一级都有对应的异常可能会产生，据此进行处理

F	取指异常
D	未知指令错误
E	溢出
M	取数异常

据此生成Exccode，且每一级（除最开始的F级）其生成Exccode的原理为：

`assign new_Exccode = old_Exccode ? old_Exccode : 新生成的Exccode`

具体的判断只需要根据具体情况分析即可，可使用宏定义简化代码

CP0模块

CP0模块是P7的新模块，关键在于如何判断是否响应异常，以及各个控制信号等的更新。

```
wire IntReq; //异常来自外部的io控制器
assign IntReq = (|(HWInt && `IM)) && (!`EXL) && `IE;

wire ExcReq; //异常来自内部指令
assign ExcReq = (|ExcCodeIn) && (!`EXL);

assign Req = IntReq | ExcReq; //最终的中断请求信号
```

```
always@(posedge clk) begin
  if(reset) begin
     SR <= 32'b0;
     Cause <= 32'b0;
     EPC <= 32'b0;
     llBit <= 1'b0;
  end
  else if(Req) begin
     `EXL <= 1'b1;
      EPC <= BDIn ? (PC_M - 32'd4) : PC_M;
     `ExcCode <= IntReq ? 5'd0 : ExcCodeIn; //异常和中断同
     `BD <= BDIn;
  end
  else begin
     `IP <= HWInt;
     //swap
     if(ll_check) begin
        llBit <= 1'b1;
     end
     if(WE_cp0) begin
        case(A2)
           5'd12 : begin
              `IM <= W_Data[15:10];
              `EXL <= W_Data[1];
              `IE <= W_Data[0];
           end
           5'd14 : begin
              EPC <= W_Data;
           end
        endcase
     end
     if(EXLClr) begin
        `EXL <= 1'b0;
        llBit <= 1'b0;
     end
  end
end
```

系统桥模块

系统桥主要是进行外设与CPU的交互，注意写地址等的处理即可

```
/* choose which device */
wire [2:0] device_sel;
assign device_sel = ((processAddr >= `DM_BEGIN) && (processAddr <= `DM_END)) ? `DM_sel :
                    ((processAddr >= `T0_BEGIN) && (processAddr <= `T0_END)) ? `T0_sel :
                    ((processAddr >= `T1_BEGIN) && (processAddr <= `T1_END)) ? `T1_sel :
                    ((processAddr >= `INT_BEGIN) && (processAddr <= `INT_END)) ? `INT_sel : 3'b000;

/* choose readdata */
assign processReadData = (device_sel == `DM_sel) ? m_data_rdata :
                         (device_sel == `T0_sel) ? T0_Dout :
                         (device_sel == `T1_sel) ? T1_Dout : 32'b0;

/* HWInt */
assign HWInt = {3'b0,interrupt,T1_IRQ,T0_IRQ};

/* DM */
assign m_data_addr = processAddr;
assign m_data_wdata = processWriteData;
assign m_data_byteen = ((device_sel == `DM_sel)&&(!Req))? processByteen : 4'b0;

/* T0 */
assign T0_Addr = processAddr[31:2];
assign T0_we = (device_sel == `T0_sel) && (|processByteen);
assign T0_Din = processWriteData;

/* T1 */
assign T1_Addr = processAddr[31:2];
assign T1_we = (device_sel == `T1_sel) && (|processByteen);
assign T1_Din = processWriteData;

/* INT */
assign m_int_addr = processAddr;
assign m_int_byteen = (device_sel == `INT_sel) ? processByteen : 4'b0;
```

添加指令之后一定要在unreserved里面添加上新添的信号，否则会误判！

Controller里面的信号还是一样需要捋一遍！新指令注意Tnew等等的信号！RW不要忘记！

tItiu这个，Tuse和useA需要改，不要归类否则容易错

新的exccode先在macro里面定义

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?

- 键盘的工作方式一般有编程扫描方式和中断扫描方式两种。
  - 编程扫描方式是利用CPU在完成其他工作的空余，调用键盘扫描子程序，来响应按键输入要求。这种方式无论键盘上有无键按下，CPU都会定时扫描键盘，经常处于空扫描状态
  - 中断扫描方式即当键盘上有键按下时产生中断请求，CPU响应中断请求后，执行异常处理程序，在该处理程序中识别按键并做相应处理。
- 鼠标发送信号时触发中断请求，CPU响应中断，执行异常处理程序处理鼠标的输入信号。

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）

- 外部中断和异常都依赖于特定的中断处理程序，使用指定好的地址能够降低CPU内部设计的难度，便于维护。
- 可以由用户提供，但是用户自定义入口地址可能会导致对原有指令的覆盖或者修改，使得原有数据丢失。并且会增加CPU实现的难度

3.为何与外设通信需要 Bridge?

外设的种类是无穷无尽的，而 CPU 的指令集却是有限的。我们并不能总是因为新加入了一个外设，就专门为这个外设增加新的 CPU 指令。我们希望的是，尽管外设多种多样，但是 CPU 可以用统一的方法访问它们。为了实现这个目标，我们设计了系统桥。

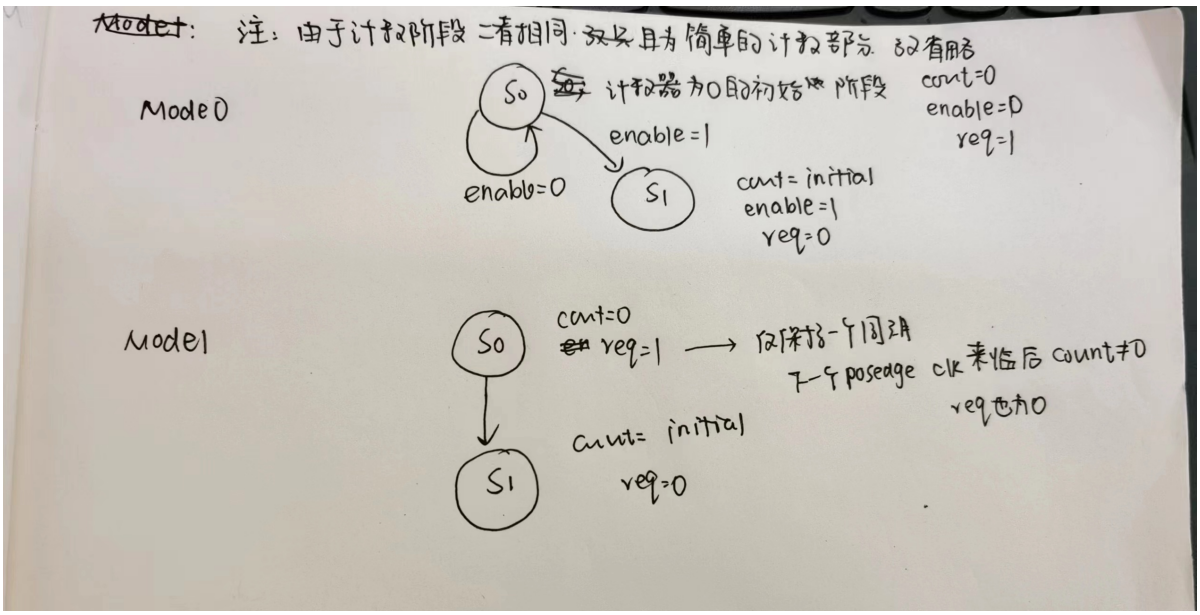
系统桥是连接 CPU 和外设的功能设备，它会给 CPU 提供一种接口，使得 CPU 可以像读写普通存储器一样（即按地址读写）来读写复杂多变的外设。系统桥统一且简化了 CPU 的对外接口，CPU 不必为每种外设单独提供接口，符合高内聚，低耦合的设计思想。

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。

两种模式产生中断的时间可能不同。

具体来说，mode0中中断持续的时间取决于enable什么时候被置位；而mode1只在计数器为0的那一个周期内产生中断信号，因为在下一个周期计数器就被更新为初值，不再为0.

状态转移图：



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？

宏观PC和写入EPC的PC都为错误的空值。

应当保留与正确PC值相关的信息，如PC值以及确定往EPC里具体写什么值的BDIn（是否为延迟槽指令）信号。

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

如果目标寄存器和源寄存器相同，会改变31号寄存器的值。倘若产生异常，重新执行这条指令时，不能得知最初31号寄存器的值，进而会进行错误的跳转。

7. 测试方案

集百家之所长进行了一些测试

- 首先是怎么添加异常处理程序，第一个方案是code里面通过增加nop使其地址达到4180，再在4180地址处开始异常处理程序的编写；第二个方案是在tb里添加

  `$readmemh("handler.txt", inst, 32'h460); //460 = (4180 - 3000) >> 2;`

  也即再新建一个txt文档，手动把异常处理程序的代码放在地址为4180的地方。

- 接下来说测试程序的构造

  主要测试了能否正确响应内外部异常中断。

  首先进行SR寄存器的值的设置以确保可以产生和响应中断

  ```
  #准备工作,允许外部中断以及全局中断使能
  ori $2,0xfc01
  mtc0 $2,$12
  ```

  接着对各种内部异常进行判断

  ```
  # Ov
  lui $1,0x8ffff
  lui $2,0x8ffff
  add $3,$1,$2
  ori $4,$0,0x0001 #这条用于检测是否正确从异常处理程序中跳转回来
  # RI
  ### 此处自己增加一条code码,注意不要和已经实现的指令重合
  # AdEL
  ## 未对齐
  lw $5,0x3415($0)
  ori $6,0x0001
  lh $5,0x3411($0)
  ori $6,0x0002
  ## lh/lb取计时器
  lh,$5,0x7f00($0)
  ori $6,0x0003
  lb,$5,0x7f10($0)
  ori $6,0x0004
  ```

```
## 超出范围
lh,$5,0xffff($0)
ori $6,0x0005
## 地址溢出
lui $7,0xefff
lh $5,0x000f($7)
ori $6,0x0006
# AdES
## sw/sh未对齐
ori $5,0xf000
sw $5,0x0001($0)
sh $5,0x0001($0)
ori $6,0x0007
## sh/sb写Timer寄存器的值
sh $5,0x7f00($0)
ori $6,0x0008
sb $5,0x7f10($0)
ori $6,0x0009
## 地址溢出
lui $7,0xefff
sh $5,0x000f($7)
ori $6,0x000a
## 超出范围
sh $5,0xffff($0)
ori $6,0x000b
## 向count寄存器存值
sw $5,0x7f0b($0)
ori $6,0x000c
sw $5,0x7f1b($0)
ori $6,0x000d
# syscall
syscall
```

同时编写了异常处理程序

```
# handler
mfc0 $k0,$12
mfc0 $k0,$13
mfc0 $k0,$14
# 以上用于对拍，  不是真正的异常处理程序中的一部分
mfc0 $k0,$13
ori  $k1, $zero, 0xfffc
and  $k0, $k0, $k1
addi $k0, $k0, 4
mtc0 $k0, $t6, 0
ori  $s0, $zero, 0xc1
addi $s6, $s6, 1
slt  $k1, $s5, $k0
bnez $k1, 0x303c
nop
beq  $s6, $s7, 0x303c
nop
eret
add  $k0, $zero, $s5 #检验延迟槽
```

```
mtc0 $k0, $t6, 0
eret
```

对于输出结果的正确性，主要采用波形图自行分析以及和同学们对拍的方式。