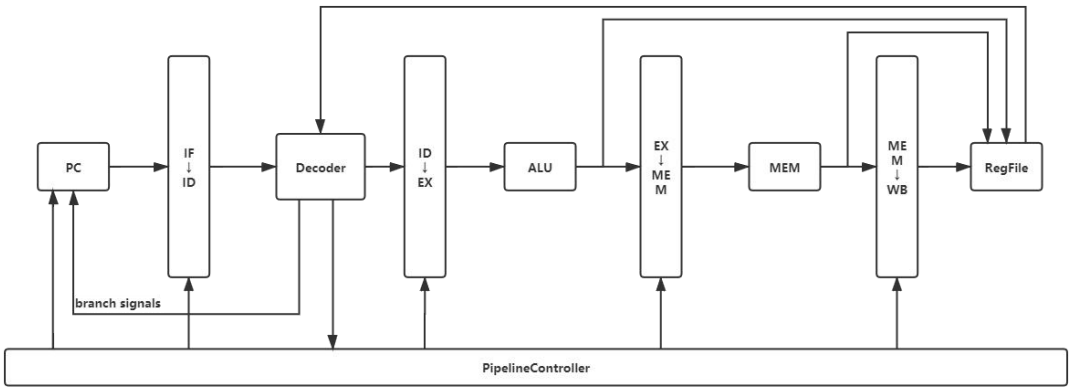
第一章 介绍

1.1 内容简介

本册内容为上册 TinyMIPS 的扩展教程，主要在之前的基础上增加了 HILO 寄存器、基于 IP 核的乘除法器以及 CP0 寄存器的设计。指令主要增加了数据移动指令、乘除法指令、特权指令以及自陷指令。

为了大家能够更好的使用“龙芯杯”系统能力培养大赛官方所提供的测试程序，在开始本册教学内容前请大家先在之前 TinyMIPS 的基础上实现 JR、ORI 以及 ANDI 指令。

1.2 TinyMIPS



本图对指导书上册中 TinyMIPS 的结构图进行了简化，主要将之前的 ReadProxy 和 RegFile 部件在图中进行合并(这里也建议大家可以在 TinyMIPS 的工程中对这两个部件进行合并)，并将 ID 中的 OpGen、FunctGen 等部件合并称为 Decoder 部件。

第二章 数据移动指令及 HOLO 寄存器的设计与实现

2.1 HILO 寄存器实现（HILO 部件）

通过 A03 指令文档可以看到，数据移动指令中的 4 条指令 MFHI、MFLO、MTHI、MTLO，均为对 HILO 寄存器进行读写的指令，因此在对这些指令进行实现之前需要先对 HILO 寄存器进行实现。

HILO 寄存器主要用于存放在运算指令中计算得出的结果，例如 32 位乘法指令中高 32 位存在 HI 寄存器中，低 32 位存在 LO 寄存器中；32 位除法指令中余数存放在 HI 寄存器中，除法结果存放在 LO 寄存器中。

对于 HILO 寄存器的实现与之前实现的 MIPS 通用寄存器部件 RegFile 以及 RegReadProxy 的实现类似。但是因为 HILO 部件中只存在 HILO 两个 32 位寄存器，因此在实现的时候更为简单。

HILO 寄存器在(包括 HILOReadProxy)流水线中的位置如图所示：

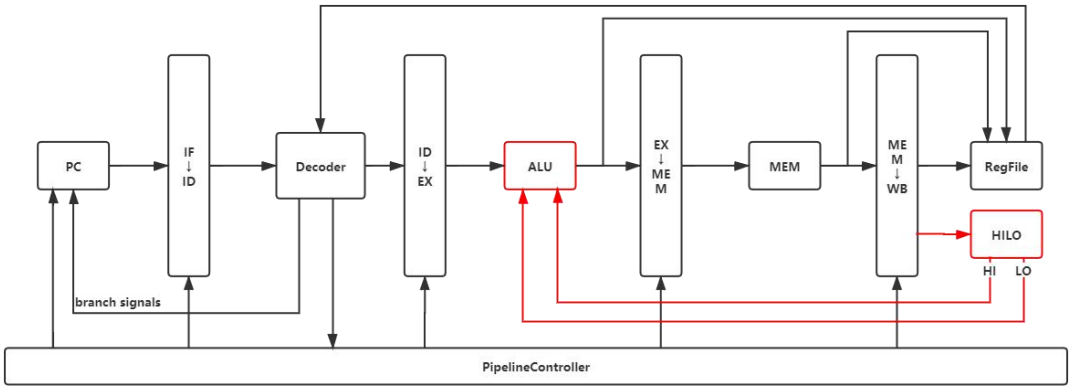


图 1 HILO 寄存器工作示意图

HILO 的接口定义如下：

HILO 的信号定义

名称	类型	宽度	方向	用途
clk	wire	1	i	时钟信号

rst	wire	1	i	复位信号
write_en	wire	1	i	HILO 寄存器写使能
hi_i	wire	32	i	HI 寄存器写入数据
lo_i	wire	32	i	LO 寄存器写入数据
hi_o	wire	32	o	HI 读出
lo_o	wire	32	o	LO 读出

HILO 具体实现如下：

```
reg [`DATA_BUS] hi;
reg [`DATA_BUS] lo;

assign hi_o = hi;
assign lo_o = lo;

always @(posedge clk) begin
    if(rst) begin
        hi <= 0;
        lo <= 0;
    end else if (write_en) begin
        hi <= hi_i;
        lo <= lo_i;
    end
end
```

2.2 HILO 寄存器实现(HILOReadProxy 部件)

根据图 1 可知，对 HILO 寄存器进行写操作是在 WB 阶段，而对 HILO 寄存器内数据进行读取和使用是在 EX 阶段，因此为了解决**数据相关**导致的冲突，需要引入 HILOReadProxy 部件进行 HILO 寄存器的**数据前递**，而前递的数据来自于 MEM 和 WB 阶段未能写入 HILO 寄存器的 HILO 寄存器数据。

HILOReadProxy 的接口定义如下：

HILOReadProxy 的信号定义

名称	类型	宽度	方向	用途
hi_i	wire	32	i	HI 寄存器数据
lo_i	wire	32	i	LO 寄存器数据
mem_hilo_write_en	wire	1	i	MEM 阶段写使能
mem_hi_i	wire	32	i	MEM 阶段 hi 数据
mem_lo_i	wire	32	i	MEM 阶段 lo 数据
wb_hilo_write_en	wire	1	i	WB 阶段写使能
wb_hi_i	wire	32	i	WB 阶段 hi 数据
wb_lo_i	wire	32	i	WB 阶段 lo 数据
hi_o	wire	32	o	hi 寄存器输出
lo_o	wire	32	o	lo 寄存器输出

具体实现如下：

```
assign hi_o =   mem_hilo_write_en ? mem_hi_i :
                wb_hilo_write_en  ? wb_hi_i  :
                hi_i;

assign lo_o =   mem_hilo_write_en ? mem_lo_i :
                wb_hilo_write_en  ? wb_lo_i  :
                lo_i;
```

2.3 增加数据移动指令

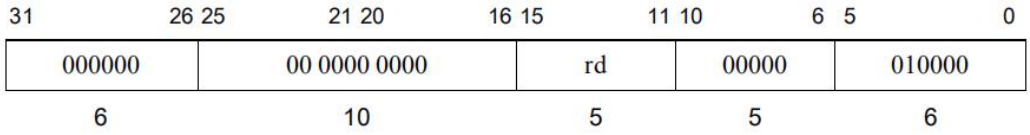
这里我们主要实现 MFHI 和 MTHI 两条指令，分别是读 HI 寄存器值与写 HI 寄存器值指令。

2.3.1 MFHI 实现

ID: 下图为 MFHI 指令的格式，该指令将从 HI 寄存器中读出的数据存入到[rd]

寄存器当中。通过观察该指令的操作码 OP 可以知道该指令的操作码为 SPECIAL，SPECAIL 指令默认读取([rs],[rt])的值，并写入[rd]中，因此不需要对 TinyMIPS 的 ID 阶段进行修改，只需要在 include 文件中加入该指令的 FUNCT(6'b010000)码即可。

MFHI



EX：通过图 1 的示意图可以知道 HILO 部件会将进行了数据前递处理后的 hi、lo 寄存器数据传输给 EX 阶段，同时 MFHI 是一条 R 型指令，可以通过 FUNCT 部分的编码对该指令进行判断。因此只需要对功能码 FUNCT 进行判断，将 EX 阶段输出的结果 result 赋值为当前 hi 寄存器的值即可。

对 EX 阶段新增接口如下：

名称	类型	宽度	方向	用途
hi_in	wire	32	i	HI 寄存器数据
lo_in	wire	32	i	LO 寄存器数据

对 EX 阶段代码修改如下：

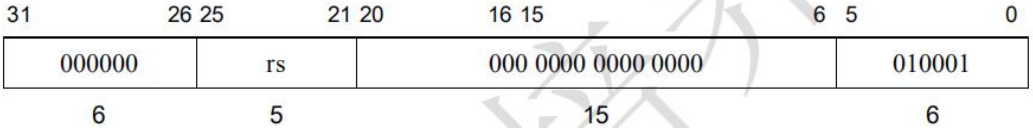
```
`FUNCT_MFHI: result <= hi_in;
```

完成到这步 MFHI 指令的添加就已经完成了，MEM 和 WB 阶段与之前的结构相同。

2.3.2 MTHI 实现

MTHI 是写 hi 寄存器指令，该指令的作用是将寄存器[rs]的值写入到 hi 寄存器当中。与之前实现的 MFHI 类似，对于 ID、MEM、WB 阶段并不需要进行过多的操作。`define FUNCT_MTHI 6'b010001

MTHI



EX：通过之前对于 HILO 部件的设计可以看出，为了写入 HILO 寄存器需要对 HILO 输入三个信号，分别为 HILO 的写使能、hi 寄存器的写数据、lo 寄存器的写数据，可以看出这里的写使能无法区分写入的具体为哪一个寄存器，因此需要对 hi、lo 同时进行写入。举个例子，如果当前寄存器 hi、lo 数据分别为 32’h20 和 32’h50，那么当我们想要通过 MTHI 写入 32’h40 这个数据时，就需要让写使能为 1，hi 写入数据为 32’h40，lo 写入数据为 32’h50。

（这里之所以需要同时写入也是为实现乘除法指令做准备）

对 EX 阶段新增接口如下：

名称	类型	宽度	方向	用途
hi_out	reg	32	o	写入 HI 寄存器数据
lo_out	reg	32	o	写入 LO 寄存器数据
hilo_write_en	reg	1	o	HILO 写使能

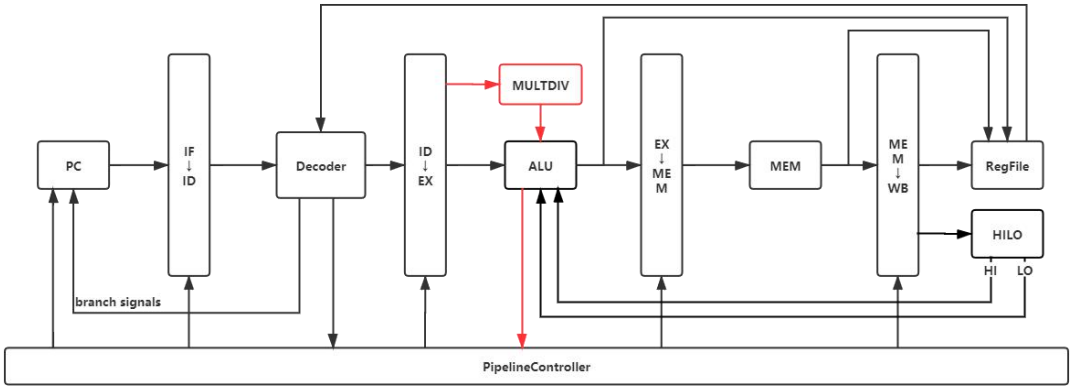
对 EX 阶段新增代码如下：

```
// HI & LO control
always @(*) begin
  case (funct)
    `FUNCT_MTHI: begin
      hilo_write_en <= 1;
      hi_out <= operand_1;
      lo_out <= lo_in;
    end
    default: begin
      hilo_write_en <= 0;
      hi_out <= hi_in;
      lo_out <= lo_in;
    end
  endcase
end
```

第三章 乘除法运算器与乘除法指令（略）

在之前本章之前我们已经实现了一些简单的运算指令，在本章主要对乘除法指令进行实现，并带领大家通过 IP 核实现一个简单的乘除法器。

3.1 乘除法运算器工作原理



如上图所示，乘除法器主要位于 EX 阶段，辅助 ALU 部件进行乘除法的计算。乘除法器的主要作用就是计算[rs][rt]寄存器数据的乘除法结果，并返回给 ALU 部件，通过 ALU 部件将结果传输给 HILO 寄存器。由于乘除法运算相比于加减法运算更为复杂，为了能够让 cpu 有更高的频率因此需要多个 cpu 周期的时间来进行乘除法的运算，那么在乘除法计算没有结束的时候就需要对流水线进行暂停，也就是图中 ALU->PipelineController 的红线所示。

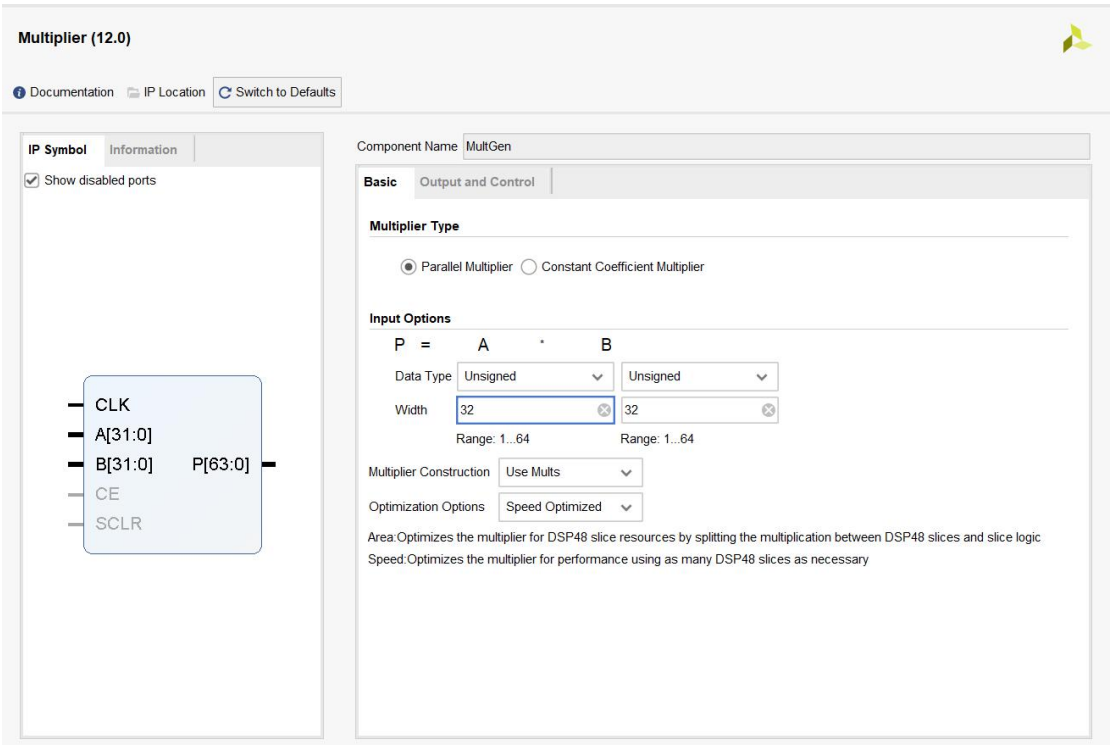
（数据前递没有在图上标出）

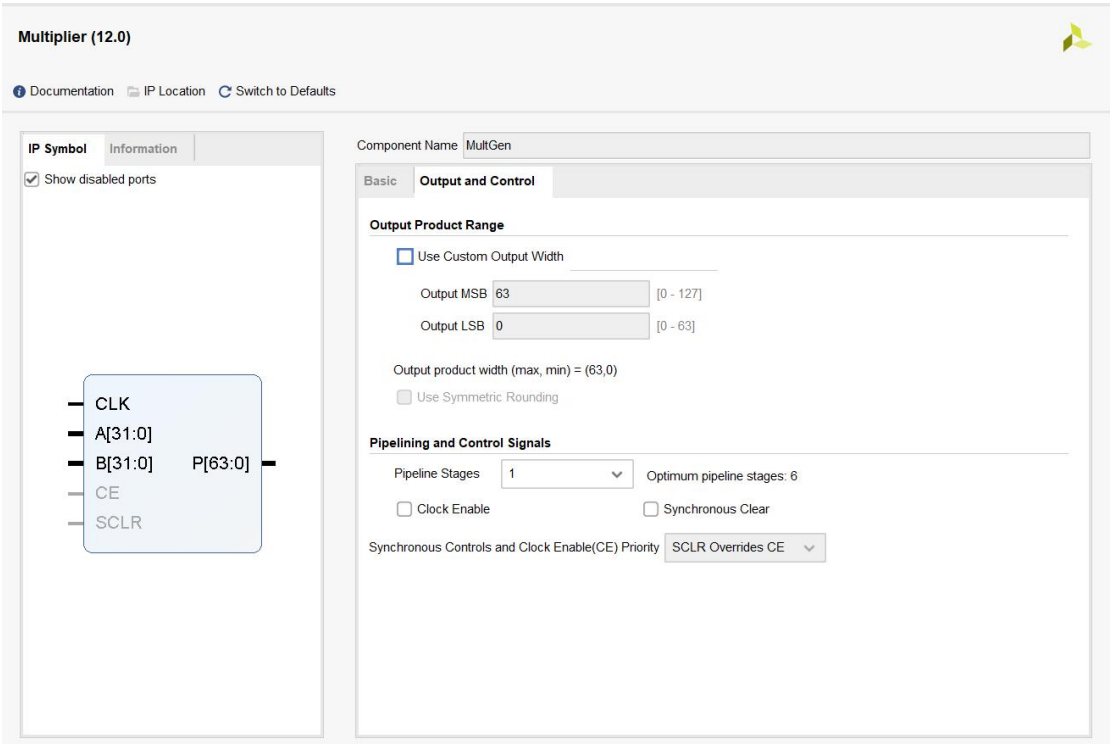
3.2 乘除法运算器的实现

那么为了能够实现一个乘除法器，就需要分别实现乘法计算器和除法计算器，而本册内容主要为了让大家了解 cpu 的运行过程，具体乘除法的计算大家可以自行设计，这里我们通过 vivado 提供的 IP 核进行实现。

3.2.1 乘法器

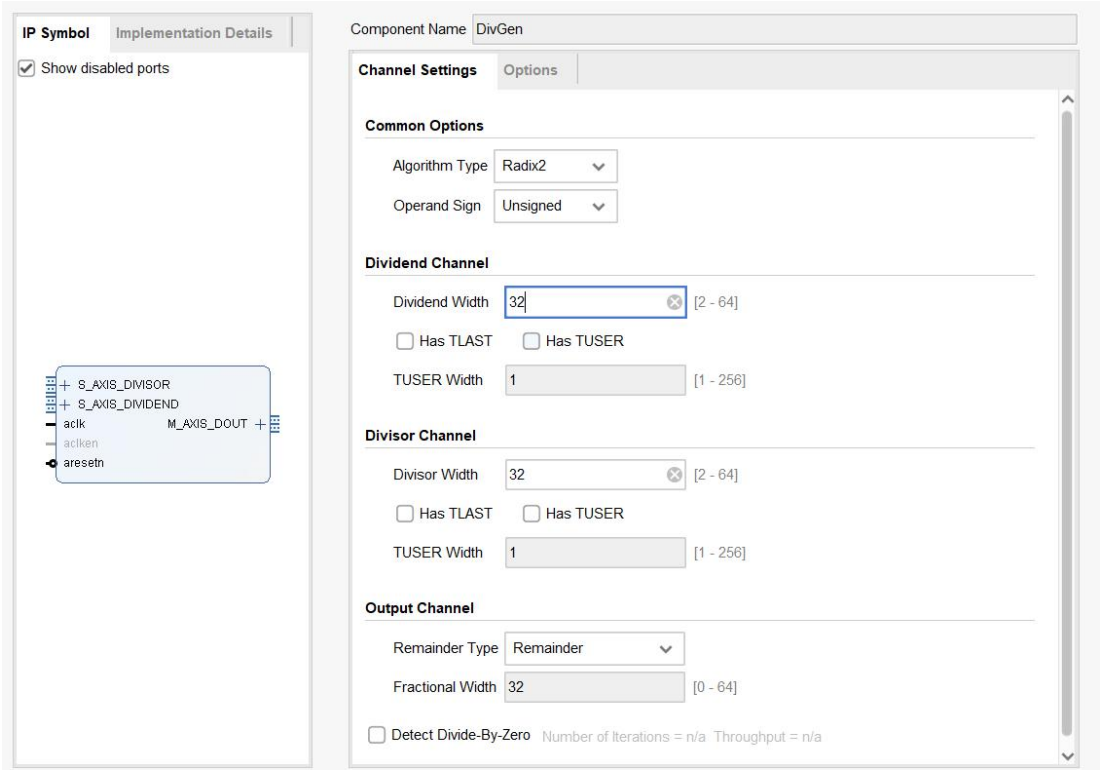
乘法器使用 Multipler 这一 IP 核进行构建，这里将乘法器设置为无符号乘法，默认 1 个周期后得到结果，具体设置参数如下：

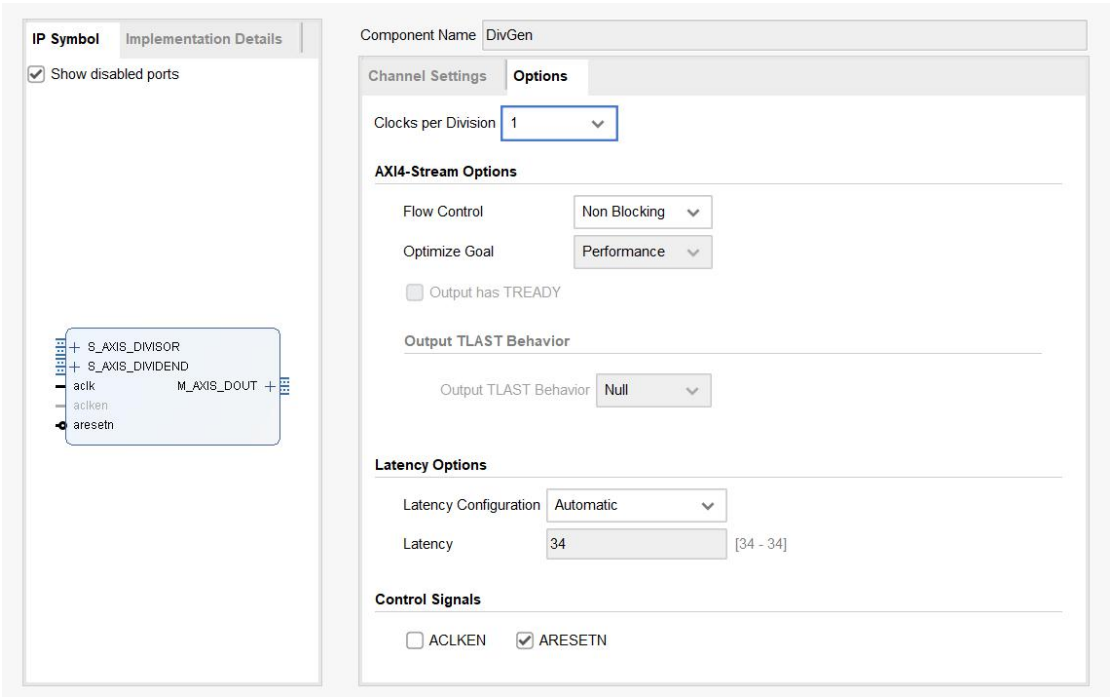




3.2.2 除法器

除法器使用 Divider Generator 这一 IP 进行构建，同样设置为无符号除法并将周期数设置为默认的 34 周期得到结果。大家也可以自行修改得到结果的周期数。具体设置参数如下：





3.2.3 整合

我们已经使用 Vivado 提供的 IP 核实现了乘除法器，但是可以发现之前实现的乘除法器都是无符号的，因此需要在整合的时候对有符号的情况进行判断，并手动的对有符号乘除法的结果进行修正。

同时因为乘除法器无法在当前周期得到结果，因此需要一个完成信号用于提示当前运算已经完成。

下面是 MULTDIV 部件的接口定义：

名称	类型	宽度	方向	用途
clk	wire	1	i	时钟信号
rst	wire	1	i	复位
stall_all	wire	1	i	core 外部暂停信号
funct	wire	32	i	功能码
operand_1	wire	32	i	操作数 1
operand_2	wire	32	i	操作数 2
done	wire	1	o	完成信号
result	reg	64	o	结果

详细代码请参考/cpu/stage/ex/MultDiv.v

（如果需要增加 cache，MultDiv 部件建议大家参考《自己动手写 CPU》使用状态机进行实现）

3.3 增加乘除法指令

通过 A03 文档中对乘除法指令(DIV、DIVU、MULT、MULTU)的描述可以知道乘除法指令都是对[rs]和[rt]两个寄存器进行乘除法运算，并将结果写入到 hi、lo 寄存器中。之所以结果需要写入到 hi、lo 寄存器中是由于 32 位乘法运算会最多产生 64 位的结果，而 32 位除法运算会分别产生 32 位商和 32 位余数。

下面为需要实现的乘除法指令的格式，可以看到乘除法指令全部为 R 型指令且 Opcode 为 special 类型，因此不需要对 ID 阶段进行修改，ID 阶段会读取[rs]、[rt]寄存器的数据。(注：大家记得补充下面指令的 FUNCT 码)

指令	31:26	25:21	20:16	15:6	5:0
DIV	6'b0	rs	rt	10'b0	011010
DIVU	6'b0	rs	rt	10'b0	011011
MULT	6'b0	rs	rt	10'b0	011000
MULTU	6'b0	rs	rt	10'b0	011001

EX：在 EX 阶段时需要计算乘除法的结果，计算结果部分由之前的乘除法器进行计算。得到结果后，EX.v 需要将乘除法器计算出的结果作为 hi、lo 寄存器写入数据进行输出，并将 hilo 写使能置为 1。

又因为乘除法器的计算过程中无法在一个周期内结束，因此需要 EX 阶段在计算完成之前发出流水线暂停请求，暂停 pc、if、id、ex。

（注：只有乘除法指令的时候需要暂停）

EX.v 增加接口如下：

名称	类型	宽度	方向	用途
mult_div_done	wire	1	i	乘除法器运算完成
mult_div_result	wire	64	i	乘除法器运算结果

stall_request	reg	1	o	EX 请求暂停信号

EX.v 增加代码如下：

```
// HI & LO control
always @(*) begin
  case (funct)
    `FUNCT_MULT, `FUNCT_MULTU,
    `FUNCT_DIV, `FUNCT_DIVU: begin
      hilo_write_en <= 1;
      hi_out <= mult_div_result[63:32];
      lo_out <= mult_div_result[31: 0];
    end
  endcase
end

always @(*) begin
  case (funct)
    `FUNCT_MULT, `FUNCT_MULTU,
    `FUNCT_DIV, `FUNCT_DIVU: begin
      stall_request <= !mult_div_done;
    end
    default: stall_request <= 0;
  endcase
end
```

在 EX.v 中增加 EX 的暂停请求信号后需要继续对 PipelineController 部件进行修改，PipelineController 增加接口如下：

名称	类型	宽度	方向	用途
request_from_ex	wire	1	i	EX 阶段暂停信号

PipelineController.v 新增代码如下：

```
else if (request_from_ex) begin
  stall <= 6'b001111;
end
```

至此就完成了乘除法器和乘除法指令的增加。

第四章 协处理器访问指令

4.1 协处理器介绍

协处理器用于表示处理器的一个可选部件，负责处理指令集的某个扩展，具有与处理器核独立的寄存器。MIPS32 中最多有 4 个协处理器，CP0~CP3，这里我们需要实现的 CP0 主要负责系统控制。（可以将 CP0 认为是多个状态寄存器）

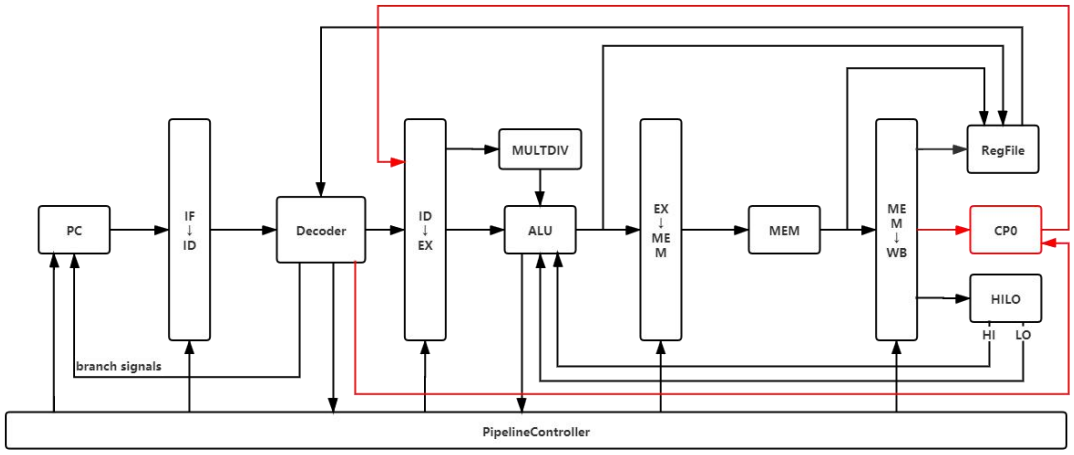
之前我们所实现的全部指令都可以认为是以某种运算指令，在实际中处理器还需要支持其他操作，例如配置 CPU 工作状态、异常处理、配置 TLB 等等，而这些操作就需要 CP0 来进行主要复杂。

4.2 协处理器的实现

这里我们按照 A03 文档的要求主要实现下面五个 CP0 中的寄存器，这里的地址与 Select 进行了合并。

（详细内容请阅读 A03 文档关于系统控制寄存器部分）

寄存器名称	寄存器地址	作用
BadVaddr	8'b01000000	记录最近一次地址例外的虚地址
Count	8'b01001000	处理器内部定时器
Status	8'b01100000	记录处理器状态
Cause	8'b01101000	记录最后一次例外原因
EPC	8'b01110000	记录例外处理后开始执行的指令的 PC



在此之前我们已经成功的实现了 RegFile 和 HILO 两个寄存器部件，从上图也可以看出 CP0 寄存器部件的功能目前来看与之前实现的功能类似，但是由于 CP0 中的寄存器存在只读的部分，因此建议这里在实现的过程中仔细参考 A03 文档进行实现。（数据前递没有在图上标出）

CP0 寄存器的接口定义：

名称	类型	宽度	方向	用途
clk	wire	1	i	时钟信号
rst	wire	1	i	复位
cp0_write_en	wire	1	i	cp0 寄存器写使能
cp0_read_addr	wire	8	i	cp0 寄存器读地址
cp0_write_addr	wire	8	i	cp0 寄存器写地址
cp0_write_data	wire	32	i	cp0 寄存器写数据
data_o	reg	32	o	cp0 读出结果

CP0.v 的具体实现如下：

```
reg[`DATA_BUS] reg_badvaddr;
reg[ 32 : 0]   reg_count;
reg[`DATA_BUS] reg_status;
reg[`DATA_BUS] reg_cause;
```

```
    reg[`DATA_BUS] reg_epc;
    // BADVADDR
    always @(posedge clk) begin
        if (rst) begin
            reg_badvaddr <= 32'h0;
        end
        // only read
        else begin
            reg_badvaddr <= reg_badvaddr;
        end
    end
    // COUNT
    always @(posedge clk) begin
        if (rst) begin
            reg_count <= 33'h0;
        end
        else if (cp0_write_en && cp0_write_addr == `CP0_REG_COUNT) begin
            reg_count <= {cp0_write_data, 1'b0};
        end
        else begin
            reg_count <= reg_count + 1;
        end
    end
    // STATUS
    always @(posedge clk) begin
        if (rst) begin
            reg_status <= 32'h0040ff00;
        end
        else if (cp0_write_en && cp0_write_addr == `CP0_REG_STATUS) begin
            reg_status[22] <= cp0_write_data[22];
            reg_status[15:8] <= cp0_write_data[15:8];
            reg_status[1:0] <= cp0_write_data[1:0];
        end
        else begin
            reg_cause <= reg_cause;
        end
    end
    // CAUSE
    always @(posedge clk) begin
        if (rst) begin
            reg_cause <= 32'h0;
```

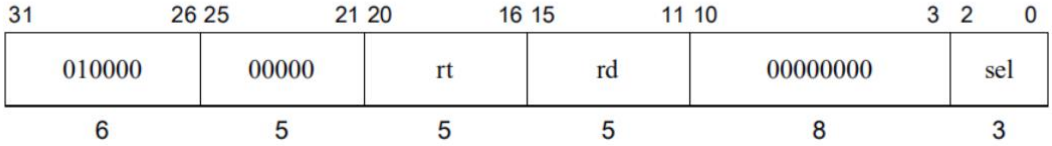
```
        end
        else if (cp0_write_en && cp0_write_addr == `CP0_REG_CAUSE) begin
            reg_cause[9:8] <= cp0_write_data[9:8];
        end
        else begin
            reg_cause <= reg_cause;
        end
    end
    // EPC
    always @(posedge clk) begin
        if (rst) begin
            reg_epc <= 32'h0;
        end
        else if (cp0_write_en && cp0_write_addr == `CP0_REG_EPC) begin
            reg_epc <= cp0_write_data;
        end
        else begin
            reg_epc <= reg_epc;
        end
    end
    always @(*) begin
        if(rst) begin
            data_o <= 32'b0;
        end
        else begin
            case (cp0_read_addr)
                `CP0_REG_BADVADDR:  data_o <= reg_badvaddr;
                `CP0_REG_COUNT: data_o <= reg_count[32:1];
                `CP0_REG_STATUS: data_o <= reg_status;
                `CP0_REG_CAUSE: data_o <= reg_cause;
                `CP0_REG_EPC: data_o <= reg_epc;
                default: data_o <= 0;
            endcase
        end
    end
```

（注：A03 文档的关于 CP0 部分参数设置与《自己动手写 CPU》不同）

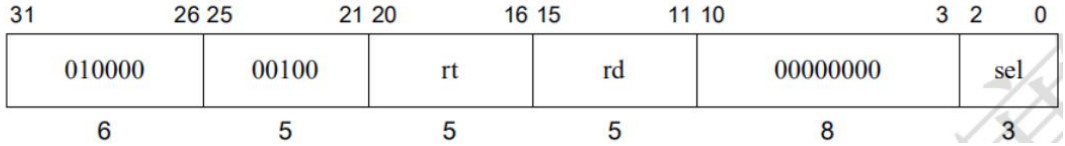
同样为了保证解决数据相关导致的冲突，需要引入 CP0ReadProxy 部件进行数据前递的处理，该部件与之前 HILO 和 RegFile 的处理方法类似，具体请参考 /cpu/core/storage/CP0ReadProxy.v

4.3 协处理器访问指令的实现

MFC0



MTC0



协处理器访问指令为 MFC0 和 MFT0，分别为读 CP0 寄存器和写 CP0 寄存器，其中读写的 CP0 寄存器为[rd,sel]。通过上面 MFC0 和 MFT0 的编码也可以看到这两条指令的 Opcode 相同，需要通过[25:21]部分的编码进行区分。

ID：与 RegFile 的读写类似，需要增加 cp0 读写相关的信号，并对指令进行解码来设置 cp0 的相关信号。因此这里需要增加一个 CP0 信号解码部件 CP0Gen，CP0Gen 的接口信号如下：

名称	类型	宽度	方向	用途
inst	wire	32	i	指令
op	wire	6	i	指令 op 段
rs	wire	5	i	指令 rs 段
rd	wire	5	i	指令 rd 段
reg_data_1	wire	32	i	regfile 读出的值
cp0_write_en	reg	1	o	cp0 寄存器写使能
cp0_read_en	reg	1	o	cp0 寄存器读使能
cp0_addr	reg	8	o	cp0 读/写地址
cp0_write_data	reg	32	o	cp0 寄存器写数据

CP0Gen 具体实现如下：

```
always @(*) begin
    case (op)
        `OP_CP0: begin
            if (rs == `CP0_MTC0 && inst[10:3] == 0) begin
                cp0_write_en <= 1;
                cp0_read_en <= 0;
                cp0_write_data <= reg_data_1;
                cp0_addr <= {rd, inst[2:0]};
            end
            else if (rs == `CP0_MFC0 && inst[10:3] == 0) begin
                cp0_write_en <= 0;
                cp0_read_en <= 1;
                cp0_write_data <= 0;
                cp0_addr <= {rd, inst[2:0]};
            end
            else begin
                cp0_write_en <= 0;
                cp0_read_en <= 0;
                cp0_write_data <= 0;
                cp0_addr <= 0;
            end
        end
        default:begin
            cp0_write_en <= 0;
            cp0_read_en <= 0;
            cp0_write_data <= 0;
            cp0_addr <= 0;
        end
    endcase
end
```

其余部分实现与 RegFile 的设计基本相同，ID 的其余解码部分与其他阶段的编写请大家自行实现。

第五章 异常相关指令的实现

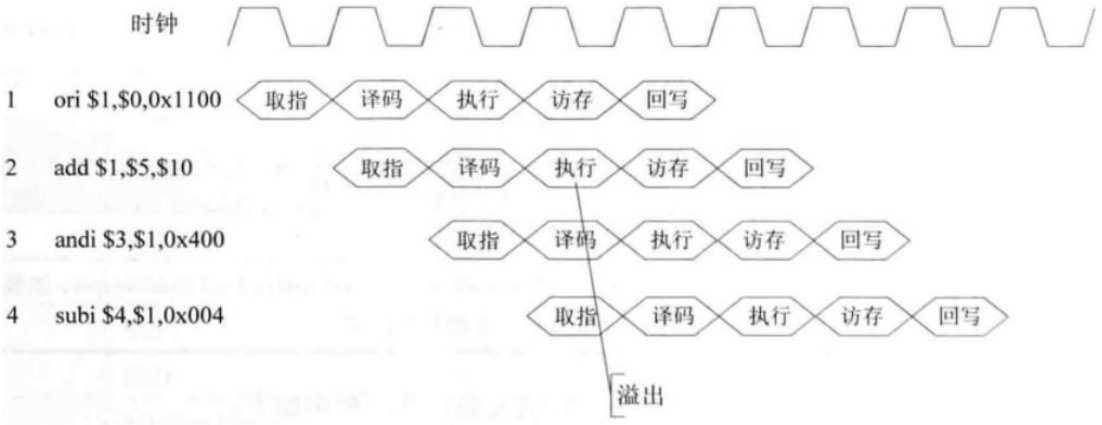
5.1 精确异常

在进行异常指令实现之前，需要先了解一下什么叫做“精确异常”。

当一个异常发生的时候，系统的顺序执行会被中断，此时会有若干条指令处于流水线上的不同阶段，处理器会转移到异常处理历程，异常处理结束后会返回原程序继续执行，因为不希望异常处理例程破坏原程序的正常执行，对于异常发生时，流水线上没有执行完的指令就必须记住它所处于的阶段，以便异常处理结束后能够恢复执行，这就是精确异常。

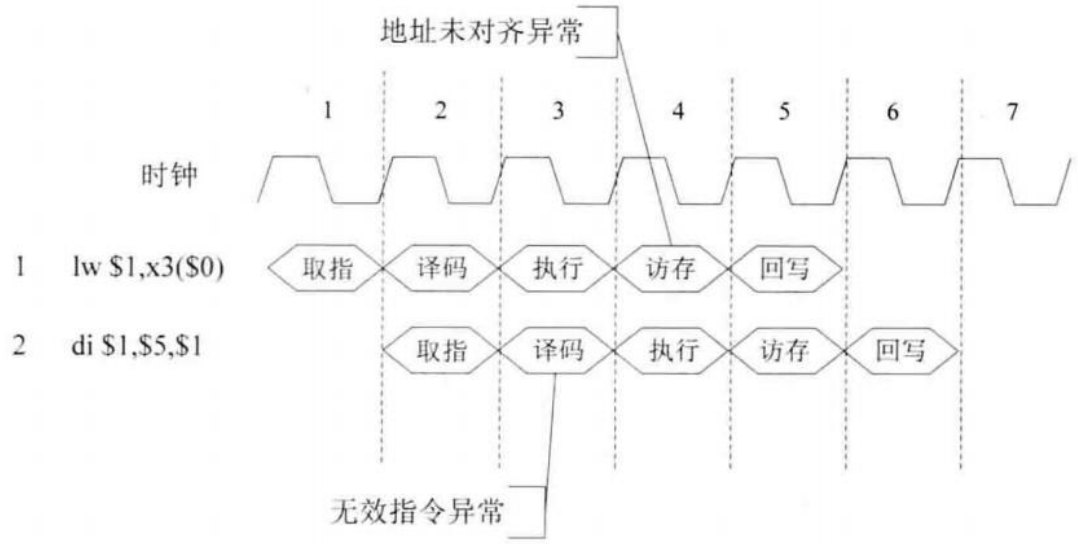
那么对于 TinyMIPS 来说，因为只有对寄存器和内存的读写才会导致系统的状态发生改变，因此想要做到精确异常，那么就需要在访存(MEM)阶段就得知异常的情况，才不会使得发生异常的指令改变了当前系统的状态。如果发生了异常那么就需要阻止异常指令执行，并将异常指令后面进入流水线的指令也进行删除，然后令 PC 跳转到异常处理程序的位置。

如下图所示，为了实现精确异常，如果 add 指令发生了溢出，那么我们需要将 add 指令以及 add 指令之后的指令进行清除，并让 pc 跳转至异常处理程序的位置执行异常处理，异常处理后再重新执行这条 add 指令。

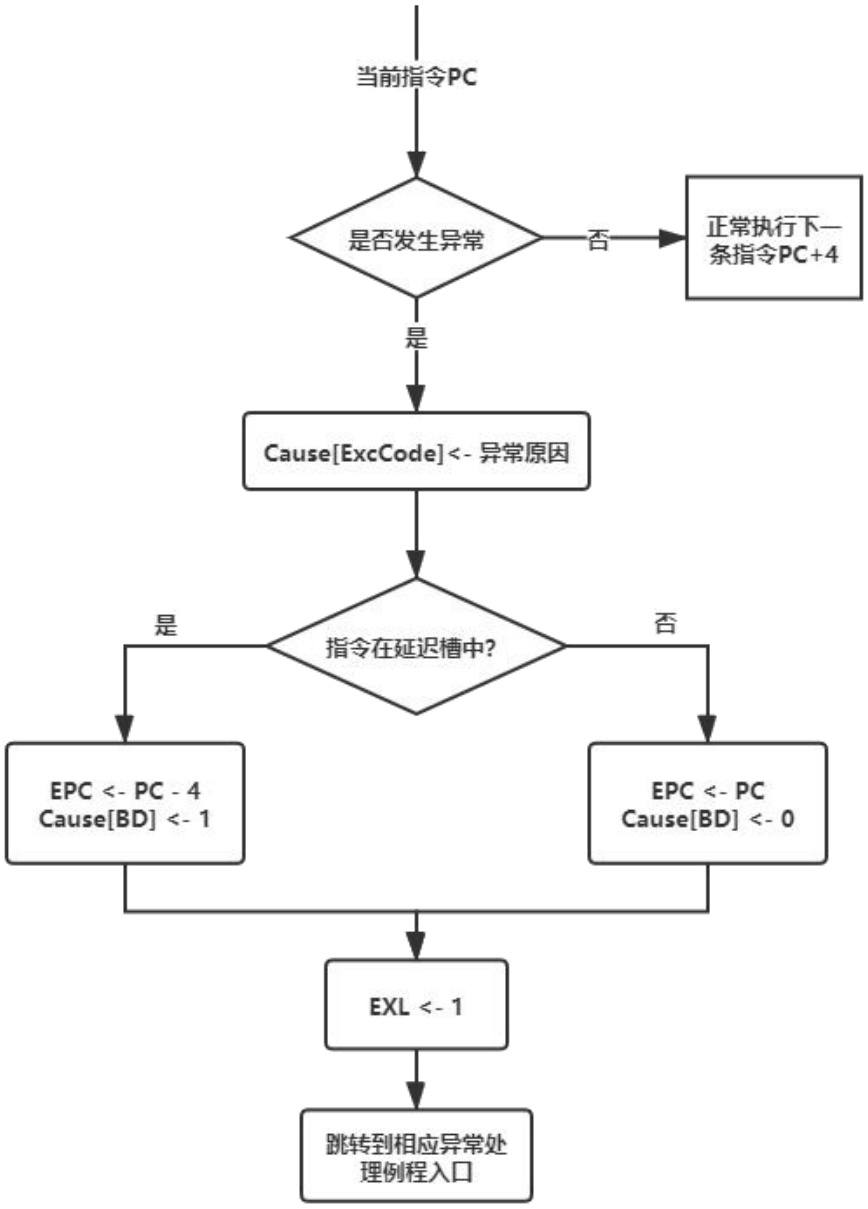


同时在流水线当中异常的发生可能并不是按顺序发生的，如下图所示第二条执行先在第二个 cpu 周期发生了译码错误的异常，而第一条指令在第四个 cpu

周期才会发生地址未对其异常，但是由于顺序执行的原则，这里需要先执行地址未对齐异常，也就是说要按照指令的执行顺序对异常进行处理，因此 TinyMIPS 将异常的处理处理设置在访存(MEM)阶段。



5.2 异常处理的过程

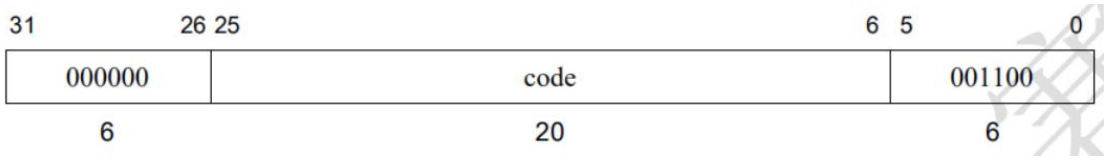


如上图所示，当指令发生异常后总共会对 3 个 CP0 寄存器进行修改，分别是 Cause、EPC 和 STATUS，其中 Cause 主要由 ExcCode 位负责记录当前指令异常原因和 BD 位记录异常指令是否是在延迟操中；EPC 寄存器负责记录异常处理结束后要从哪条指令重新开始执行，这里需要注意如果该指令是延迟槽指令，那么就需要从 PC-4 也就是延迟槽的前一条指令开始重新执行；Status[EXL]负责记录当前指令是否是处于例外状态。其中异常原因的代码大家可以查阅 A03 文档中关于中断与例外部分。

5.3 添加异常相关指令

5.3.1 系统调用指令 syscall

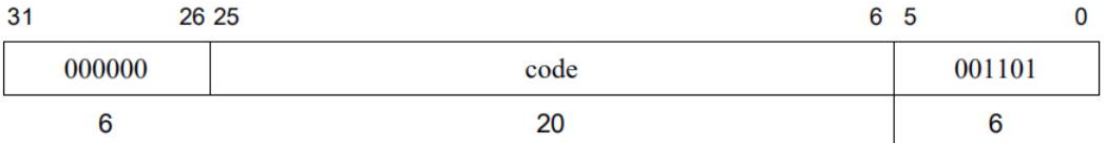
下图为 syscall 的指令格式，通过该图可知 syscall 的指令码时 SPECIAL，因此可以通过[5:0]部分的功能码对其进行判断，其中 code 部分可以不需要进行考虑。



该指令在 TinyMIPS 中的作用十分简单，主要通过使用 syscall 引发系统异常，从而进入异常处理例程。在 TinyMIPS 中所有的例外入口均为 0xbfc00380。

5.3.2 断点例外指令 break

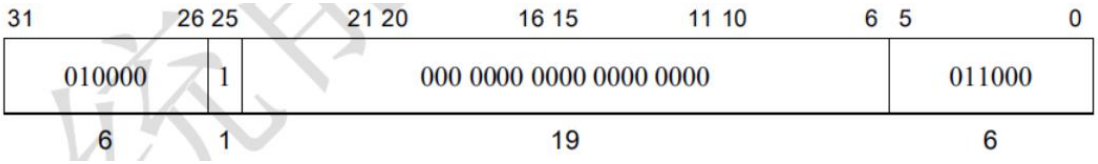
下图为 break 的指令格式，很好看出该指令与 syscall 指令格式相似。其作用也是触发系统异常，从而进入异常处理例程。



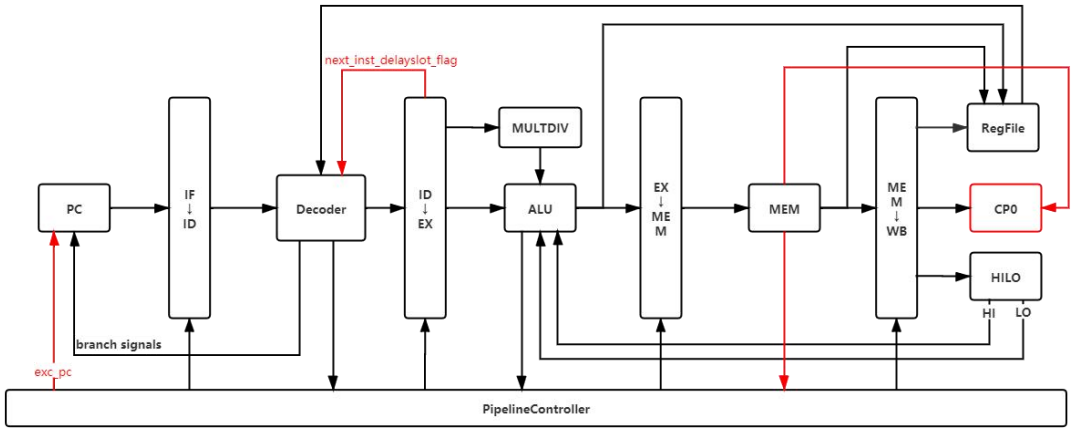
这里大家可能会感到疑惑，这两条作用类似的指令导致的例外如何被系统区分进行处理，回到之前的异常处理流程图就可以知道，在异常处理的过程中会将 Cause 的 ExcCode 字段设置成异常编码，这样异常处理例程就可以区分并进行处理，并完成相应指令的内容。

5.3.3 异常返回指令 eret

下图为 eret 指令的格式，eret 指令的处理与其他例外的处理有所区别，该指令作用为从中断或例外处理中返回到原程序，处理 eret 的时候首先要将 status[EXL]至为 0，表示结束异常处理。同时将 EPC 中的地址赋值给 pc 寄存器，也就是让 pc 回到异常处理开始前的位置继续执行。该指令一般处于异常处理例程的最后，用于让 pc 返回到异常开始前的位置。



5.3.4 具体实现



（1） 流水线先在 ID 阶段对指令进行判断是否是调用指令 syscall、break、eret 或者保留指令例外，如果发生则将对应 flag 标志为 1 并向下一阶段传递。 同时在 ID 阶段如果当前指令是一条跳转指令，则会将信号 next_inst_delayslot_flag 标志为 1，表示下一条指令是延迟槽内指令。（延迟槽相关知识请参考上册）

（2） 在 EX 阶段会进一步判断是否有整型溢出例外，如果发生则会将对应 flag 标志为 1 并向下一阶段传递。并将 next_inst_delayslot_flag 信号传递回 ID 阶段告知 ID 阶段当前指令是否在延迟槽内。

（3） 在 MEM 阶段首先会判断是否发生了地址错误例外，如果发生就将对应 flag 设置为 1，之后将所有例外对应的 flag 传输给 CP0 模块。同时要将例外 flag 信号传递给 PipelineController 模块，如果出现了需要处理的例外或异常，那么就需要通过 PipelineController 模块送出 flush 信号送到 PC 和各阶段中寄存器中，消除异常指令以及其后的指令，同时还要生成异常处理地址送到 PC 中。

（4） 如果发生异常，协处理器 CP0 模块会对响应需要修改的寄存器的值进行修改。

注：该部分不只适用于异常相关指令的添加，也适用于其他指令发生例外的情况

取指阶段

对于 PC 模块需要增加 flush 刷新信号和 exc_pc 发生异常跳转地址，这里需要注意 flush 信号的优先级要比 branch_flag 和 stall_pc 的优先级更高。如果发生了异常就需要将当前流水线内 MEM 阶段及以前的指令全部清除，重新从 exc_pc 处开始执行。

修改后 PC 模块代码如下：

```
  // generate value of next PC
  always @(*) begin
    if (flush) begin
      next_pc <= exc_pc;
    end
    else if (!stall_pc) begin
      if (branch_flag) begin
        next_pc <= branch_addr;
      end
      else begin
        next_pc <= pc + 4;
      end
    end
    else begin
      // pc & rom_addr stall
      next_pc <= pc;
    end
  end

  always @(posedge clk) begin
    if (!rom_en) begin
      pc <= `INIT_PC - 4;
    end
    else if (flush || !stall_pc) begin
      pc <= next_pc;
    end
  end
```

译码阶段

在译码阶段需要对异常相关指令进行译码，因此在 ID 模块增加了一个关于异常指令译码的模块 ExceptionGen，该模块主要用于对指令异常进行判断，从而生成相对应的异常信号。

ExceptionGen 模块代码如下：

```
module ExceptionGen (
    input       [`INST_BUS]     inst,
    input       [`INST_OP_BUS]  op,
    input       [`FUNCT_BUS]    funct,

    output                      eret_flag,
    output                      syscall_flag,
    output                      break_flag
);

    assign eret_flag = (inst == `CP0_ERET_FULL) ? 1 : 0;
    assign syscall_flag = (op == `OP_SPECIAL && funct == `FUNCT_SYSCALL) ? 1 : 0;
    assign break_flag = (op == `OP_SPECIAL && funct == `FUNCT_BREAK) ? 1 : 0;
```

同时因为异常指令的加入，在发生异常的时候我们需要知道该指令是否是延迟槽指令，因此在 BranchGen 中需要增加 next_inst_delayslot_flag 信号，如果当前指令是跳转指令，无论是否发生了跳转下一条指令均为延迟槽指令，需要将 next_inst_delayslot_flag 信号设置为 1。

增加 next_inst_delayslot_flag 信号后，需要在下一个周期再传输回 ID 阶段，用于告知 ID 模块当前指令是延迟槽内的指令，因此在 ID 阶段加入 delayslot_flag_in 和 delayslot_flag_out 接口，用于接收来自 EX 阶段的 next_inst_delayslot_flag 信号。

```
assign delayslot_flag_out = delayslot_flag_in;
```

执行阶段

如果需要增加溢出例外需要在该部分进行判断，请大家自行添加。

注：发生溢出例外的加法指令不能写回寄存器堆。

访存阶段

在访存阶段会将各异常相关信号传输给 PipelineController 模块，并通过 PipelineController 模块生成 exc_pc 和 flush 信号，实现对流水线的控制。

PipelineController 增加接口如下：

名称	类型	宽度	方向	用途
cp0_epc	wire	32	i	EPC 寄存器值
eret_flag	wire	1	i	发生 eret 例外
syscall_flag	wire	1	i	发生 syscall 例外
break_flag	wire	1	i	发生 break 例外
flush	wire	1	o	流水线刷新信号
exc_pc	reg	32	o	异常跳转 pc

PipelineController 增加代码如下：

```
assign flush = stall_all ? 0 : (eret_flag ||
                                syscall_flag || break_flag) ? 1 : 0;
always @(*) begin
  if (eret_flag) begin
    exc_pc <= cp0_epc;
  end
  else if (syscall_flag || break_flag) begin
    exc_pc <= 32'hbfc0_0380;
  end
  else begin
    exc_pc <= 32'hbfc0_0000;
  end
end
```

在 MEM 模块中还可能需要增加关于地址错误的例外判断，请大家自行添加。

CP0 寄存器

CP0 寄存器在接收到异常相关信号后，需要对相应的寄存器进行修改，这里只需要根据相关异常修改相应寄存器即可。需要注意，接口在加入异常信号的同时，还需要增加延迟槽判断信号。并且因为 PipelineController 模块在发生 eret 例外时可能需要 EPC 寄存器的数据对 pc 的跳转地址进行更新，还要增加 EPC 寄存器的读取信号。（EPC 寄存器的值也需要进行数据前递处理）

CP0 增加接口定义如下：

名称	类型	宽度	方向	用途
eret_flag	wire	1	i	发生 eret 例外
syscall_flag	wire	1	i	发生 syscall 例外
break_flag	wire	1	i	发生 break 例外
delayslot_flag	wire	1	i	是否在延迟槽中
current_pc_addr	wire	32	i	当前 pc
epc_o	wire	32	o	epc 的值

CP0 增加代码如下：

```
assign epc_o = reg_epc;

// STATUS
always @(posedge clk) begin
    if (rst) begin
        reg_status <= 32'h0040ff00;
    end
    else if (break_flag || syscall_flag) begin
        reg_status[1] <= 1;
    end
    else if (eret_flag) begin
        reg_status[1] <= 0;
    end
    else if (cp0_write_en && cp0_write_addr == `CP0_REG_STATUS) begin
        reg_status[22] <= cp0_write_data[22];
```

```
            reg_status[15:8] <= cp0_write_data[15:8];
            reg_status[1:0] <= cp0_write_data[1:0];
        end
        else begin
            reg_cause <= reg_cause;
        end
    end

    // CAUSE
    always @(posedge clk) begin
        if (rst) begin
            reg_cause <= 32'h0;
        end
        else if (break_flag || syscall_flag) begin
            reg_cause[31] <= delayslot_flag;
            reg_cause[6:2] <= break_flag ? `CP0_EXCCODE_BP :
                              syscall_flag ? `CP0_EXCCODE_SYS : 0;
        end
        else if (cp0_write_en && cp0_write_addr == `CP0_REG_CAUSE) begin
            reg_cause[9:8] <= cp0_write_data[9:8];
        end
        else begin
            reg_cause <= reg_cause;
        end
    end

    // EPC
    always @(posedge clk) begin
        if (rst) begin
            reg_epc <= 32'h0;
        end
        else if (break_flag || syscall_flag) begin
            reg_epc <= exc_epc;
        end
        else if (cp0_write_en && cp0_write_addr == `CP0_REG_EPC) begin
            reg_epc <= cp0_write_data;
        end
        else begin
            reg_epc <= reg_epc;
        end
    end
```

PipelineDeliver 模块

因为在增加了异常指令之后需要对 IF/ID、ID/EX、EX/MEM、MEM/WB 进行刷新，因此首先需要对 PipelineDeliver 模块进行修改，在该模块中加入 flush 信号的接口，如果发生 flush 就对寄存器进行重置。

PipelineDeliver 修改后代码如下：

```
always @(posedge clk) begin
  if (rst) begin
    out <= 0;
  end
  else if (flush) begin
    out <= 0;
  end
  else if (stall_current_stage && !stall_next_stage) begin
    out <= 0;
  end
  else if (!stall_current_stage) begin
    out <= in;
  end
end
```

修改该模块后大家记得对使用该模块的部分进行修改。

附件

自行搭建编译环境

需要自行搭建编译环境的同学请参考下面这篇博客

https://skyblond.info/archives/737.html

TinyMIPS 新增指令

OpCode	Description
JR	N/A
ORI	N/A
ANDI	N/A
MFHI	N/A
MFLO	N/A
MTHI	N/A
MTLO	N/A
DIV	N/A
DIVU	N/A
MULT	N/A
MULTU	N/A
MFC0	N/A
MTC0	N/A
ERET	N/A
BREAK	N/A
SYSCALL	N/A

代码结构

```
├──cpu
    ├──bus
    ├──core                          // Core 部分代码
    ├──include                       // 信号定义
    ├──ip                            // IP 核文件
    │   ├──DivGen
    │   ├──InternalCrossbar
    │   └──MultGen
    ├──mmu
    ├──sim
    └──testbench                     // 测试用文件
```

testbench 中包含 golden_trace.txt、inst_ram.coe、start.S 和 test.s 四个文件，这些文件是针对新增指令(包含 JR、ORI、ANDI)进行验证可能需要的测试文件。（MFC0、MTC0、ERET 没有单独测试用例）

在使用的时侯需要将 CDE/cpu132_gettrace/golden_trace.txt 和 CDE/soft/func/obj/inst_ram.coe 两个文件替换成 testbench 中提供的相应文件即可。

start.S 内是进行编译前的汇编代码，标注了使用了“龙芯杯”提供的具体测试用例编号。

大家可以通过 test.s 对错误的 pc 地址的实际指令进行查看。