**北京科技大学 计算机与通信工程学院**

**课程设计报告**

**课程名称**： 计算机组成原理课程设计

**学生姓名**： 刘伟浩

**专 业**： 计算机科学与技术

**班 级**： 计202

**学 号**： 42024093

**指导教师**： 张磊

**报告成绩**：\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**实验地点**： 腾讯会议+线下机电楼304

**实验时间**：2022年9月11日----2022年11月6日

**北京科技大学实验报告**

学院： 计算机与通信工程学院 专业：计算机科学与技术 班级：计202

姓名： 刘伟浩 学号： 42024093 实验日期：2022 年 9 月 11 日

**一、课设目的与要求**

* 学会处理器的设计方法：单周期/流水线。
* 掌握处理器设计过程中指令扩展的方法。
* 能够运用现代工具独立实现一个完整的处理器。
* 了解处理器功能测试的方法：仿真测试及FPGA测试。
* 计算机系统观的建立，对所设计的处理器在整个计算机系统中的位置有所了解。

**二、实验设备（环境）及要求**

龙芯实验箱一体化实验平台/CG平台/流水线CPU关键技术虚拟仿真实验平台

OS：Win10 64位

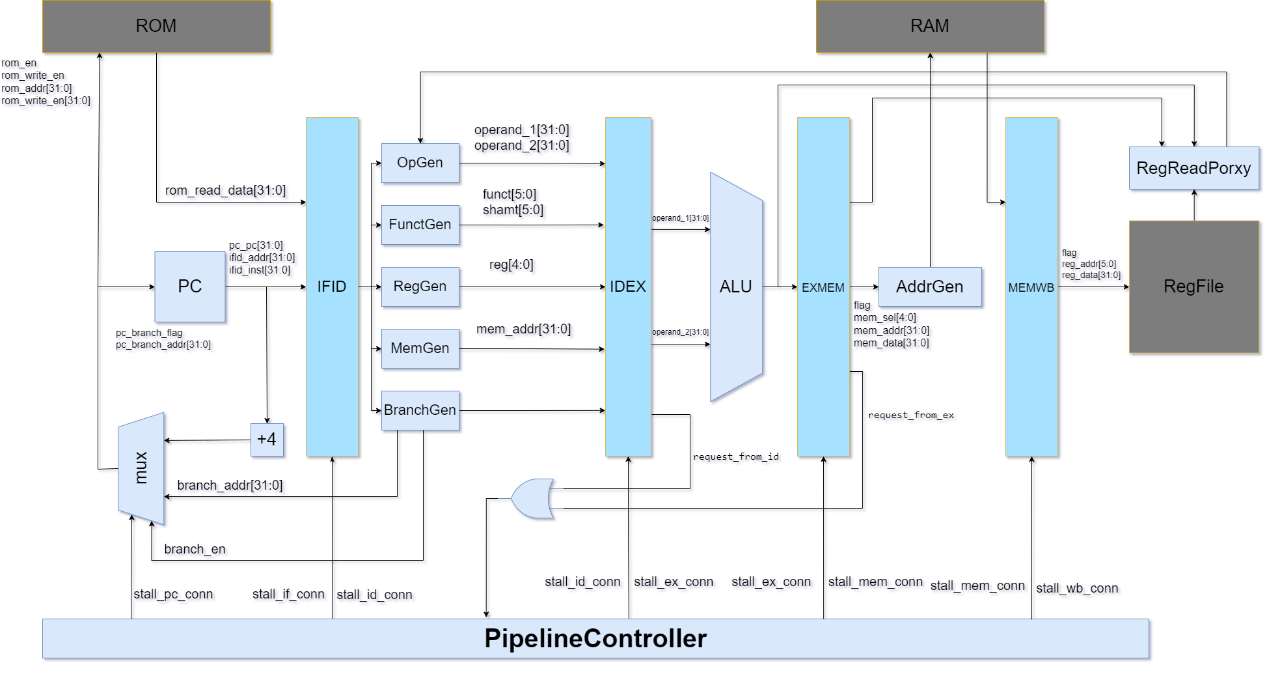
Software：Vivado2018.3开发工具

VirtualBox虚拟机+Ubuntu16.04.6（正文用五号宋体）

**三、设计过程与结果分析**

**Part 1 个人任务，代码阅读分析**

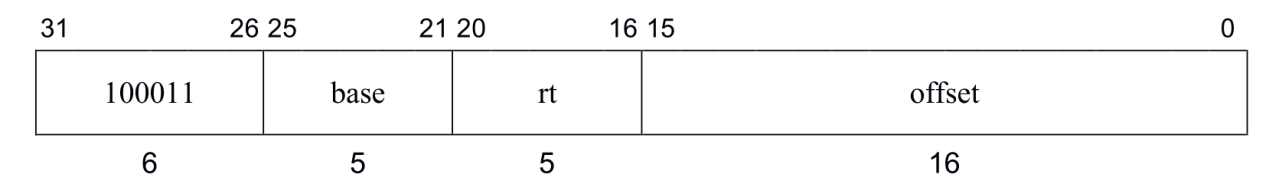
**1 TinyMIPS总体结构框图**（自己动手重新画图，推荐画图工具：gliffy或draw.io或者其它）



**2 LW指令（典型指令）的设计过程**

**2.1指令格式**

这是一条I型指令，格式如下：



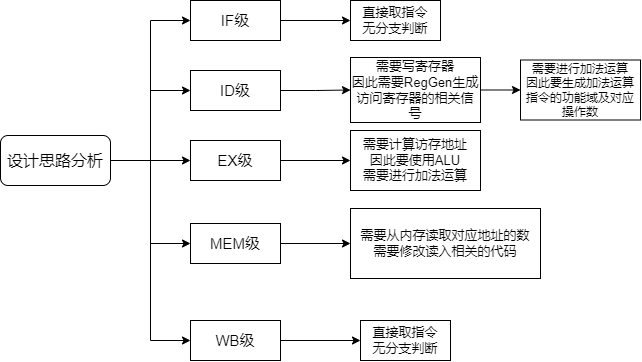
汇编格式：LW rt, offset(base)

功能描述：将 base 寄存器的值加上符号扩展后的立即数 offset 得到访存的虚地址，如果地址不是 4 的整数倍则触发地址错例外，否则据此虚地址从存储器中读取连续 4 个字节的值并进行符号扩展，写入到rt寄存器中。

**2.2设计分析与实现代码（需有指令设计分析过程图,实现代码需有恰当的注释）**

LW指令是一条访存指令，并且其寻址方式是寄存器间接寻址，因此也涉及ALU运算。从MEM级取得存储器数后需要在WB级写回rt寄存器。综上，LW指令涉及全部的5个流水级，需要在每个流水级考虑该级需要完成的功能。

设计分析过程图如下：



首先在IF级，直接读取指令，由于不需要跳转，因此PC级无特殊判断。

在ID级，对于从内存读入的数据，需要写回到寄存器堆中，因此我们需要设置寄存器堆的写使能信号。同时需要读出作为base地址的rs寄存器的值。

FunctGen中对应代码如下：

// generate read address

  always @(\*) begin

case (op)

...

      // memory accessing

      `OP\_LW: begin

        reg\_read\_en\_1 <= 1; // rs寄存器读使能

        reg\_read\_en\_2 <= 0;

        reg\_addr\_1 <= rs; // 读寄存器中的rs寄存器

        reg\_addr\_2 <= 0;

      end

...

end

// generate write address

  always @(\*) begin

case (op)

...

      `OP\_LW: begin

        reg\_write\_en <= 1; // rt寄存器写使能

        reg\_write\_addr <= rt; // 写入rt寄存器

      end

...

  end

同时，由于需要计算读取内存的地址signed\_ext(offset) + GPR(base)，因此需要操作数生成相关信号与FUNCT域相关信号。

如下是 OperandGen中的相关代码：

  // generate operand\_1

  always @(\*) begin

case (op)

...

      // memory accessing

      `OP\_LW: begin

        operand\_1 <= reg\_data\_1; //运算的数来自rs寄存器

      end

...

    endcase

  end

  // generate operand\_2

  always @(\*) begin

    case (op)

      ...

      // memory accessing

      `OP\_LW: begin

        operand\_2 <= sign\_ext\_imm; //运算数offset的符号扩展

      end

  ...

    endcase

  end

如下是FunctGen中的相关代码：

// generating FUNCT signal in order for the ALU to perform operations

  always @(\*) begin

case (op)

...

      `OP\_LW: funct <= `FUNCT\_ADDU; // 生成无符号数加的信号

      ...

    endcase

  end

EX级直接执行两个数的相加，因此直接考虑MEM级对内存的访问，这里要生成访存的读使能型号，读取的符号信号与选择信号。

  always @(\*) begin

    case (op)

     `OP\_LW: mem\_read\_flag <= 1; // 读使能信号

      ...

    endcase

  end

  always @(\*) begin

    case (op)

      `OP\_LW: mem\_sign\_ext\_flag <= 1; // 读入有符号数

      ...

    endcase

  end

  // mem\_sel: lb & sb -> 1, lw & sw -> 1111

  always @(\*) begin

case (op)

...

      `OP\_LW: mem\_sel <= 4'b1111; // 读取的是一个字，所以选取4个存储体

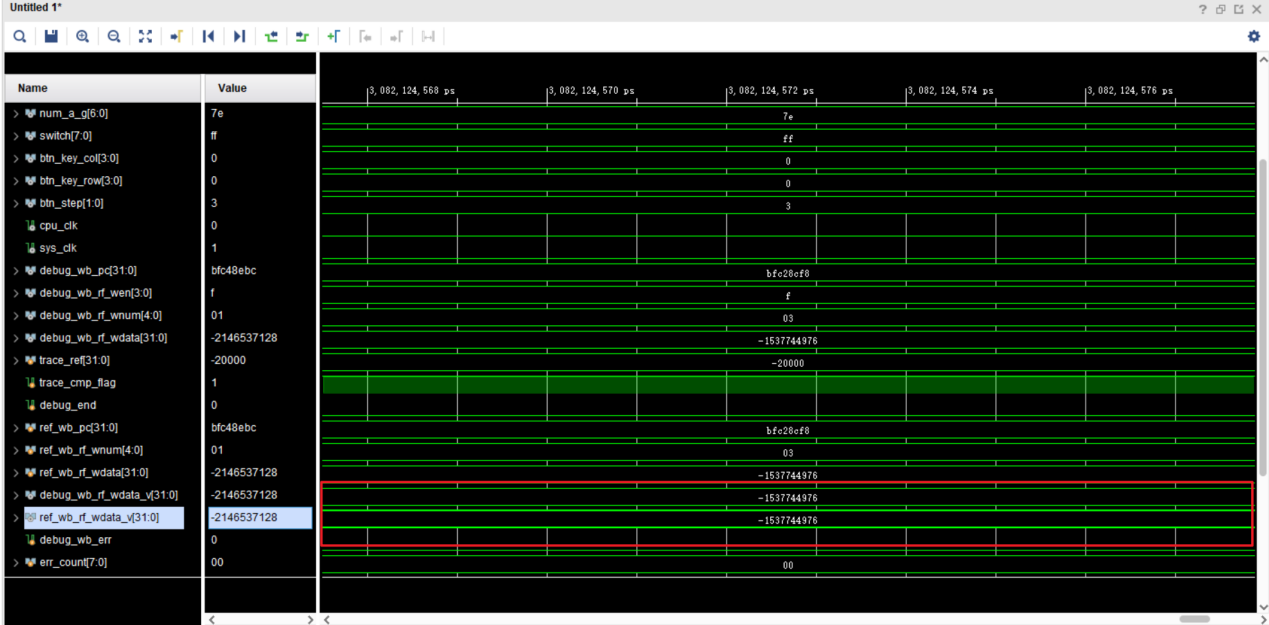
      ...

    endcase

  end

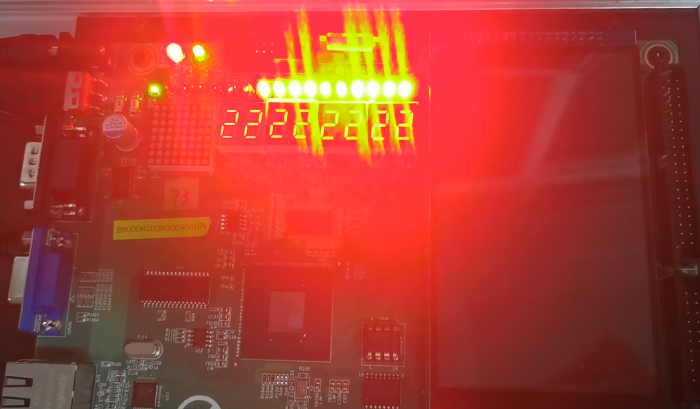
在写回级根据之前生成的访存信号将从内存读入的数据扩充为32位以写入对应寄存器。至此整条指令执行完成。

**2.3 Trace比对的方法进行仿真测试**（需列出指令的测试波形以及程序段的测试结果并分别说明）



LW测试点中对应的写回至寄存器的值和trace比对中的值相等，说明结果是正确的。

**2.4 FPGA上板验证过程**



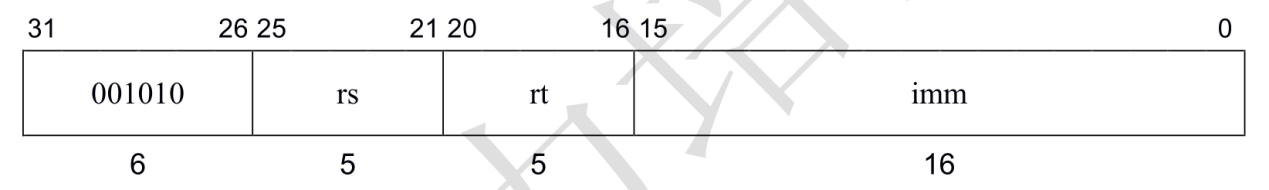
上板后会将参考值输出在7段数码管上，当有错误测试点时会停止。结果表明我们的上板测试是正确的。

**Part 2 个人任务--扩展实验（在TinyMIPS基础上扩展指令，建议运算类、跳转类、访存类均有）**

**总述:共实现了5条指令扩展，分别是:SLTI, SLTIU, LH, LHU, SH其中指令SLTI的设计实现过程如下：**

**1 指令格式**

SLTI是一条I型指令，用于比较，指令格式如下：



汇编格式：SLTI rt, rs, imm

功能描述：将寄存器 rs 的值与有符号扩展至 32 位的立即数 imm 进行有符号数比较，如果寄存器 rs 中的值小，则寄存器 rt置 1；否则寄存器 rt置 0。

**2分析指令功能及执行过程，画出数据通路图**

IF级：取出该指令。

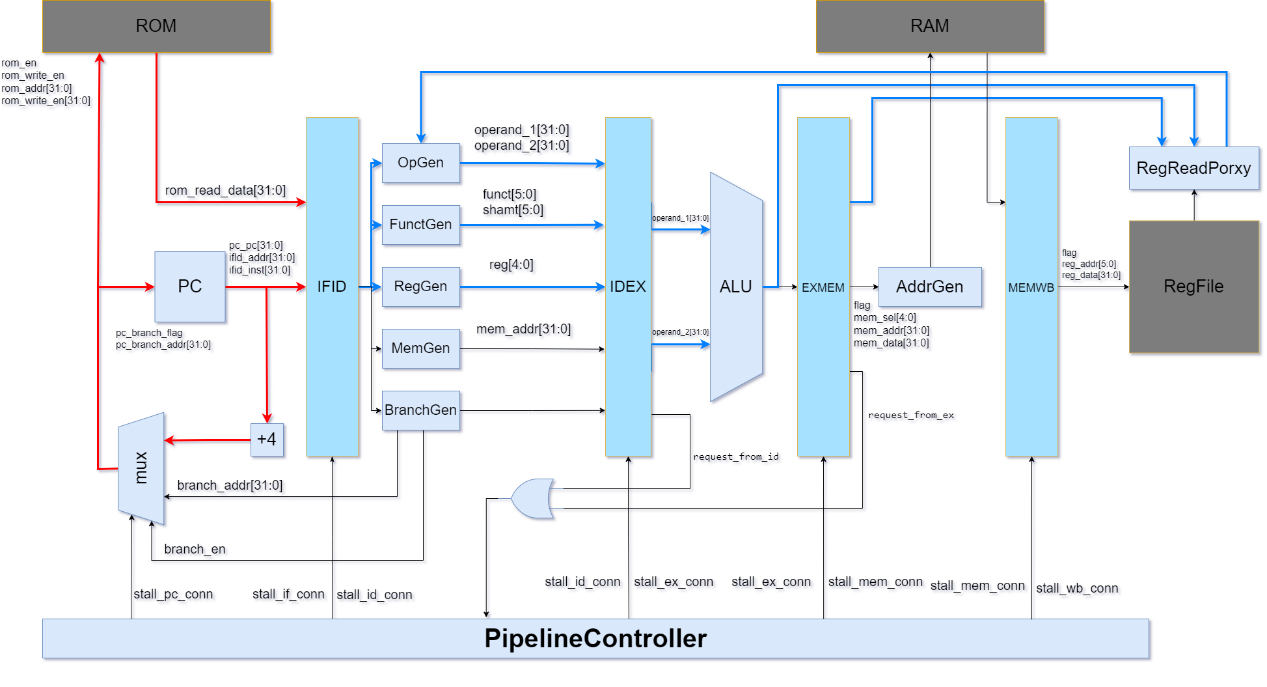
ID级：译码产生相关信号。涉及到rs号寄存器的读操作，rt寄存器的读操作；FUNCT域设为SLT对应的FUNCT域，指示ALU实现有符号比较运算；操作数生成上使用rs寄存器操作数和imm立即数的符号扩展。

EX级：完成比较运算并生成结果输出。

MEM级：该指令不涉及访问内存，因此本级只完成信号和结果的传递。

WB级：将EX级生成的运算结果写回至寄存器堆。

数据通路如下：



如图，红色为指令流，蓝色为数据流部分数据通路。

**3 代码实现**

ID级修改部分：

(RegGen.v)

always @(\*) begin

case (op)

...

    `OP\_SLTI: begin

        reg\_read\_en\_1 <= 1; // 寄存器读使能信号

        reg\_read\_en\_2 <= 0;

        reg\_addr\_1 <= rs; // 读入 rs 号寄存器

        reg\_addr\_2 <= 0;

      end

      ...

    endcase

  end

always @(\*) begin

...

    case (op)

`OP\_SLTI: begin

        reg\_write\_en <= 1; // 寄存器写使能信号

        reg\_write\_addr <= rt; // 写入 rt 号寄存器

      end

...

    endcase

  end

(OperandGen.v)

always @(\*) begin

    case (op)

    ...

    `OP\_SLTI: begin

        operand\_1 <= reg\_data\_1; // 第一个操作数是rs寄存器操作数

      end

     ...

    endcase

  end

  // generate operand\_2

  always @(\*) begin

case (op)

...

    `OP\_SLTI: begin

        operand\_2 <= sign\_ext\_imm;//第二个操作数是立即数的符号扩展

      end

    ...

    endcase

  end

(FnctGen.v)

always @(\*) begin

case (op)

...

`OP\_SLTI: funct <= `FUNCT\_SLT; // 设置有符号比较运算功能域

...

    endcase

  end

EX级修改部分:

wire[`DATA\_BUS] operand\_2\_mux =

      (funct == `FUNCT\_SUB || funct == `FUNCT\_SUBU || funct == `FUNCT\_SLT)

        ? (~operand\_2) + 1 : operand\_2;

// 比较运算时第二个操作数取立即数的相反数，因为是使用减法后判断结果正负来比较大小

  // sum of operand\_1 & operand\_2

  wire[`DATA\_BUS] result\_sum = operand\_1 + operand\_2\_mux;

  // flag of operand\_1 < operand\_2

  wire operand\_1\_lt\_operand\_2 = funct == `FUNCT\_SLT ?

        // op1 is negative & op2 is positive

        ((operand\_1[31] && !operand\_2[31]) ||

          // op1 & op2 is positive, op1 - op2 is negative

          (!operand\_1[31] && !operand\_2[31] && result\_sum[31]) ||

          // op1 & op2 is negative, op1 - op2 is negative

          (operand\_1[31] && operand\_2[31] && result\_sum[31]))

      : (operand\_1 < operand\_2);

// 通过符号位和运算结果实现比较运算，得到对应结果

  // calculate result

  always @(\*) begin

    case (funct)

      // comparison

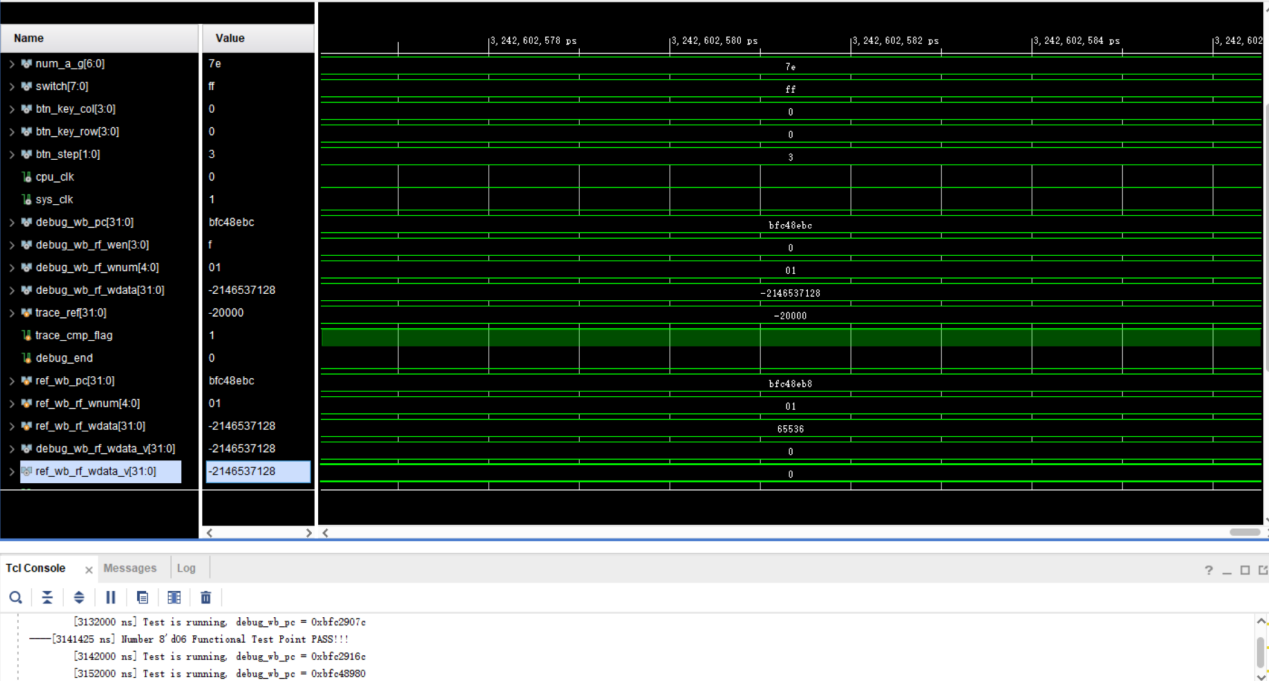
  `FUNCT\_SLT, `FUNCT\_SLTU: result <= {31'b0, operand\_1\_lt\_operand\_2};

      // 比较运算的结果选择

    endcase

  end

**4 Trace比对方法进行指令功能测试**（阐述采用Trace比对进行测试的过程，该测试部分可将所有扩展指令放在一起做。）



通过了所有的扩展指令。

**5自行编写Testbench进行指令功能仿真测试（**需有测试激励，仿真波形截图及分析，该测试部分可将所有扩展指令放在一起做。**）**

**使用的指令的汇编代码如下，测试了个人扩展的所有5条指令。**

nop

addiu $8, $0, 258

addiu $9, $0, 1

addiu $13, $0, 2

sh $8, 2($13)

lh $9, 2($13)

slti $7,$9, 255

sltiu $6,$9, 259

lhu $7, 2($13)

测试激励如下：

`timescale 1ns / 1ps

module core\_tb\_top;

    reg clk;

    reg rst;

    reg stall;

    wire                  rom\_en;

    wire  [`MEM\_SEL\_BUS]  rom\_write\_en;

    wire  [`ADDR\_BUS]     rom\_addr;

    wire   [`DATA\_BUS]     rom\_read\_data;

    wire  [`DATA\_BUS]     rom\_write\_data;

    // RAM control

    wire                  ram\_en;

    wire  [`MEM\_SEL\_BUS]  ram\_write\_en;

    wire  [`ADDR\_BUS]     ram\_addr;

    wire  [`DATA\_BUS]     ram\_read\_data;

    wire  [`DATA\_BUS]     ram\_write\_data;

    // debug signals

    wire                  debug\_reg\_write\_en;

    wire  [`REG\_ADDR\_BUS] debug\_reg\_write\_addr;

    wire  [`DATA\_BUS]     debug\_reg\_write\_data;

    wire  [`ADDR\_BUS]     debug\_pc\_addr;

    initial begin

        stall <= 0;

        clk <= 0;

        rst <= 1;

        #6 rst <= 0;

    end

    always #5 clk<=~clk; // T = 10

    Core core(clk,rst,stall,

        rom\_en,

        rom\_write\_en,

        rom\_addr,

        rom\_read\_data,

        rom\_write\_data,

        ram\_en,

        ram\_write\_en,

        ram\_addr,

        ram\_read\_data,

        ram\_write\_data,

        debug\_reg\_write\_en,

        debug\_reg\_write\_addr,

        debug\_reg\_write\_data,

        debug\_pc\_addr);

    RAM ram(

        clk,

        ram\_en,

        ram\_write\_en,

        ram\_addr,

        ram\_write\_data,

        ram\_read\_data

    );

    ROM rom(

        clk,

        rom\_en,

        rom\_write\_en,

        rom\_addr,

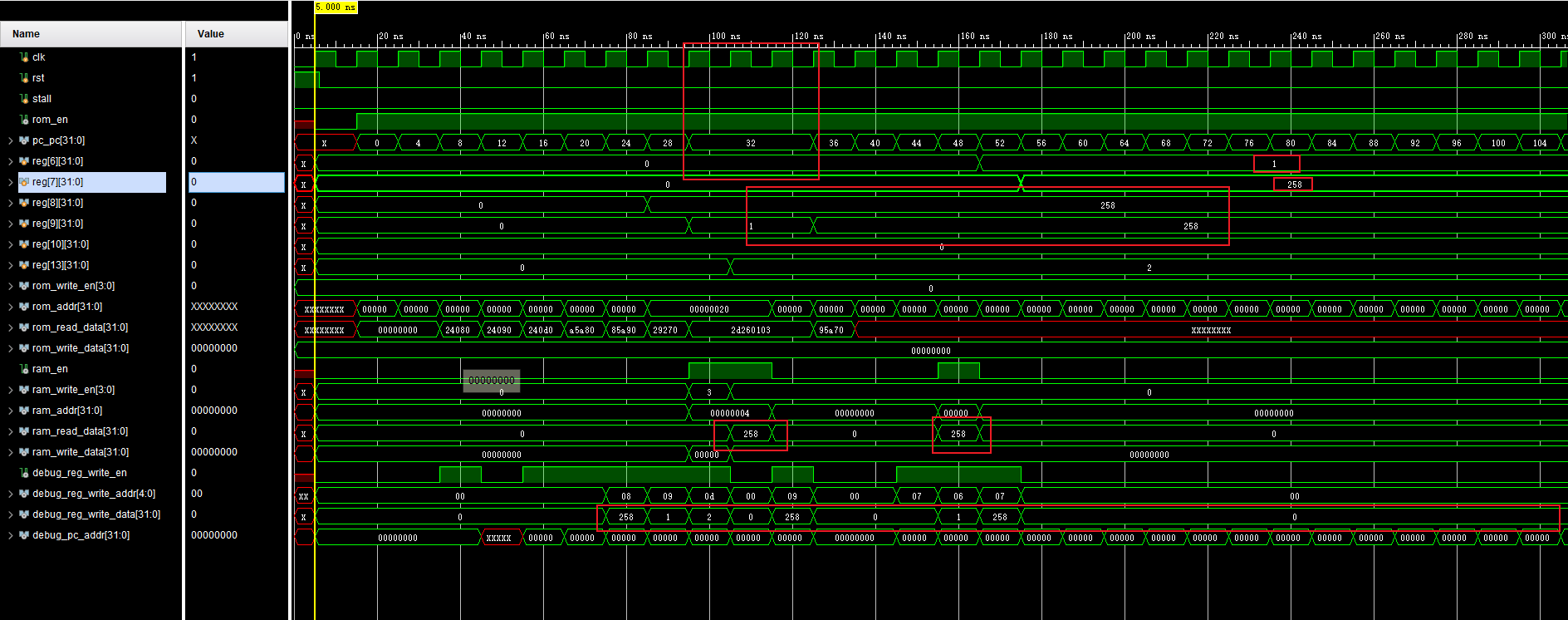
        rom\_write\_data,

        rom\_read\_data

    );

endmodule

对应的仿真波形图如下：



我们重点关注红框标注部分。首先要说明的是，为了方便测试，将起始PC的值设置为了4。

分析上述指令，首先执行了：

addiu $8, $0, 258

addiu $9, $0, 1

addiu $13, $0, 2

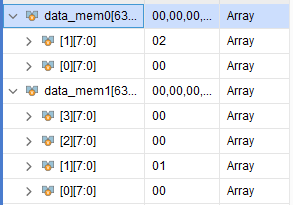
即将8，9，13号寄存器分别设置为258，1，2。在仿真波形图中可以看到当PC等于28时，寄存器的值才开始陆续被赋值成正确的值，这符合流水线工作的原理。

接下来执行下列指令：

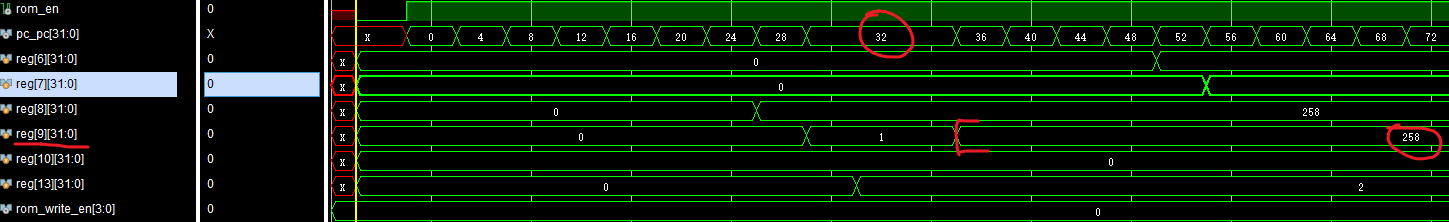
sh $8, 2($13)

lh $9, 2($13)

测试sh指令，即在内存地址($13)+2=4处写入8号寄存器低半字的值，也就是258的低半字，写成16进制就是0x0102。查看内存后可以发现成功在对应的内存地址处写入了该数值，如下图：



之后测试了lh指令，其将在($13)+2=4内存地址处读入半字并写入到9号寄存器内。这里我们从仿真波形中看到一个特殊现象，就是PC在32处停留了很长时间。产生这个的原因就是下一条指令要从寄存器中读入这次的数值，因此会从寄存器代理模块读入，但是该数值要在MEM级才会被读回，可是此时该指令还处于EX级，因此引发了流水线暂停以等待数据被真正读回。如下图：



当暂停了两个时钟周期后流水线才恢复正常工作，这时寄存器的值也变为了读入的数值即258。这说明sh和lh都正确执行了。

再往后测试slti和sltiu指令：

slti $7,$9, 255

sltiu $6,$9, 259

这里依次将9号寄存器中的数值与立即数255，259进行有符号/无符号比较，并将结果分别写回7号和6号寄存器。在前面的波形中可以看到两个寄存器的值分别被赋值为0和1，这符合预期结果。

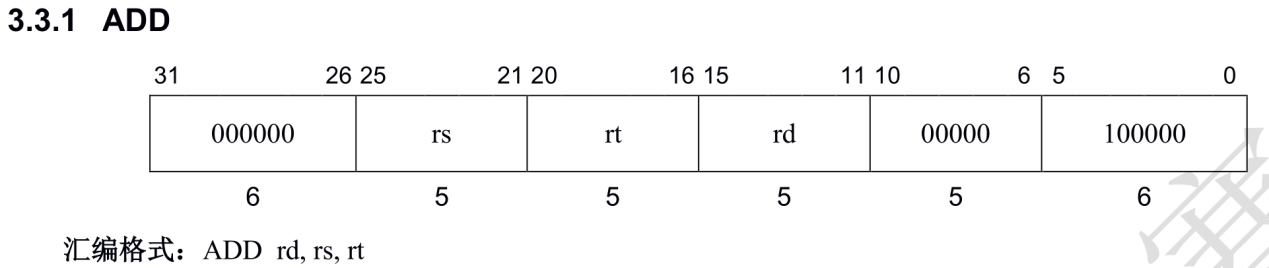
最后一条指令测试了lhu指令，它也能正确将内存对应处的值读入到7号寄存器。从波形中可以看到此指令正确读入了数据并将其写回至寄存器。

**6其他类型指令的分析**

上面的部分分析的是一条I型运算指令。接下来简单分析一下R型运算指令和J型指令的执行过程。

**R型运算指令：以ADD为例**

指令格式如下：



该指令功能是将rs寄存器与rt寄存器中的值进行无符号相加，并将结果送入rd寄存器中。

首先其不会产生分支，因此PC、IF级无修改。

在ID级，由于是R型指令，FUNCT域无需在此生成而是在指令中已指定，我们只需要生成rs和rt这两个寄存器操作数，其会从RegReadProxy模块中得到并沿着流水线向下传递至EX级。

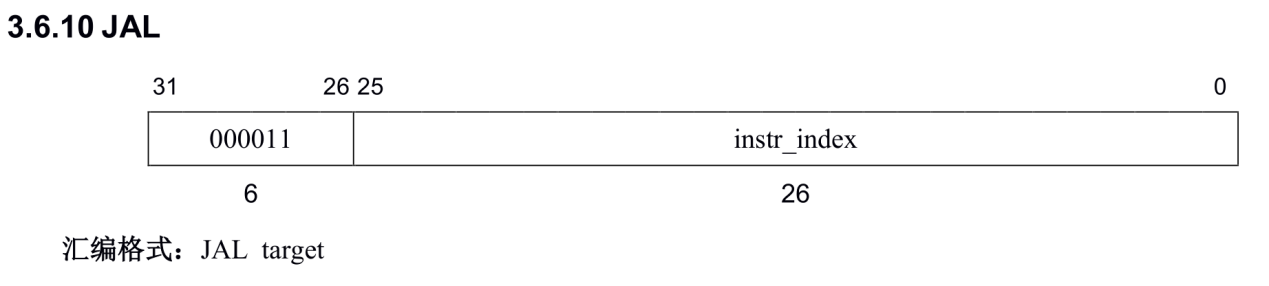
在EX级，由于FUNCT域设置为有符号数的加法，因此ALU执行有符号数加法，并将结果沿流水线传递。

在MEM级，由于不涉及访存操作，因此没有影响。

在WB级，指定了要回写的数是ALU的输出结果，将计算结果写回至rd寄存器中，至此指令执行完毕。

**J型指令：以JAL为例**

其指令格式如下：



无条件跳转，跳转目标由该分支指令对应的延迟槽指令的 PC 的最高 4 位与立即数 instr\_index 左移2 位后的值拼接得到。同时将该分支对应延迟槽指令之后的指令的 PC 值保存至第 31 号通用寄存器中。

那么在IF、PC级，由于约定，在J型指令后会插入延迟槽指令，因此不用特别修改。而是当JAL指令执行到ID级给出跳转信号和对应地址后，将PC修改为对应的数值。

ID级，需要计算出跳转到的地址并给出跳转信号。同时给出FUNCT域为无符号加法，使ALU能计算出要保存到寄存器中的返回。

EX级，计算出返回地址。

MEM级，无需访存，无修改，只进行数据传递。

WB级，将返回地址保存到31号通用寄存器中。

至此指令结束。

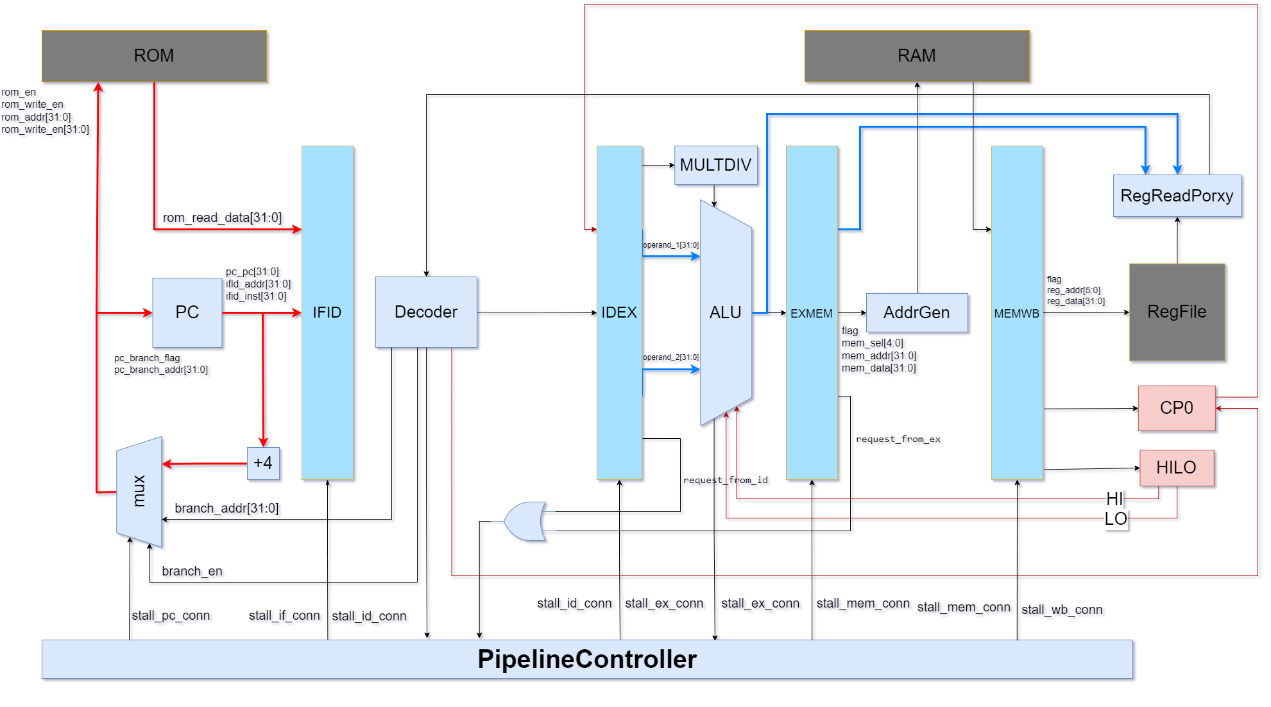
**Part 3 创新实验（演示系统介绍）实验分组编号：5组**

**系统概述：实现了什么功能，扩展了哪些内容，如何测试，有何发现。**

我们实现了多周期的华莱士树乘法器，迭代除法器，HOLO寄存器及相关数据转移指令，一级直接映射cache；扩展了CP0协处理器部分内容以及所有的异常指令的处理。

测试方面，我们对于乘除法器和Cache部分首先编写了单独的testbench，以此模拟这些部件工作的信号情况，根据相关输出信号判断其是否正常工作，有无错误。在单独测试后，再将这些部件集成入CPU中，通过trace比对的方式判断实现的功能是否正确。

**1 系统总体架构图**

****

**2 乘法器部分**

**设计过程：**

乘法器的设计上，我们计划实现的是一个在7个时钟上升沿后得到计算结果的华莱士树乘法器。其设计的核心思想是将乘法转换为多个数的加法。

首先我们需要得到这些相加的数，也就是部分积。该部分通过booth编码解决，基于以下公式，我们可以将原本的32个部分积转换为仅17个部分积。

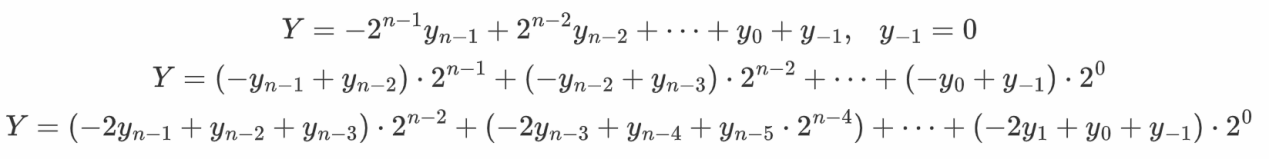


图3.1 Booth编码原理

之后，我们的加法用到的并不是普通的加法器，而是进位保留加法器，这也是华莱士树乘法器的核心部件。他可以将三个数相加转换为两个数相加，换言之其是一个有三个输入，两个输出的部件，可以将相加的数的个数规模减小。据此我们可以通过一级级的进位保留加法器将要相加的数的个数不断减少，直至只有两个加数，最后再通过一个普通的64位全加器输出最后的结果。

该部分我们采用了类似流水线的结构以提高效率，具体结构图如下：

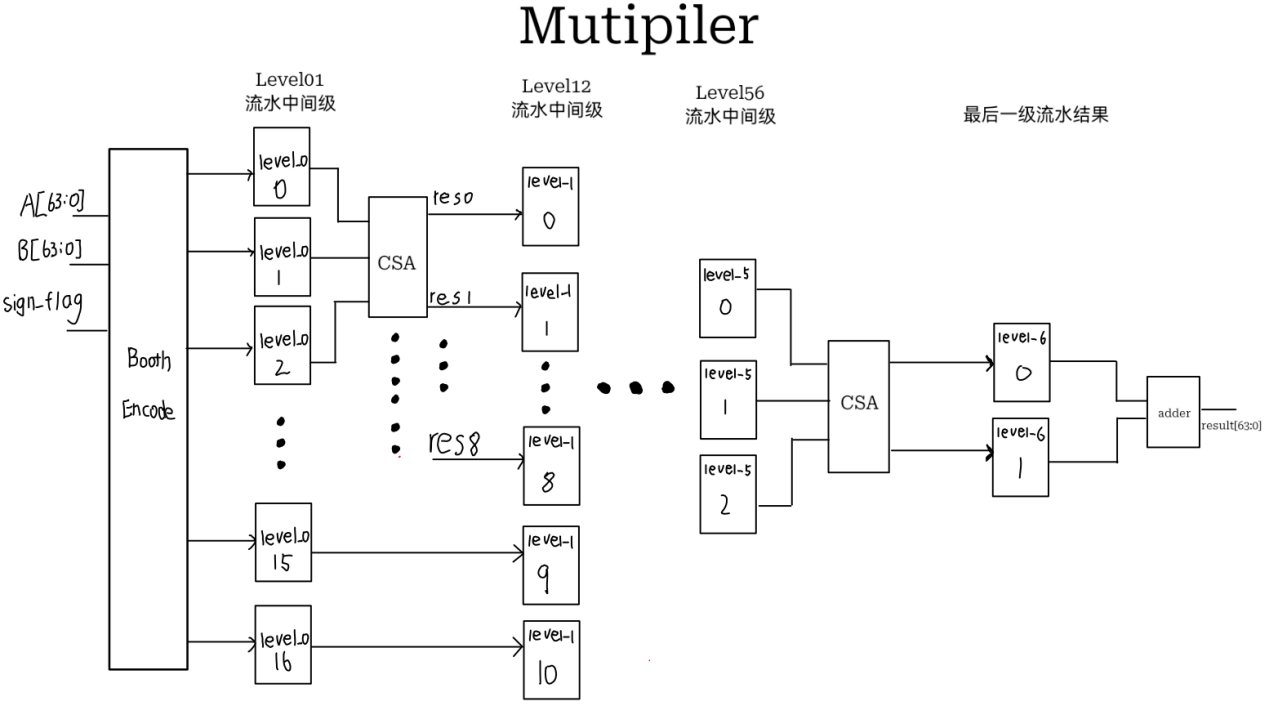


图3.2 流水化华莱士树结构简图

之后考虑将乘法器集成至CPU中，我们的方法是约定好乘法器将要进行的时钟周期数。开始计算时首先给出乘法器的使能信号，这样在下一个时钟上升沿乘法器检测到使能信号有效便会开始进行计算。同时在乘法器外计时，当约定好的时钟周期到时才从乘法器的输出端口取得结果。在乘法器计算期间，EX级前的流水线都是暂停的。

**实验现象及分析：**

我们采用了单独编写testbench的方法首先对乘法器进行单独测试，波形图如下：

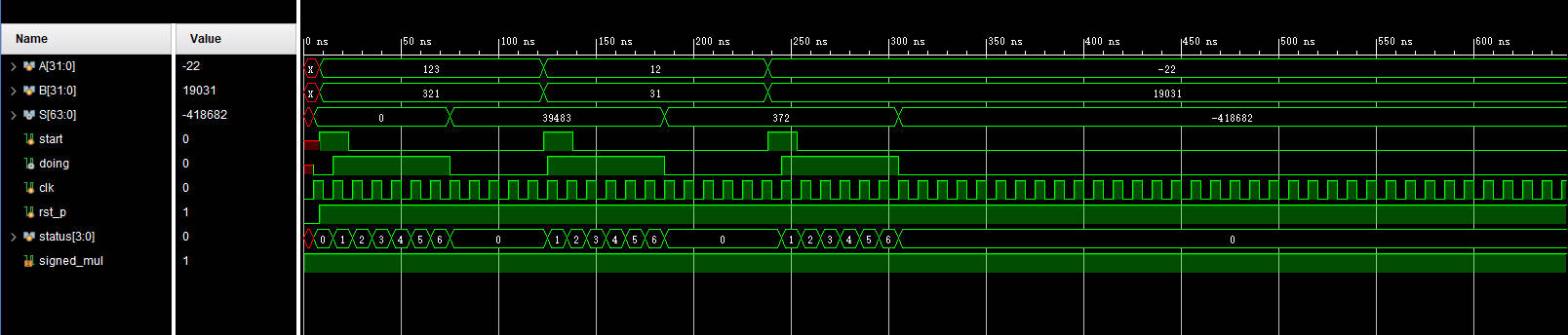
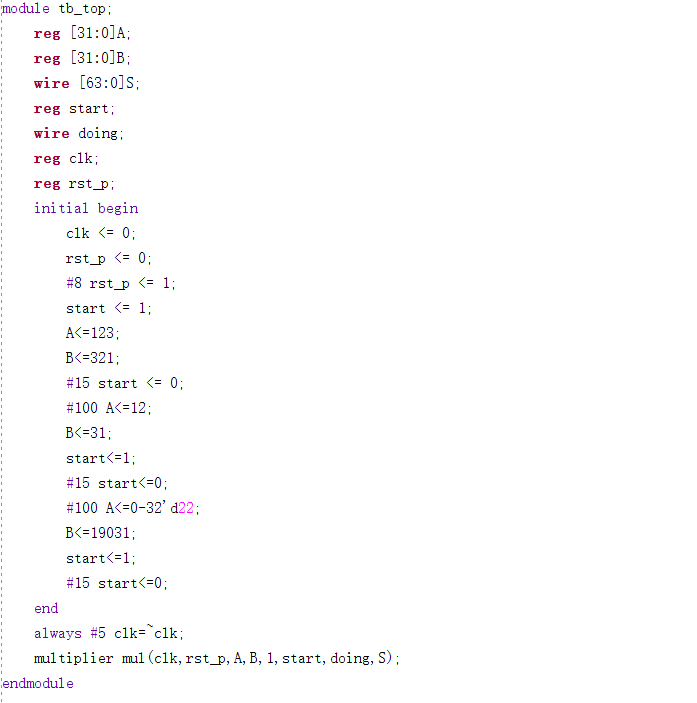


图3.3 乘法器测试波形图

这个过程采用的testbench代码如下：



由波形图可以看出，乘法器可以在7个时钟上升沿后输出正确的结果，并且可以正常的连续工作，说明我们的乘法器的功能是正确的。

**3 除法器部分**

1. **设计原理**

我们小组设计的迭代除法器原理方法为试商法，其核心思想是将除法转为减法进行处理。换而言之，便是将纸笔运算的流程用程序来实现。试商法的原理如下：

首先为方便阐述，谁当下述数据的最低位的索引为1。假设除数为a=1101，被除数为b = 0100，商为c，辅助参数k初始值为4。首先取a[k]作为被减数，取b作为减数，进行减法运算，得到减法结果d。如果d>0，则赋值商c[k]=1，并且更新被减数为{减法结果, a[k-1]}；如果d<0，则赋值商c[k] = 0，并更新被减数为{a[k],a[k-1]}；

然后赋值k = k – 1，利用上一轮的更新值进行下一轮的减法运算。

当k=0时，循环结束，至此也获得了四位的商与余数（恰为被减数的值）。

1. **程序设计**

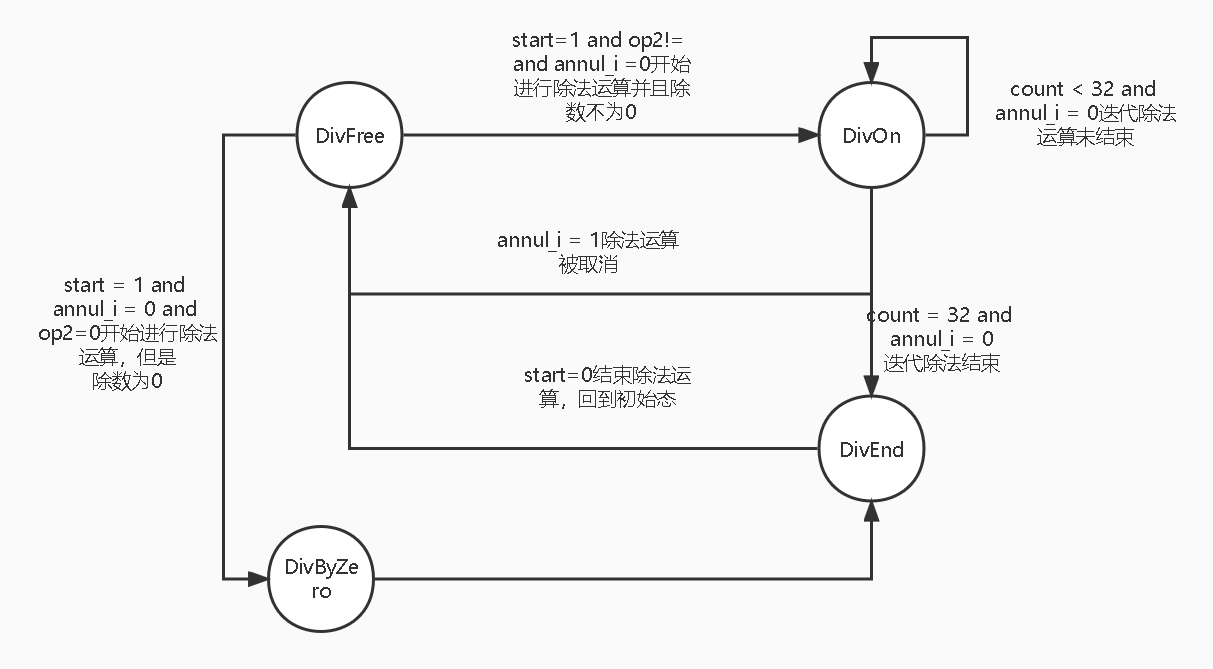
**首先是模块的输入输出说明：**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **序号** | **接口名** | **宽度** | **输入/输出** | **作用** |
| **1** | **rst** | **1** | **输入** | **复位** |
| **2** | **clk** | **1** | **输入** | **时钟** |
| **3** | **signed\_div** | **1** | **输入** | **为1代表有符号** |
| **4** | **op1** | **32** | **输入** | **被除数** |
| **5** | **op2** | **32** | **输入** | **除数** |
| **6** | **start** | **1** | **输入** | **是否开始运算** |
| **7** | **annul** | **1** | **输入** | **是否取消运算** |
| **8** | **result\_o** | **64** | **输出** | **除法结果，高32位为余数，低32为商** |
| **9** | **ready\_o** | **1** | **输出** | **除法是否结束** |

从输入输出端口的角度来看，整个运行流程为：首先是rst有效，使触发器状态机进入初始状态，然后start为高电平有效，除法器接收op1,op2，并根据signed\_div判断是否进行有符号运算，在33个周期的运算内，如果annul=1，说明异常除法，需要清空流水线，状态机回到初始状态，否则则输出运算结果result\_o，并向外给出运算完成的信号ready\_o。

然后是状态机说明：

为实现上述原理，需要设计一个四状态的状态机，具体如下图所示：



状态机说明：首先是DivFree状态，这是rst有效后进入的初始状态，它的作用是初始化各个变量（第一轮的被减数，减数，计数变量等），并根据signed\_div判断是否对负数的操作数进行变补操作。以及根据start,annual,以及除数是否为0判断下一个时钟上升沿进入哪个状态。

然后时DivByZero状态，它代表除数为0，在这个状态中，会将next\_state置为DIvEnd。

下一个是DivOn状态，这是除法运算的核心状态，在此状态中，将根据cnt的值，进行32个周期的运算，运算内容便是实验原理中提到的试商法不断迭代。这里不再赘述。最终的结果会存在dividend中，除此之外，在第33个上升沿时，此状态会根据signed\_div以及除数，被除数，商，余数的符号对结果的符号进行调整，规则便是，op1,op2异号，则商为负数，余数的符号与op1保持一致。

最后一个状态时DivEnd状态，在这个状态中，会将结果输出，并给出运算结束的信号，当再接收start=0后，其会将输出信号全部置为0并且进入DivFree状态。

1. **最后说明与其他模块的交互**

与其他模块的交互为三部分：运算结果写入HILO寄存器；流水线暂停；异常处理，取消除法运算。

对于第一部分而言，其与除法器关系其实不大，这里不再赘述。

第二部分，因为除法运算需要33个周期，所以在进行除法运算的时候需要将流水线暂停。这个操作在EX级实现，首先EX会判断运算类型是否为DIVU,DIV，如果是，stall\_all置为1暂停流水线，并且start置为1，除法器开始工作，当除法器完成运算后，EX级别，根据ready\_o将stall\_all置为0，开启流水线，与此同时start变为0，除法器回到DIvFree状态。

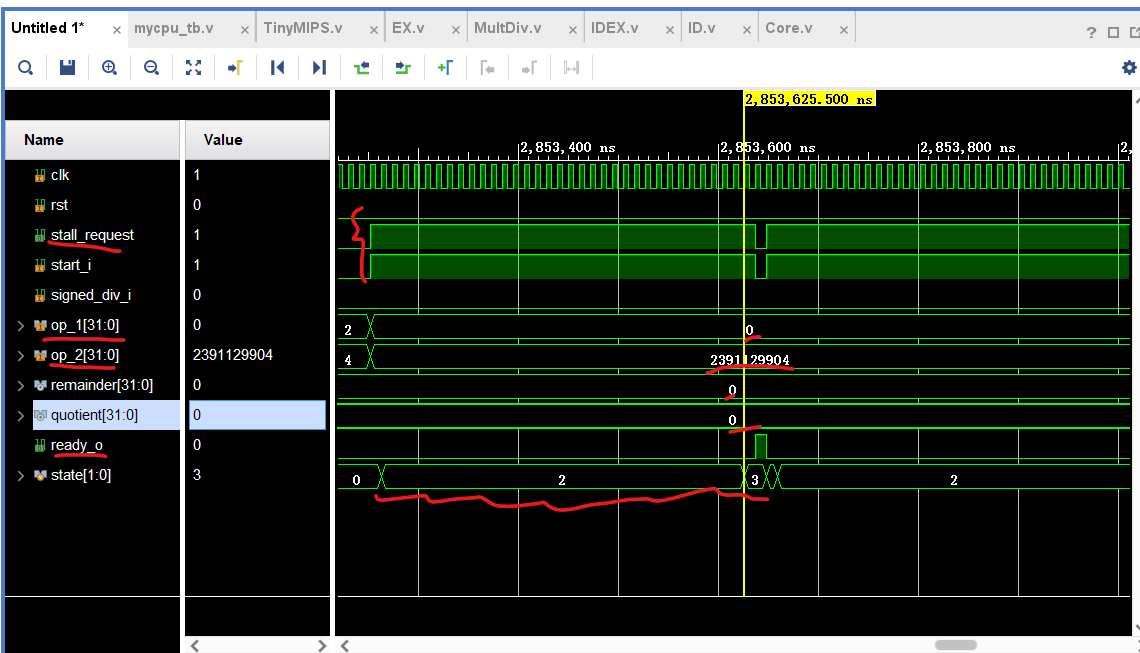
第三部分主要涉及annual信号，当某处发生异常时，需要将当前流水线执行的指令全部清空，所以此时annual的值会变为1，除法器回到DivFree状态，并且所有变量值清空。

1. **实验现象及分析**

直接使用gold\_trace进行仿真并查看波形图进行分析：

**波形图1：**

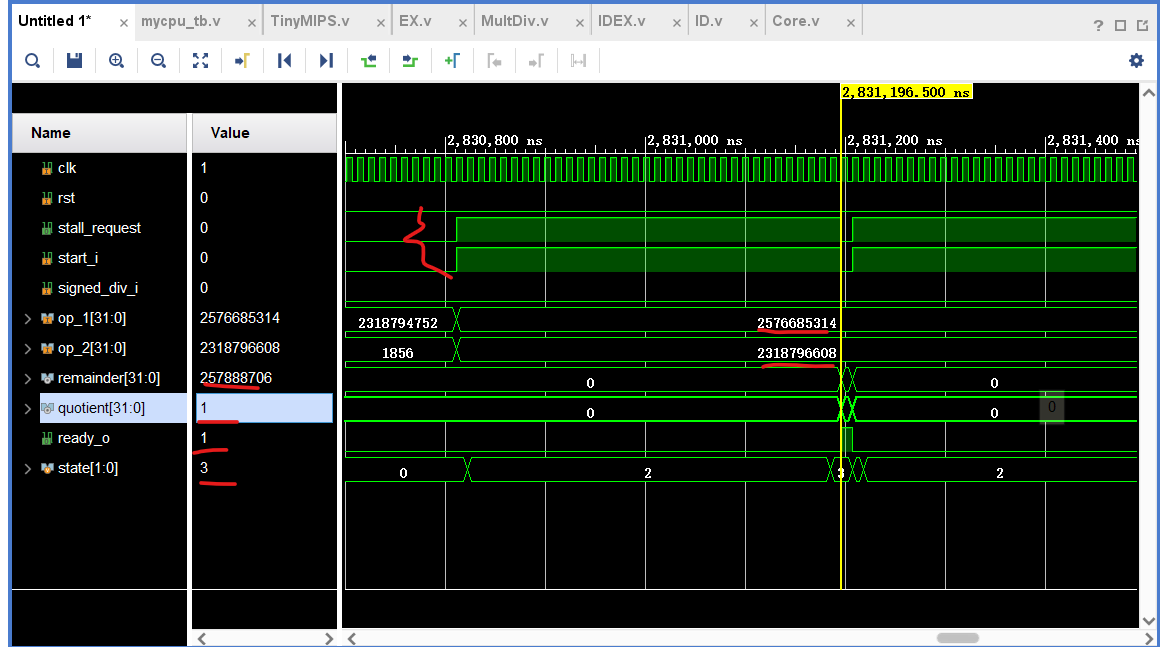
被除数为0，除数为2391129904，无符号运算。



通过上述波形图，我们可以看到除法器的余数，商均正确，并且ready\_o也正确给出，状态state的变化也正确。流水线的暂停也暂停与开启也顺利实现。

**波形图2：**

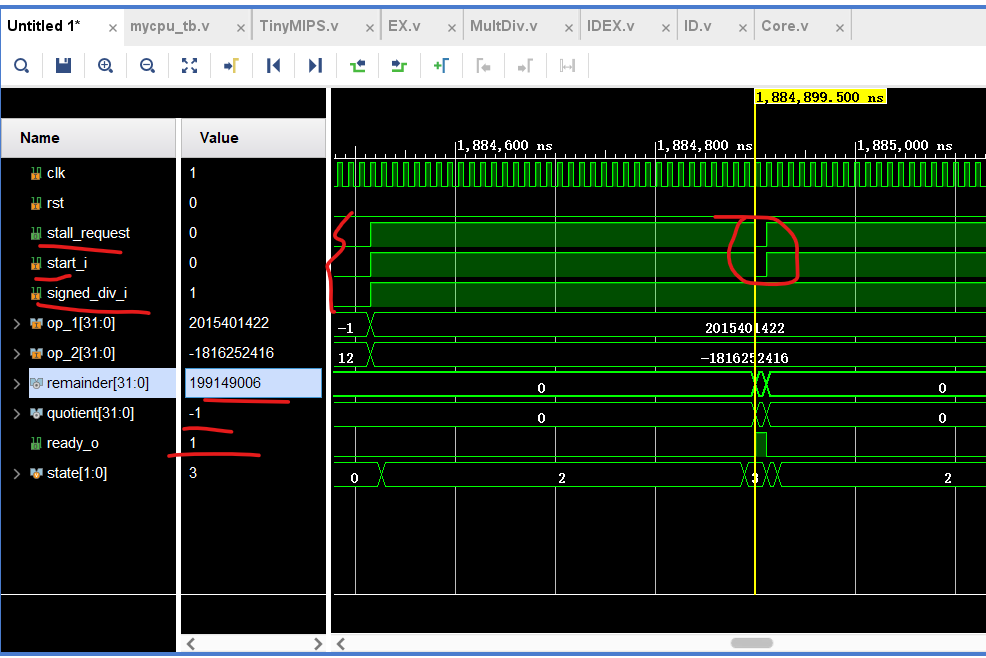
被除数为2576685314，除数为2318796608，无符号运算。



通过上述波形图，我们可以看到除法器的余数为257888706，商为1，均正确，并且ready\_o也正确给出，状态state的变化也正确。流水线的暂停也暂停与开启也顺利实现。

**波形图3：**

被除数为2015401422，除数为-1816252416，有符号运算。



通过上述波形图，我们可以看到除法器的余数为199149006，商为-1，均正确，并且ready\_o也正确给出，状态state的变化也正确。流水线的暂停也暂停与开启也顺利实现。

**4 HILO寄存器部分**

设计过程：

首先是创建一个hilo寄存器模块，用于将hi和lo寄存器的值输出以及将乘除法器的值进行保存。

module HILO (

input clk,

input rst,

input write\_en,

input [`DATA\_BUS] hi\_i,

input [`DATA\_BUS] lo\_i,

output [`DATA\_BUS] hi\_o,

output [`DATA\_BUS] lo\_o

);

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

endmodule // HILO

然后在顶层模块创建hilo相关的信号和变量进行数据和信号传递，主要在EX级。定义：

wire [`DATA\_BUS] hilo\_rp\_hi, hilo\_rp\_lo, ex\_hi, ex\_lo, exmem\_hi, exmem\_lo;

wire ex\_hilo\_write\_en, exmem\_hilo\_write\_en;

以及在MEM阶段定义：

wire [`DATA\_BUS] mem\_hi, mem\_lo, memwb\_hi, memwb\_lo;

wire mem\_hilo\_write\_en, memwb\_hilo\_write\_en;

还需要创建数据前递的模块hiloreadproxy：

module HILOReadProxy (

input [`DATA\_BUS] hi\_i,

input [`DATA\_BUS] lo\_i,

input mem\_hilo\_write\_en,

input [`DATA\_BUS] mem\_hi\_i,

input [`DATA\_BUS] mem\_lo\_i,

input wb\_hilo\_write\_en,

input [`DATA\_BUS] wb\_hi\_i,

input [`DATA\_BUS] wb\_lo\_i,

output [`DATA\_BUS] hi\_o,

output [`DATA\_BUS] lo\_o

);

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;

endmodule // HILOReadProxy

最后再顶层模块创建实例，并在EX级传送回去：

wire [`DATA\_BUS] hilo\_hi, hilo\_lo;

HILO u\_HILO (

.clk ( clk ),

.rst ( rst ),

.write\_en ( wb\_hilo\_write\_en ),

.hi\_i ( wb\_hi ),

.lo\_i ( wb\_lo ),

.hi\_o ( hilo\_hi ),

.lo\_o ( hilo\_lo )

);

HILOReadProxy u\_HILOReadProxy (

.hi\_i ( hilo\_hi ),

.lo\_i ( hilo\_lo ),

.mem\_hilo\_write\_en ( mem\_hilo\_write\_en ),

.mem\_hi\_i ( mem\_hi ),

.mem\_lo\_i ( mem\_lo ),

.wb\_hilo\_write\_en ( wb\_hilo\_write\_en ),

.wb\_hi\_i ( wb\_hi ),

.wb\_lo\_i ( wb\_lo ),

.hi\_o ( hilo\_rp\_hi ),

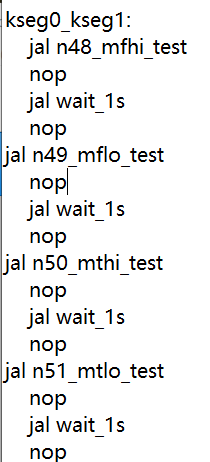
.lo\_o ( hilo\_rp\_lo )

);

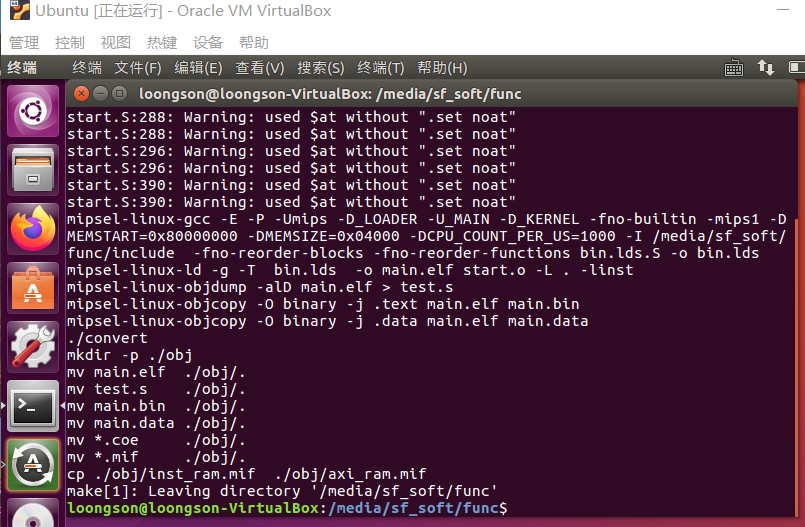
在EX级输入的hilo即为数据前递模块的输出hilo\_rp\_hi和hilo\_rp\_lo。

实验现象及分析：

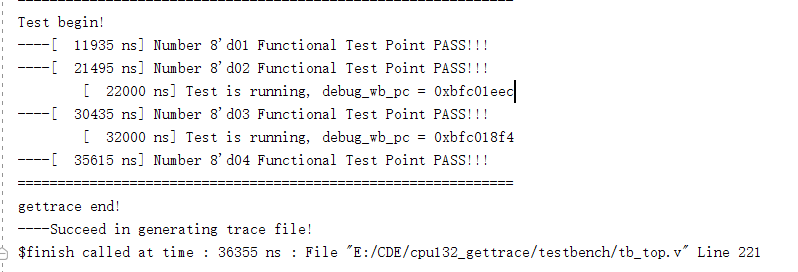
通过了MTHI,MFHI,MTLO,MFLO测试点,如下是测试start.s部分：



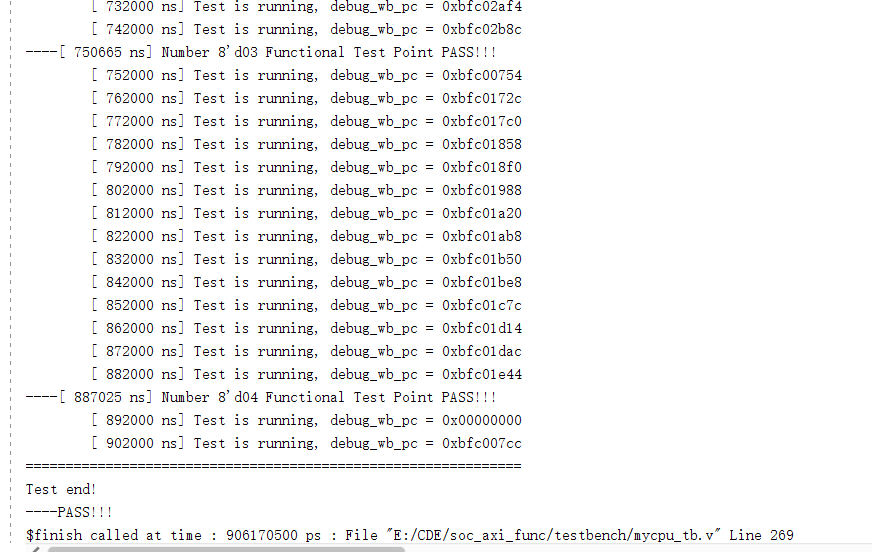
下面是测试过程截图：



这里为生成gold\_trace:



最后为测试结果：



**6 CP0协处理器和异常扩展部分**

**设计过程：**

1. **协处理器部分：**

本部分首先介绍 MIPS32 架构中的协处理器，说明了协处理器的作用。由于 OpenMIPS 计划实现其中的一个协处理器——CP0, 其实现方式有点类似 HI、LO 寄存器的实现方式。接着说明协处理器访问指令 mfcO、mtcO 的格式、作用、用法。最后节给出了协处理器访问指令的实现思路，以及对系统结构的修改。通过修改 OpenMIPS, 实现了协处理器访问指令,最后编写测试程序,在 ModelSim中进行仿真验证。 协处理器一词通常用来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 架构提供了最多 4 个协处理器，分别是 CP0 CP3,作用如下图:

|  |  |
| --- | --- |
| **协处理器** | **作用** |
| cp0 | 系统控制 |
| cp1 | fpu |
| cp2 | 特定实现 |
| cp3 | fpu |

协处理器 CP0 用作系统控制，CPI、CP3 用作浮点处理单元，而 CP2 被保留用于特定实现。除 CP0 外的协处理器都是可选的，OpenMIPS 没有实现浮点运算，所以 CPI、CP3 不用实现，CP2 也没有作用，不用实现。而 CP0 是不可选的，需要实现，所以下面重点介绍协处理器 CPO .配置 CPU 工作状态：符合 M1PS32 架构的硬件通常是很灵活的，可以通过读/写一个或一些内部寄存器来改变 一些很根木的 CPU 特性（ 如:将字节次序从 MSB 变为 LSB, 或者从 LSB 变为 MSB）。高速缓存控制：符合 MIPS32 架构的 CPU 一般会集成缓存控制器，用来控制、读、写缓存。 异常控制：异常发生时的检测和处理都由CP0的一些控制寄存器来定义和控制。存储管理单元控制：对系统的存储区域进行合理的控制管理分配，主要对MMU、TLB的一些配置、管理、访问。

1. **协处理器 CP0 中的寄存器：**

OpenMIPS 的设计目标是一个轻量级的处理器，并不打算实现缓存、MMU、TLB、调试等复杂功能，所以相关的寄存器都可以不用实现。下面依次介绍这几个主要寄存器的格式、作用。

1. Count 寄存器

Count 寄存器是一个不停计数的 32 位寄存器，计数频率一般与 CPU 时钟频率相同，当计数达到 32 位无符号数的上限时，会从 0 升始重新计数。Count 寄存器可读、可写。

// 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};

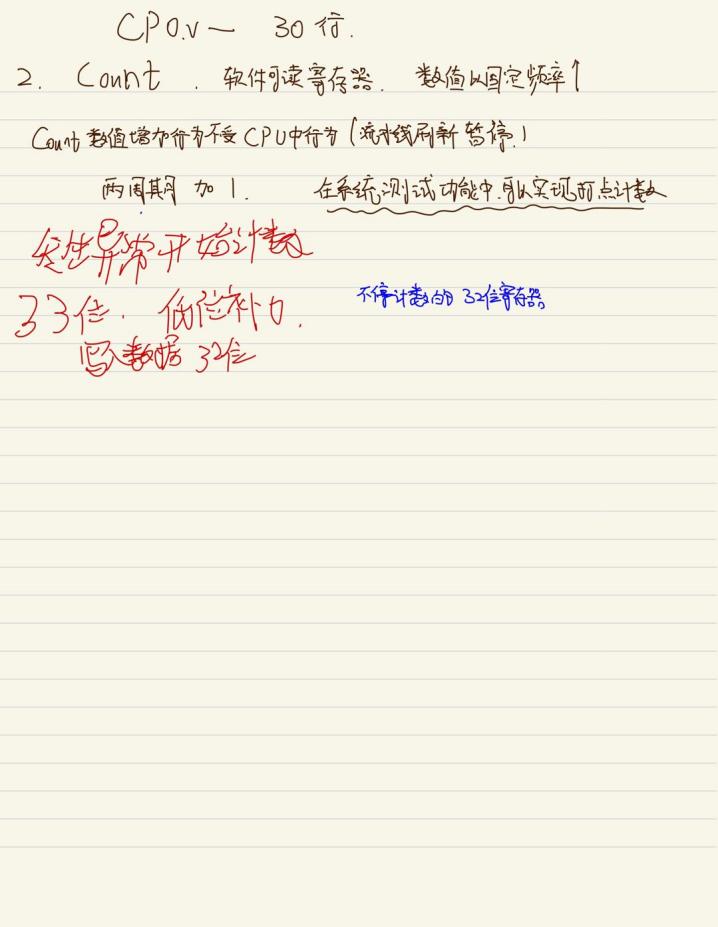
end

else begin

reg\_count <= reg\_count + 1;

end

end



2. Status寄存器

Status 寄存器也是个 32 位、可读、可写的寄存器, 用来控制处理器的操作模式、中断使能以及诊断状态。

// STATUS

always @(posedge clk) begin

if (rst) begin

reg\_status <= 32'h0040ff00;

end

else if (exc == `EXC\_ERET) begin

reg\_status[1] <= 0;

end

else if (exc != `EXC\_NULL) begin

reg\_status[1] <= 1;

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];

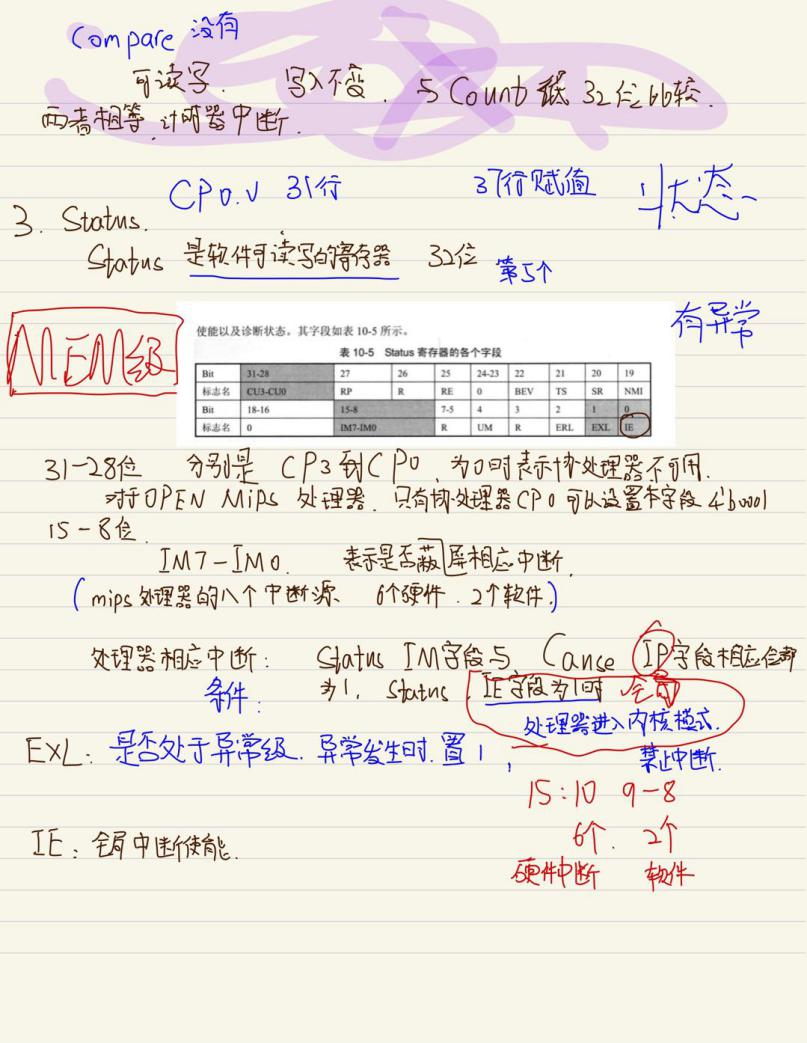
end

else begin

reg\_cause <= reg\_cause;

end

end



表中标识为 R 的字段是保留字段，下面逐一介绍其中的非保留字段。读者朋友如果  
没有时间，可以只理解其中使用灰色背景的字段，TinyMIPS 处理器也只实现了这些字段。

CU3-CU0表示协处理器是否可用 (Coprocessor Usability), 分别控制协处理器 CP3、CP2、CP1、CP0。为 0 时，表示相应的协处理器不可用；为 1 时，表示相应的协处理器可用。对于 OpenMIPS处理器而言，只有协处理器 CP0, 所以可以设置本字段为 4七0001。

• SR：表示是否是软重启 (Soft Reset), 为 1 表示重启异常是由软重启引起的。

• NMI：表示是否是不可屏蔽中断 (Non-Maskable Interrupt), 为 1 表示重启异常是由不可屏蔽中断引起的。

• IM7-IM0：表示是否屏蔽相应中断 (Interrupt Mask), 0 表示屏蔽，1 表示不屏蔽，MIPS 处理器可以有 8 个中断源，对应 IM 字段的 8 位，其中 6 个中断源是处理器外部硬件中断，另外 2 个是软件中断，中断是否能够被处理器响应是由 Status 寄存器与 Cause 寄存器共同决定的，如果 Status寄存器的 IM 字段 与 Cause 寄存器的 IP 字段的相应位都为 1 , 而且 Status 寄存器的 IE 字段也为 1 时，处理器才响应相应中断。

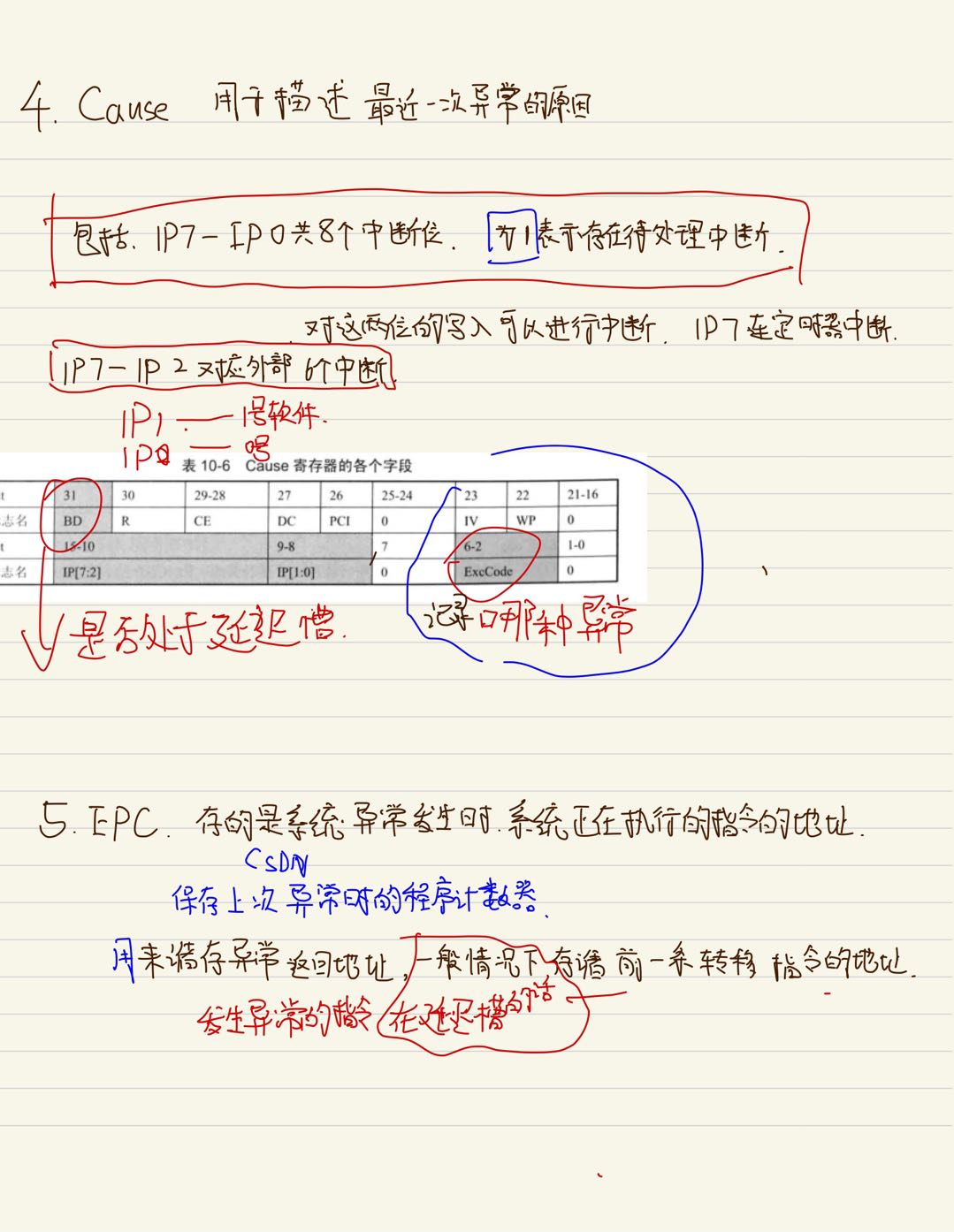
• ERL：表示是否处于错误级，当处理器接收到坏的数据时设置本字段为 1。有一些 MIPS 处理器在接收来自缓存或内存中的数据块时，能够检验数据中附带的奇偶校验位或纠错码，当发现数据错误且无法纠正时，处理器就设置 ERL 字段为 1, 并进入奇偶校验\ECC 错误的异常处理过程，这是一个特殊的异常处理过程 (有别于一般的异常处理过程)。读者只需知道，OpenMIPS 处理器没有对奇偶校验位或纠错码的检验过程，所以不用考虑 ERL 字段。

• EXL：表示是否处于异常级 (Exception Level), 当异常发生时，会设置本字段为 1, 表示处理器处于异常级，此时，处理器会进入内核模式下工作，并且禁止中断。

• IE：表示是否使能中断 (Interrupt Enable), 这是全局中断使能标志位。为 1 表示中断使能，为 0表示中断禁止。

1. Cause寄存器(标号13)

Cause 寄存器主要记录最近一次异常发生的原因，也控制软件中断请求。Cause 寄存器的各字段如表，除了IV 和 WP, 其余字段都是只读的。



// CAUSE

always @(posedge clk) begin

if (rst) begin

reg\_cause <= 32'h0;

end

else if (exc == `EXC\_INT) begin

reg\_cause[31] <= delayslot\_flag;

reg\_cause[6:2] <= `CP0\_EXCCODE\_INT;

end

else if (exc == `EXC\_RI) begin

reg\_cause[31] <= delayslot\_flag;

reg\_cause[6:2] <= `CP0\_EXCCODE\_RI;

end

else if (exc == `EXC\_BP) begin

reg\_cause[31] <= delayslot\_flag;

reg\_cause[6:2] <= `CP0\_EXCCODE\_BP;

end

else if (exc == `EXC\_SYS) begin

reg\_cause[31] <= delayslot\_flag;

reg\_cause[6:2] <= `CP0\_EXCCODE\_SYS;

end

else if (exc == `EXC\_OV) begin

reg\_cause[31] <= delayslot\_flag;

reg\_cause[6:2] <= `CP0\_EXCCODE\_OV;

end

else if (exc == `EXC\_ADEL) begin

reg\_cause[31] <= delayslot\_flag;

reg\_cause[6:2] <= `CP0\_EXCCODE\_ADEL;

end

else if (exc == `EXC\_ADES) begin

reg\_cause[31] <= delayslot\_flag;

reg\_cause[6:2] <= `CP0\_EXCCODE\_ADES;

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

BD：当发生异常的指令处于分支延迟槽 (Branch DelaySlot) 时, 该字段被置为 1

CE：当协处理器不可用异常发生时，将发生协处理器错误 (Coprocessor Error) 的协处理器序号存储到本字段。

PCI：这是在 MIPS32/64 架构中新增加的字段，当协处理器 CP0 的性能计数器溢出时

( Performance Count Interrupt ), 设置本字段为 1, 以产生中断。

WP：观测挂起 (Watch Pending) 字段，该字段与调试有关，为 1 表示有一个观测点被触发，处理器处于异常模式。

IP[7:2]：中断挂起 (Interrupt Pending) 字段，相应位用来指明外部硬件中断是否发生，1 表示发生，0 表示没有发生。本字段的 6 位与外部硬件中断的对应关系如下。

IP[7] ——5 号硬件中断IP[6] 4 号硬件中断

IP[5] ——3 号硬件中断IP[4] ——2 号硬件中断

IP[3] ——1 号硬件中断1P[2]——0 号硬件中断

IP[1:O]也是中断挂起字段，但是对应的是软件中断。

IP[1] ——1 号软件中断IP[0] ——0 号软件中断

1. EPC 寄存器

EPC 是异常程序计数器 (Exception Program Counter), 用来存储异常返回地址，一般情况下，存储发生异常的指令的地址，但是，如果发生异常的指令位于延迟槽中，那么 EPC 存储的是前一条转移指令的地址。该寄存器可读、可写。其字段如表所示。

// EPC

always @(posedge clk) begin

if (rst) begin

reg\_epc <= 32'h0;

end

else if (exc != `EXC\_NULL && exc != `EXC\_ERET) begin

reg\_epc <= exc\_epc;

end

else if (cp0\_write\_en && cp0\_write\_addr == `CP0\_REG\_EPC) begin

reg\_epc <= cp0\_write\_data;

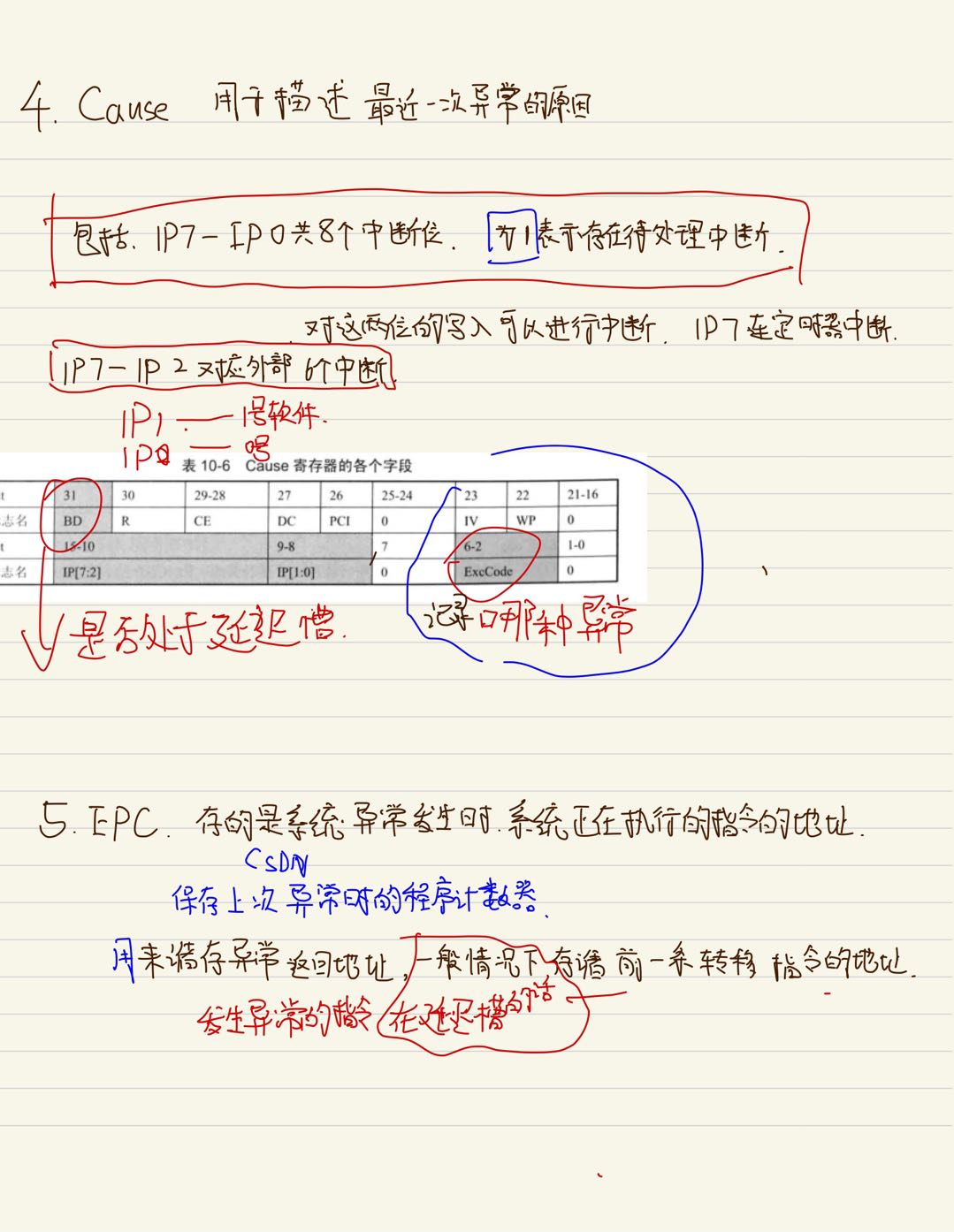
end

else begin

reg\_epc <= reg\_epc;

end

End



1. BADVADDAR寄存器

// BADVADDR

always @(posedge clk) begin

if (rst) begin

reg\_badvaddr <= 32'h0;

end

else if (exc == `EXC\_ADEL || exc == `EXC\_IF || exc == `EXC\_ADES) begin

reg\_badvaddr <= cp0\_badvaddr;

end

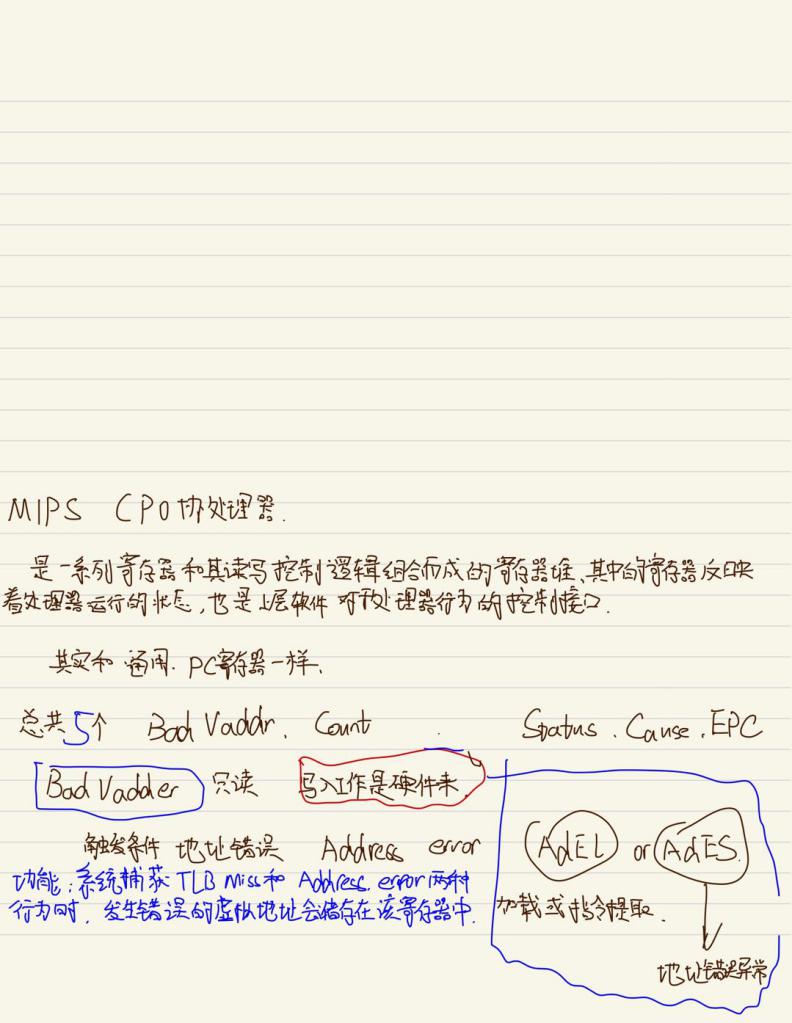
// only read

else begin

reg\_badvaddr <= reg\_badvaddr;

end

End



1. **协处理器 CP0 中的寄存器：**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **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 |

1. **协处理器访问指令**

要实现 CPO 的控制功能，需要对 CPO 中的有关寄存器进行设置，这涉及对 CPO 中寄存器的访问，需要使用协处理器访问指令。MIPS32 指令集架构中定义了 2 条协处理器访问指令:mtcO、mfcO, 前者实现修改 CPO 中的寄存器，后者实现读取 CPO 中的寄存器。指令格式如 ：

MFC0

31 26 25 21 20 16 15 11 10 3 2 0

010000 00000 Rt rd 00000000 sel

6 5 5 5 8 3

汇编格式： MFC0 rt, rd, sel

功能描述： 从协处理器 0 的寄存器取值

操作定义： GPR[rt] ← CP0[rd, sel]

MTC0

31 26 25 21 20 16 15 11 10 3 2 0

010000 00100 Rt rd 00000000 sel

6 5 5 5 8 3

汇编格式： MTC0 rt, rd, sel

功能描述： 向协处理器 0 的寄存器存值

操作定义： CP0[rd, sel] ← GPR[rt]

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;

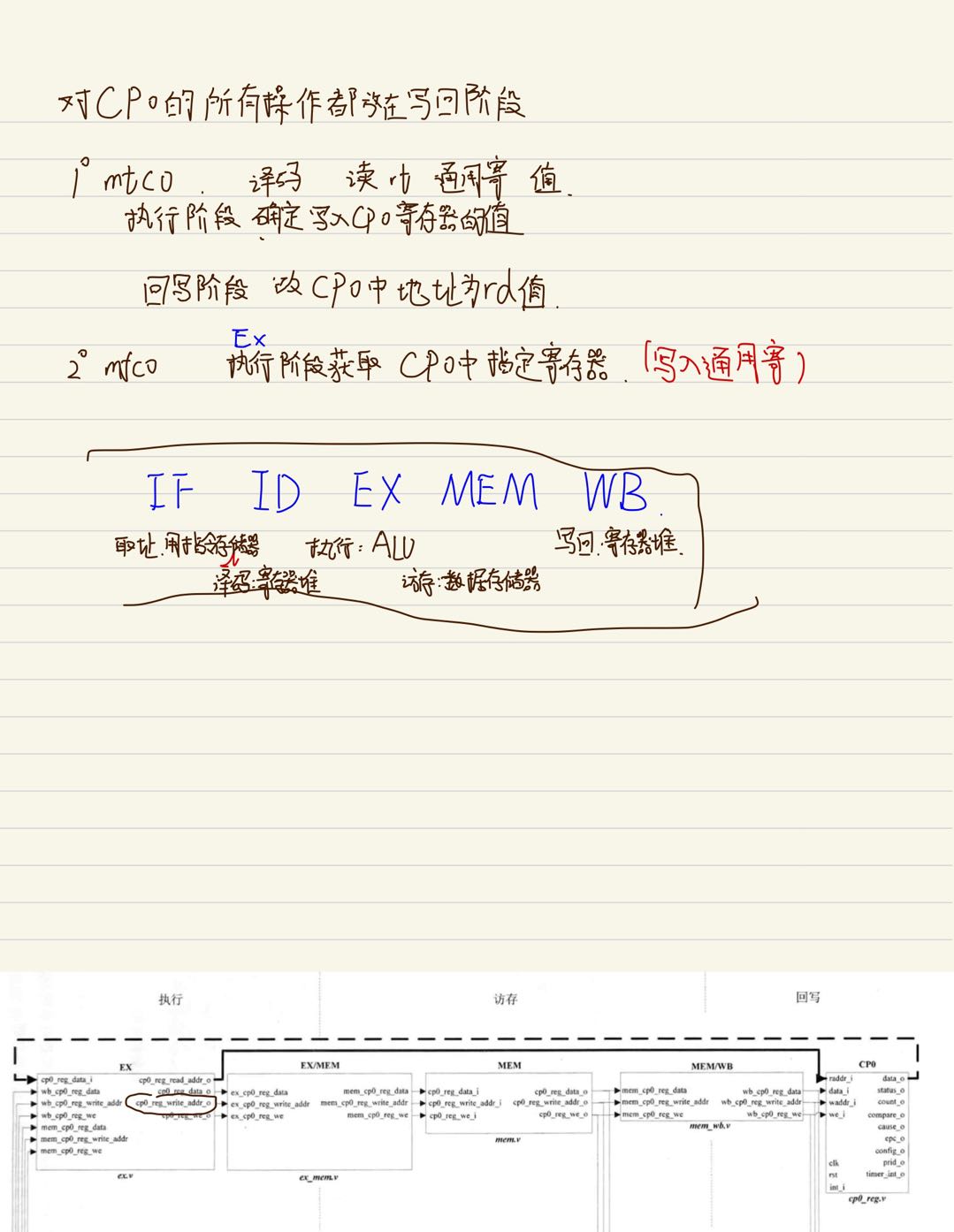
cp0\_write\_data <= 0;

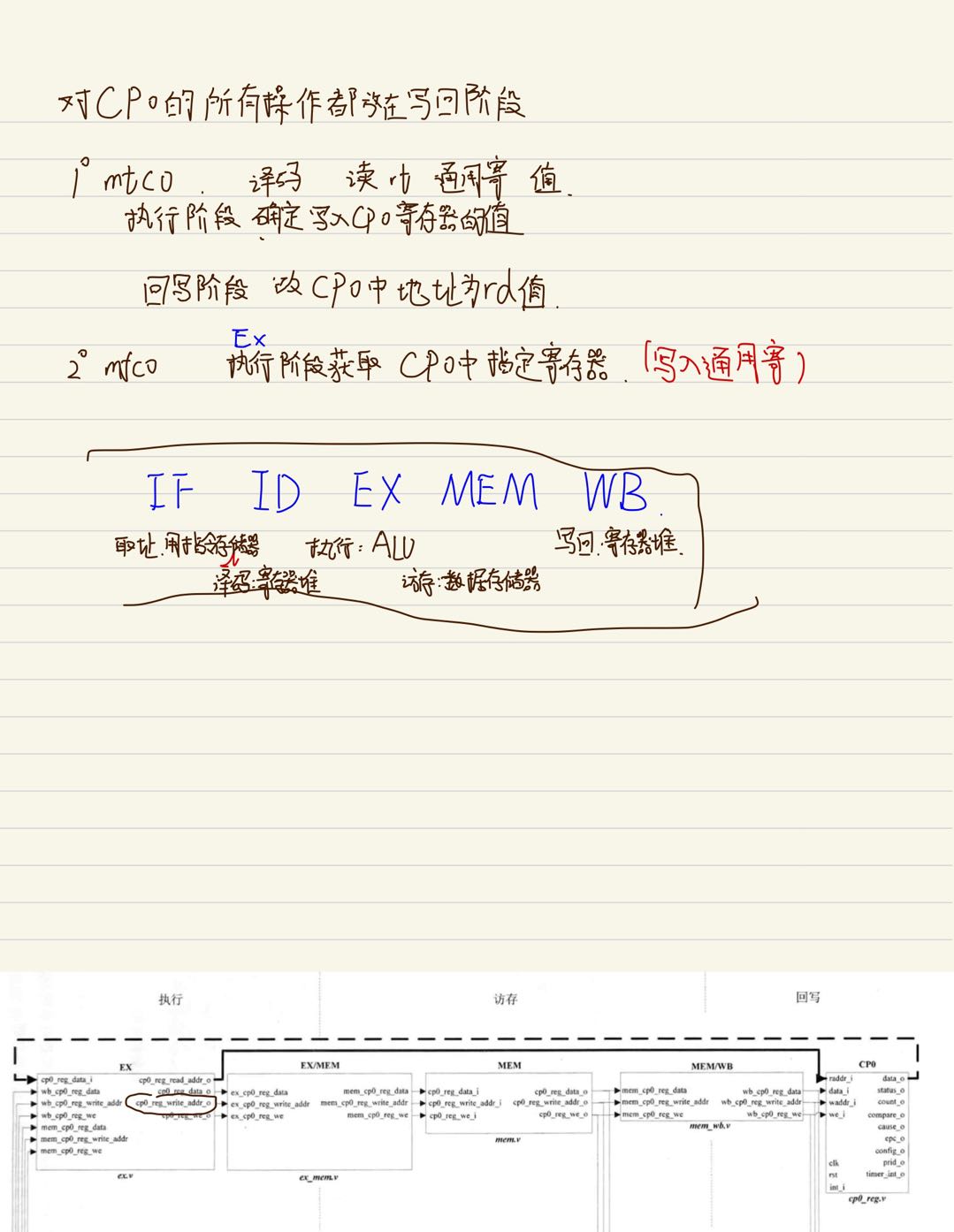
cp0\_addr <= 0;

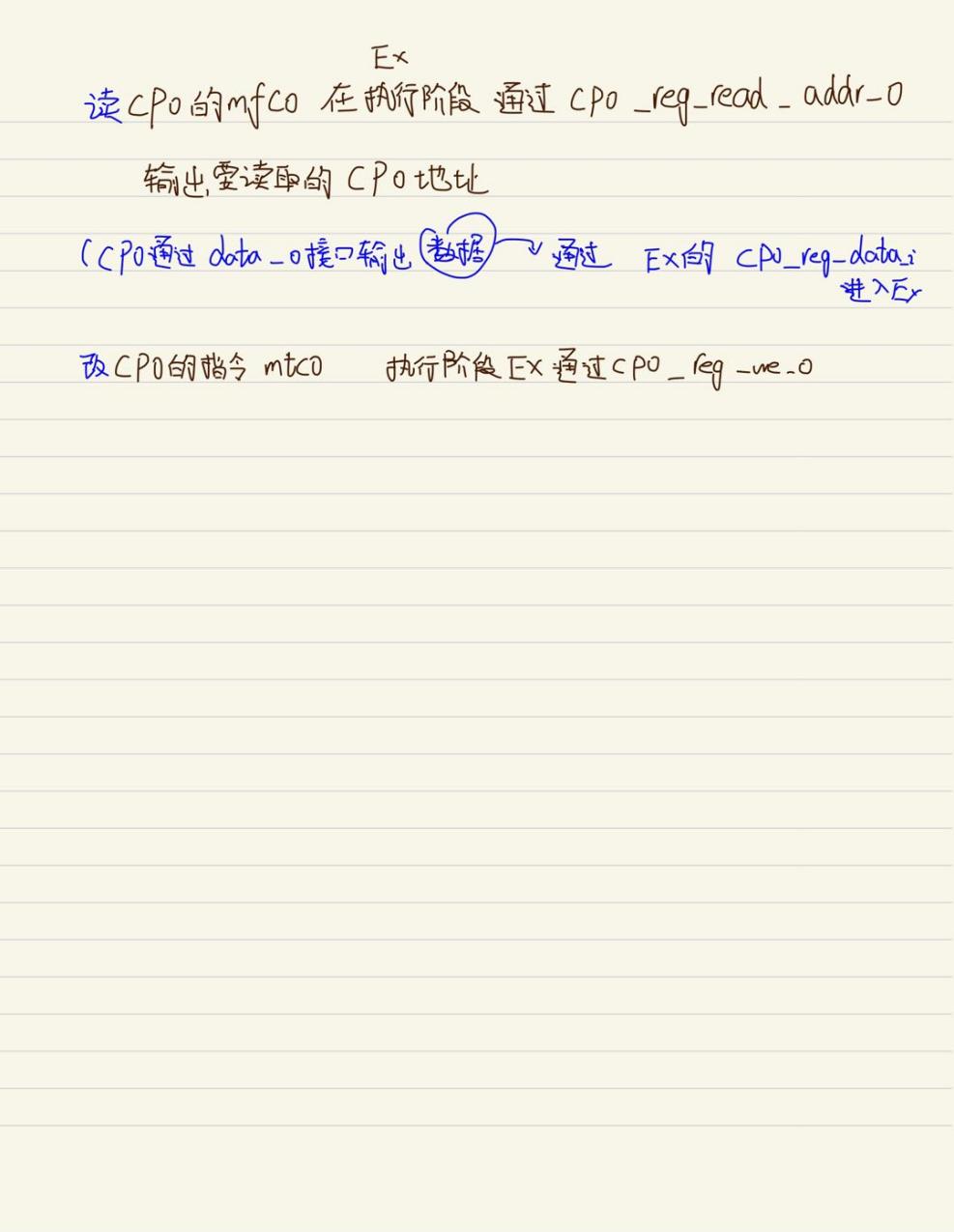
end

endcase

end







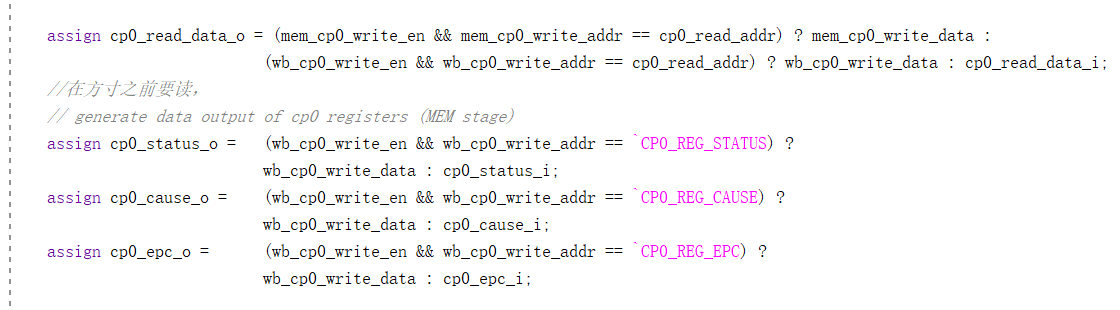
mtcO 实现思路

1. 在译码阶段依据指令，读出地址为 rt 的通用寄存器的值。
2. 在执行阶段确定要写入 CPO 中寄存器的值，其实就是译码阶段读出的地址为 rt 的通用寄存器的值，将这些信息传递到访存阶段。
3. 访存阶段再将这些信息传递到回写阶段。
4. 回写阶段依据这些信息修改 CPO 中的地址为 rd 的寄存器。

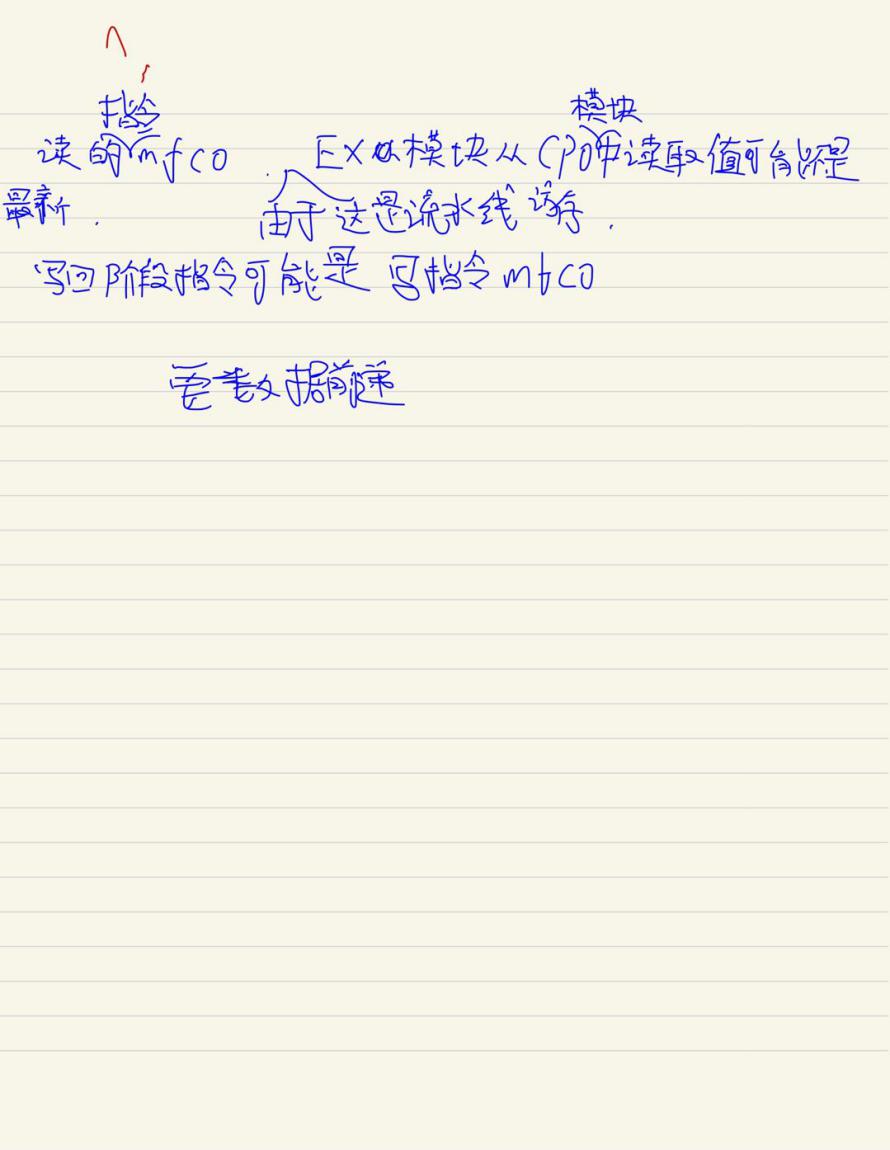
mfcO 实现思路

（1）在执行阶段获取 CPO 中指定寄存器的值，作为要写入目的通用寄存器的数据，并将这些信息传递到访存阶段。（2）访存阶段再将这些信息传递到回写阶段。（3）回写阶段依据这些信息修改地址为 rt 的通用寄存器。

1. **数据前递模块CP0proxy**



对于读取 CPO 中寄存器的指令 mfcO, EX 模块从 CPO 模块中读取的值可能不是最新的值，因为此时处于流水线访存、回写阶段的指令可能是 mtcO, 也就是说可能会修改 CPO 中的寄存器 将访存、回写阶段对CPO 中寄存器的写信息前推到执行阶段的 EX 模块，由EX 模块判断得到最新的值。这也就是图 10-4 中，MEM模块、MEM/WB 模块的输出会回送到 EX 模块的原因。



修改译码阶段, 在RegGen中实现:

`OP\_CP0: begin

reg\_read\_en\_1 <= 1;

reg\_read\_en\_2 <= 0;

reg\_addr\_1 <= rt;

reg\_addr\_2 <= 0;

End

`OP\_CP0: begin

if (rs == `CP0\_MFC0) begin

reg\_write\_en <= 1;

reg\_write\_addr <= rt;

end

else begin

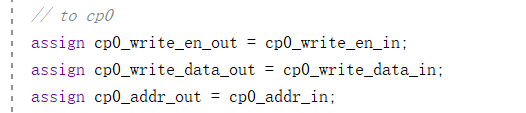
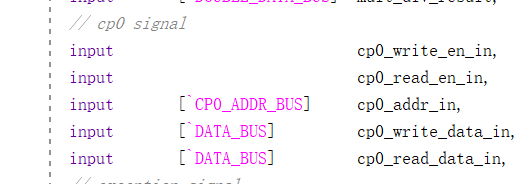
reg\_write\_en <= 0;

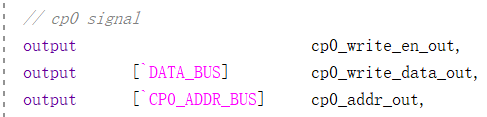
reg\_write\_addr <= 0;

end

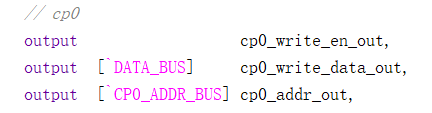
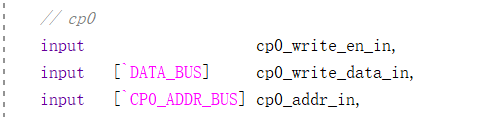
end

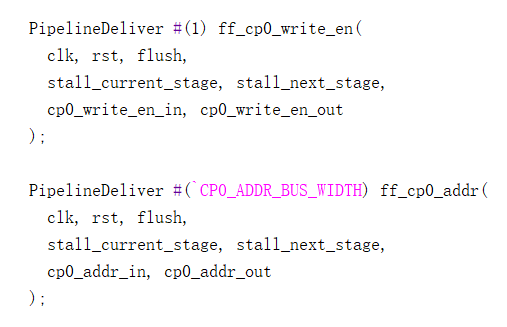
执行阶段:





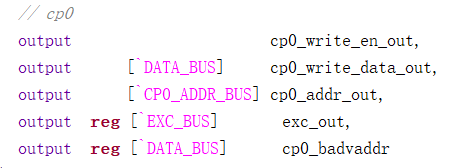
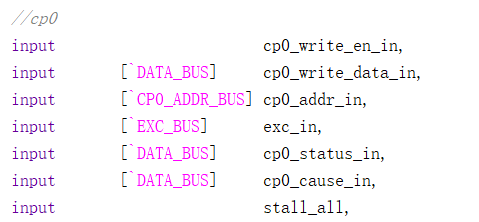
修改 EX/MEM 模块:

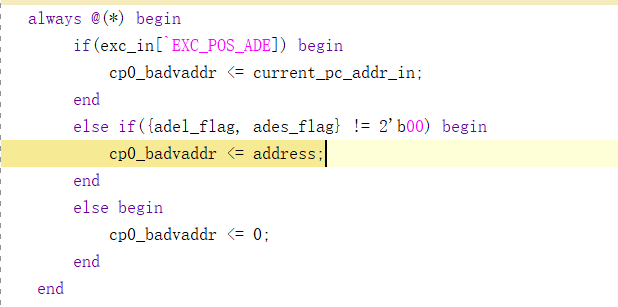




修改访存阶段:

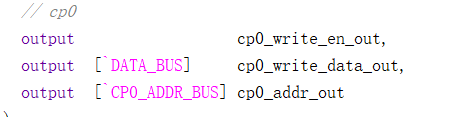
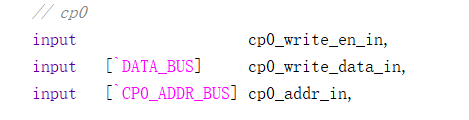
MEM 模块会将执行阶段传递过来的，对 CP0 中寄存器的写信息继续传递到流水线下一级 。

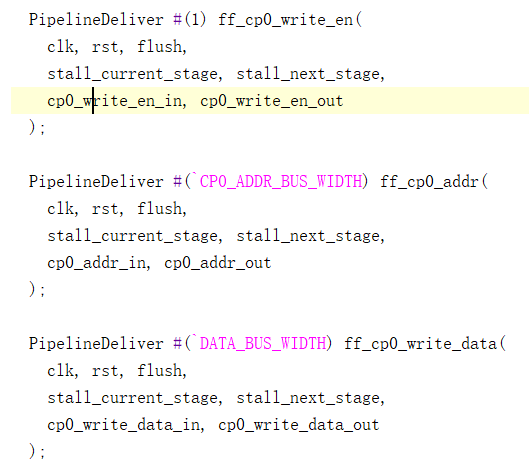




改 MEM/WB 模块:

MEM/WB 模块会将 MEM 模块传递过来的，对 CP0 中寄存器的写信息传递到回写阶段:





协处理器访问指令接口:

module CP0Gen (

input [`INST\_BUS] inst,

input [`INST\_OP\_BUS] op,

input [`REG\_ADDR\_BUS] rs,

input [`REG\_ADDR\_BUS] rd,

input [`DATA\_BUS] reg\_data\_1,

output reg cp0\_write\_en,

output reg cp0\_read\_en,

output reg [`CP0\_ADDR\_BUS] cp0\_addr,

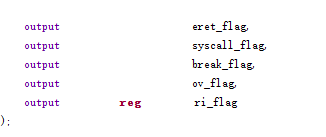
output reg [`DATA\_BUS] cp0\_write\_data

);

1. **异常处理部分**

**异常基础：**

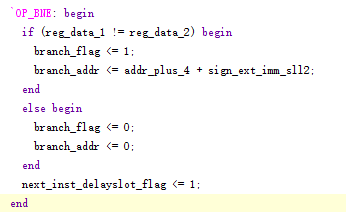
首先先实现对eret、break和syscall指令的判断。这三条指令在译码阶段就可以被识别判断出来，因此分别设置flag作为输出，先将异常的信息保存下来，沿流水线逐级后递。这是因为当一个异常发生后，系统的顺序执行会被中断掉，此时有若干条指令处在流水线上不同阶段，处理器会转移到异常处理例程，异常处理结束后返回原程序继续执行，因为不希望异常处理例程破坏原程序的正常执行，所以对于异常发生时，流水线上没有执行完的指令，必须记住它处于流水线的哪一个阶段，以便异常处理结束后能恢复执行，这便是精确异常。如此便能保证依照指令顺序逐条对指令异常进行处理。





此外，还需要注意异常发生时的pc地址。具体而言，当异常发生时，cp0的epc寄存器需要记录下此时的pc地址，在异常处理结束后返回到原来的pc地址。一种特殊情况是当发生的异常指令为延迟槽指令时，即异常指令为跳转指令的下一条指令时，返回的地址并不是当前pc地址。这是因为延迟槽指令不一定会是实际要执行的指令，它可能因为跳转指令而被跳过。但是儒过epc存的地址为延迟槽指令所在地址，则异常处理完后返回到延迟槽指令执行，再执行延迟槽指令的下一条，这与原本的指令运行过程不一。为避免出现这种情况，需要让epc存储的地址为延迟槽的上一条指令的地址，即pc-4.这样在异常处理完后会返回到跳转指令所在的地址运行，进行是否跳转的判断。

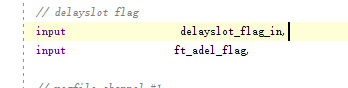
延迟槽flag的判断在id级的branchgen文件下，这里只截取了其中一条判断实现，具体为根据给定的指令是否为跳转指令来判断:



Pc0中根据传递下来的指令是否为延迟槽指令的flag，判断exc存储的地址为pc还是pc-4



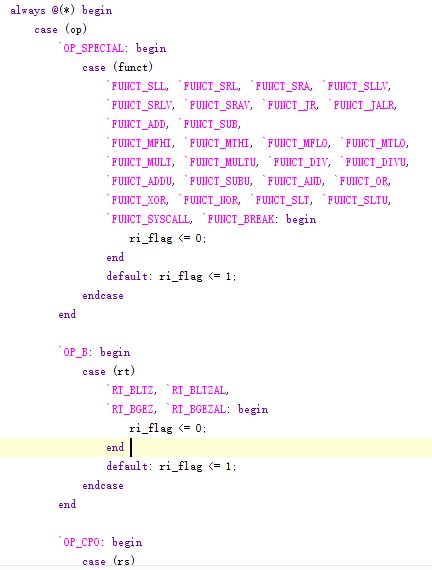
增加next\_inst\_delayslot\_f1ag信号后，需要在下一个周期再传输回ID阶段，用于告知D模块当前指令是延迟槽内的指令，因此在ID阶段加入de1ayslot\_f1ag\_in接口，用于接收来自EX阶段的next\_inst\_delayslot\_f1ag信号。



**扩展异常：**

扩展异常主要需要实现在ex级的溢出判断和在mem级的地址异常判断，并在mem级对所有异常信息作统一的处理，判断是否要进入异常处理例程，并判断处理的异常类型。

Id级新增一个无效指令的判断，具体即当判断给的指令不在case的项目中时，即认为该指令并没有被实现，为无效指令。这里截取了一部分无效指令ri的判断



首先看ex级，ex级根据运算数和运算结果为正负数来判断是否出现了运算溢出的情况，并将该flag给到exc\_out中作为异常信息存储起来。

wire ov\_flag = ((!operand\_1[31] && !operand\_2\_mux[31]) && result\_sum[31]) || ((operand\_1[31] && operand\_2\_mux[31]) && (!result\_sum[31]));

wire ov\_exc = exc\_in[`EXC\_POS\_OV] ? ov\_flag : 0;

assign exc\_out = {exc\_in[7:3], ov\_exc, exc\_in[1:0]};

Mem级则首先判断是否有地址的读写异常，分别为读地址异常adel和写地址异常ades。Mem\_sel\_in是一个存储了调用具体哪几位地址的寄存器。这两条assign实现了当读写最后一字节地址时，无需判断；读写末两字节地址时，判断地址是否为2的倍数；读写四位地址时，判断是否为4的倍数 的功能。如此可以判断读写的地址是否出现异常。当判断地址异常时，flag会被置为1。

assign adel\_flag = (mem\_read\_flag\_in && (~((mem\_sel\_in == 4'b0000) || (mem\_sel\_in == 4'b0001) ||

((mem\_sel\_in == 4'b0011) && (address[1:0] == 2'b00 || address[1:0] == 2'b10)) ||

((mem\_sel\_in == 4'b1111) && (address[1:0] == 2'b00))))) ? 1: 0;

assign ades\_flag = (mem\_write\_flag\_in && (~((mem\_sel\_in == 4'b0000) || (mem\_sel\_in == 4'b0001) ||

((mem\_sel\_in == 4'b0011) && (address[1:0] == 2'b00 || address[1:0] == 2'b10)) ||

((mem\_sel\_in == 4'b1111) && (address[1:0] == 2'b00))))) ? 1: 0;

此外，pc级还有一条pc地址异常的判断:

assign ft\_adel\_flag = pc[1:0]!=2'b00 ? 1: 0;

这些异常信息最后都会在mem级被处理，根据发生的异常，进入对应的异常处理地址，进行异常的处理，并在处理完后返回到原有的pc地址。

always @(\*) begin//异常

if(int\_occured && int\_enabled && !stall\_all) begin

exc\_out <= `EXC\_INT;

end

else if (exc\_in[`EXC\_POS\_OV]) begin

exc\_out <= `EXC\_OV;

end

else if (exc\_in[`EXC\_POS\_BP]) begin

exc\_out <= `EXC\_BP;

end

else if (exc\_in[`EXC\_POS\_SYS]) begin

exc\_out <= `EXC\_SYS;

end

else if (exc\_in[`EXC\_POS\_ERET]) begin

exc\_out <= `EXC\_ERET;

end

else if (exc\_in[`EXC\_POS\_ADE]||adel\_flag) begin

exc\_out <= `EXC\_ADEL;

end

else if (ades\_flag) begin

exc\_out <= `EXC\_ADES;

end

else if (exc\_in[`EXC\_POS\_RI]) begin

exc\_out <= `EXC\_RI;

end

else begin

exc\_out <= `EXC\_NULL;

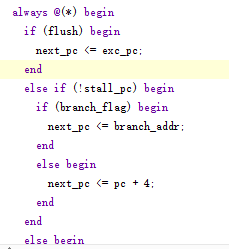
end

end

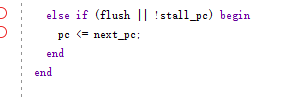
判断异常发生后，流水线中的数据会被全部丢弃，等待异常处理完后重新运行各条指令。在访存阶段会将各异常相关信号传输给PipelineControl1er模块，并通过PipelineController模块生成exc\_pc和f1ush信号，实现对流水线的控制。在pipelinecontrol中，首先会判断是否出现了异常。如若出现异常，则将flush置为1

assign flush = stall\_all ? 0 : (exc != `EXC\_NULL) ? 1 : 0;

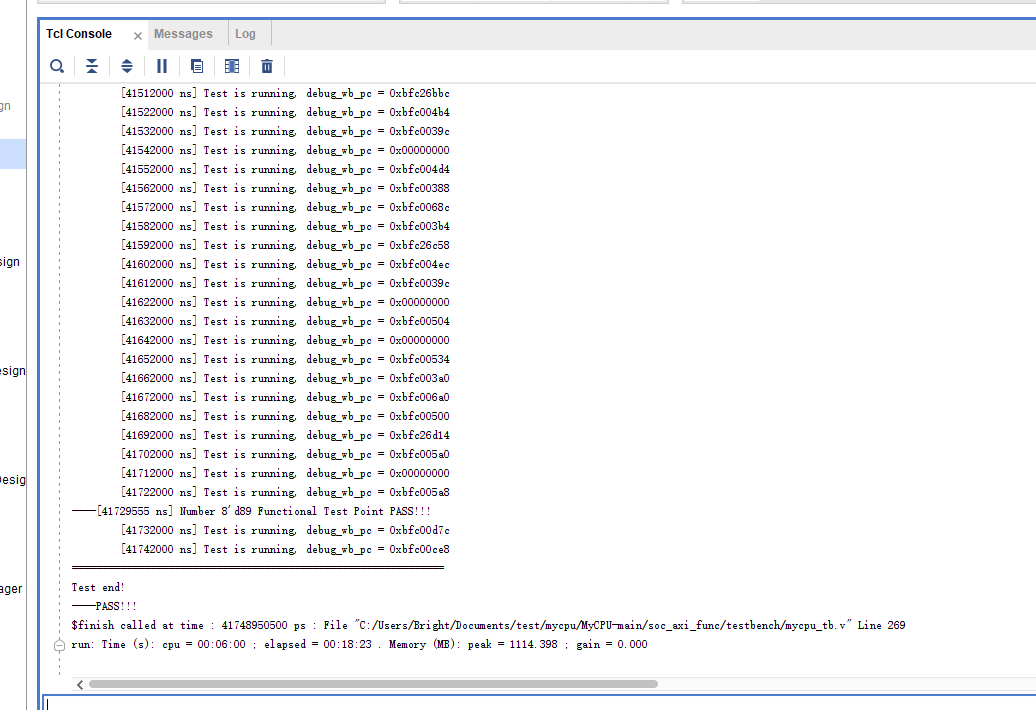
而pipelinedeliver检测到flush被置为1后，会对数据作一个清除处理，如此便实现了异常时的数据清空。此外，flush也负责给pc一个跳转地址的信号。



Pc读取到flush后便将异常处理的地址给到下一地址，在下一跳跳转过去。



**实验现象及分析：**



成功通过了全89条指令的测试。

**5 Cache部分**

**设计过程**

这部分只实现了cache的控制器模块，实际上还没有设计专门的cache寄存器模块。

首先定义cache行的格式和内存地址的格式：内存地址为12位，其中[1:0]是块内偏移，[6:2]是索引，[11:7]是Tag。Cache行为cache=V+D+Tag+Data=1+1+5+128=135位。且Cache缓存中cache大小为32块，主存大小为1024块，1块=4字，1字=32bit。

然后是cache控制状态机：

分别有4个状态，IDLE为空闲状态， CompareTag为通过Tag比较判断是否命中，执行两种情况的状态，Allocate为没有命中时的情况从内存取出数据放到cache里面，然后更新修改位D和有效位V。在WriteBack阶段执行将cache的修改行写入到内存中。

其中写回的时候，只有当没有命中且查询cache的那一行的修改位为1时，才将原本的cache行写入到内存里。然后去内存里面通过cpu发送过来的tag和index寻找cache行放入cache中，再次命中cpu即可取出。

测试方面，在这个部分中需要测试6种情况，分别是写未命中（writemiss），写命中（writehit），读命中（readhit），读未命中（readmiss），读写回（readdirtymiss），写写回（writedirtymiss）。在test.v里面测试。

这里我们主要是使用cache第一行和第二行以及cache第一行对应于内存里面的第二块进行测试。且设置cache第二行第一个对应的内存里面存的是1，其余内存里面都是0，也就是mem[4]=32’d1。首先是writemiss部分，此时cache行未命中，首先通过cpu那边给出的tag和index将内存里面的那一行数据都取出，然后放到cache行里，并将有效位设置为1，修改位设置为0，然后将cpu那边给出的数据写入对应cache行里面，并且将修改位设置为1.，先使用以下代码测试：

WriteMiss:begin

cpu\_req\_rw<=1'b1;

cpu\_req\_addr<=12'd0;

cpu\_data\_write<=32'd8;

cpu\_req\_valid<=1'b1;

end

cpu\_req\_rw表示写有效， cpu\_req\_addr表示CPU发过来的内存访问地址，cpu\_data\_write表示CPU写入内存的数据，cpu\_req\_valid<=1'b1表示从cpu发过来的这些数据和信号有效，即cache控制器中状态机开始从IDLE空闲状态转变为下一状comparetag。

然后是writehit部分：

WriteHit:begin

cpu\_req\_rw<=1'b1;

cpu\_req\_addr<=12'd0;

cpu\_data\_write<=32'd9;

cpu\_req\_valid<=1'b1;

End

接下来是readmiss：

ReadMiss:begin

cpu\_req\_rw<=1'b0;

cpu\_req\_addr<=12'd4;

cpu\_data\_write<=cpu\_data\_write;

cpu\_req\_valid<=1'b1;

End

readhit 部分：

ReadHit:begin

cpu\_req\_rw<=1'b0;

cpu\_req\_addr<=12'd0;

cpu\_data\_write<=cpu\_data\_write;

cpu\_req\_valid<=1'b1;

End

writedirtymiss部分：

WriteDirtyMiss:begin

cpu\_req\_valid<=1'b1;

cpu\_req\_rw<=1'b1;

cpu\_data\_write<=32'd6;

cpu\_req\_addr<=12'd128;

end

readDirtyMiss部分：

ReadDirtyMiss:begin

cpu\_req\_valid<=1'b1;

cpu\_req\_rw<=1'b0;

cpu\_req\_addr<=12'd0;

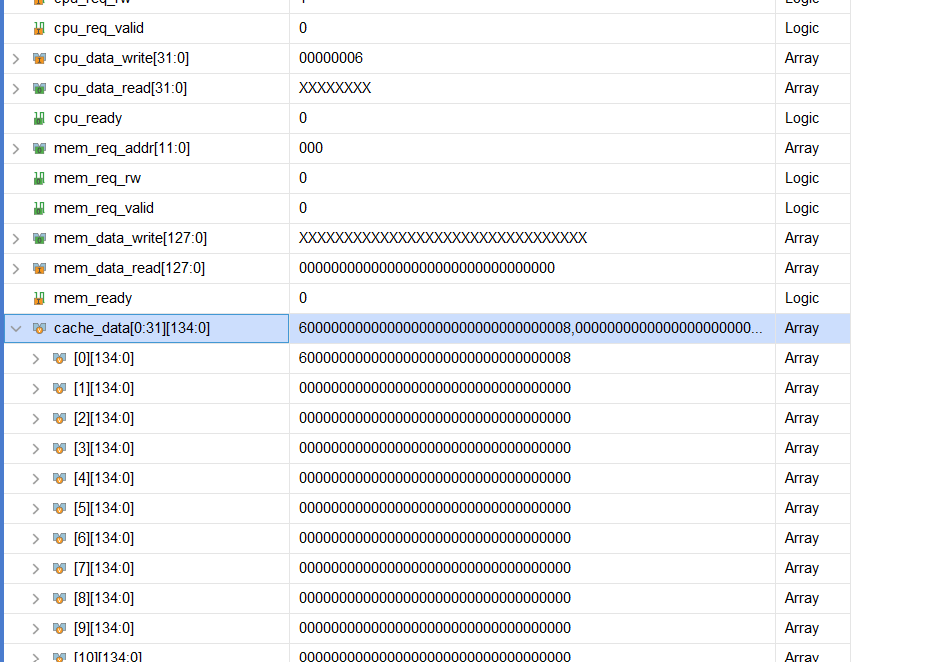
cpu\_data\_write<=cpu\_data\_write;

End

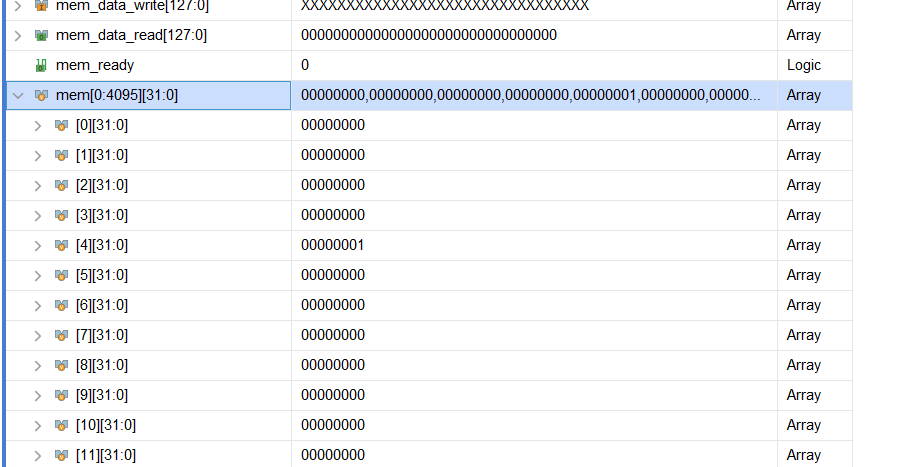
**实验现象与分析**

readmiss的情况：

Cache部分：



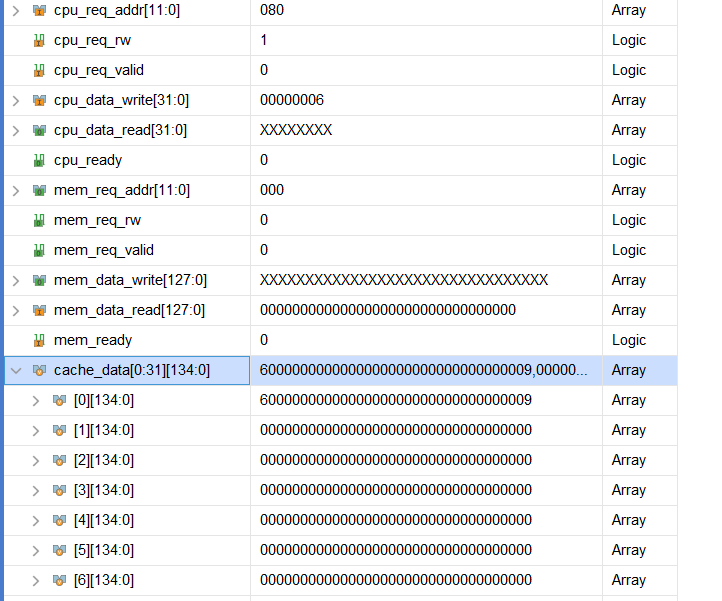
Mem的情况：



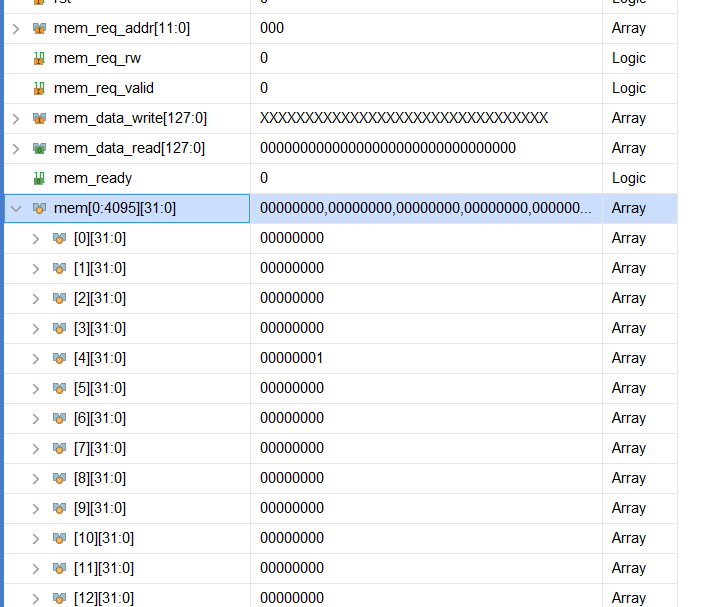
此时cache第一行中6位0110，由于我们是135位cache行所以最高位0无效，第一位为1表示V=1,有效，第二位为1表示D=1，已修改，最后为8表示数据8以及写入，此时cache行第一个字对应与mem[0]，此时并未写入。

然后是命中的情况：

Cache部分：



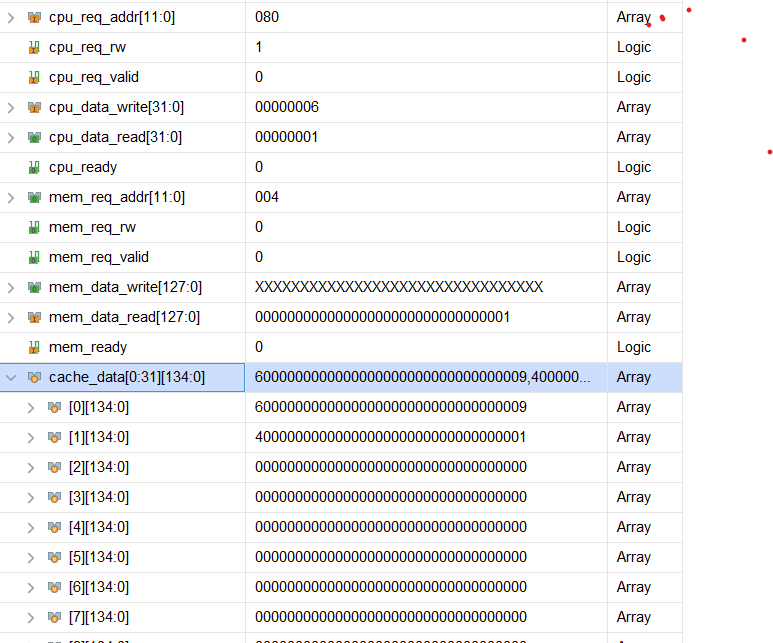
Mem部分：



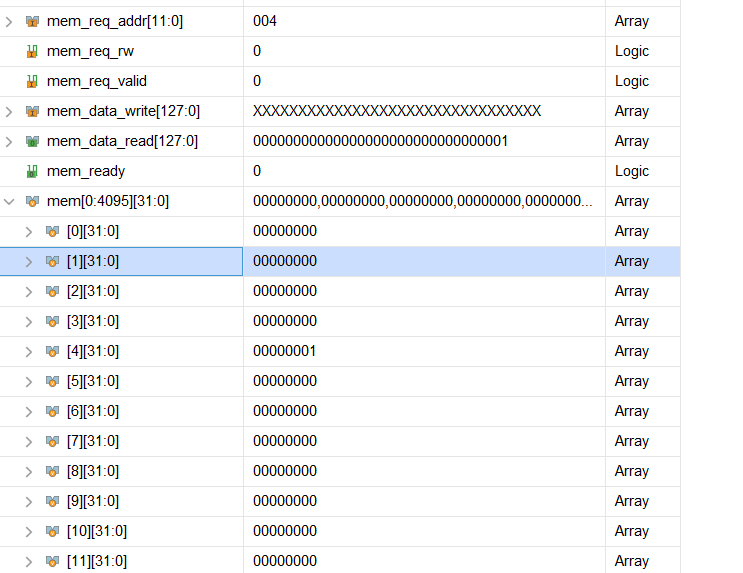
此时cache中的数据部分值修改为了9，此时V和D均为1，但是没有写回到mem中，因为此时tag是命中的。只有当不命中的时候且cache那一行是修改过的时候才写回。

Readmiss的情况：

Cache部分：



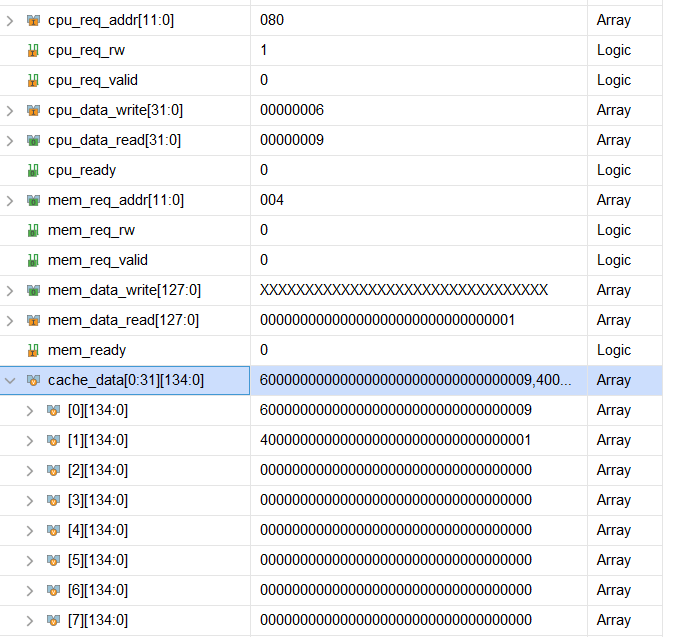
Mem部分：



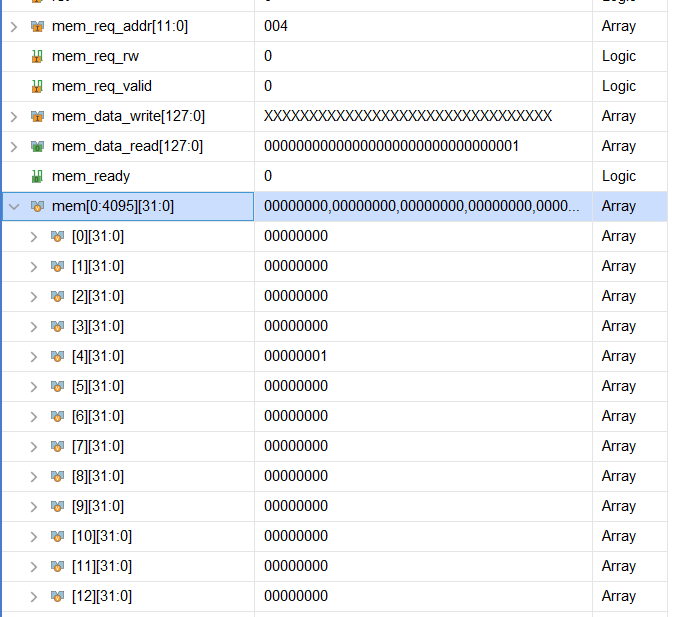
此时从mem[4]那里将数据读取到了cache第二行，4表示0100，即有效位为1，修改位为0。

接下来是 readhit:

Cache部分：



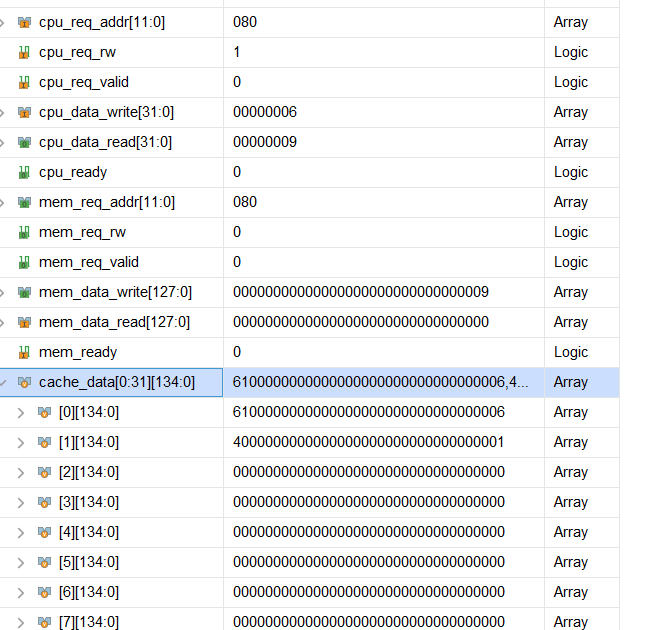
Mem部分：



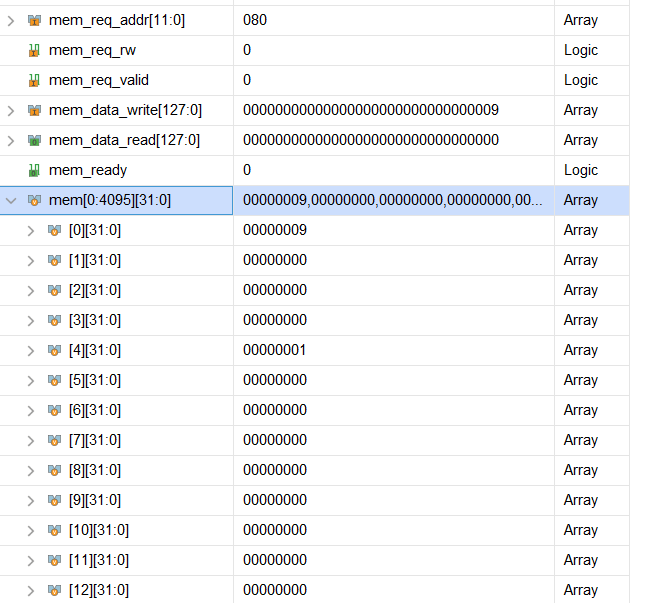
和上一次的结果一样，其实这里不太好看是否命中，不过可以从cpu\_data\_read那得到cpu接受的数据是9，即命中。

writedirty的情形：

Cache部分：



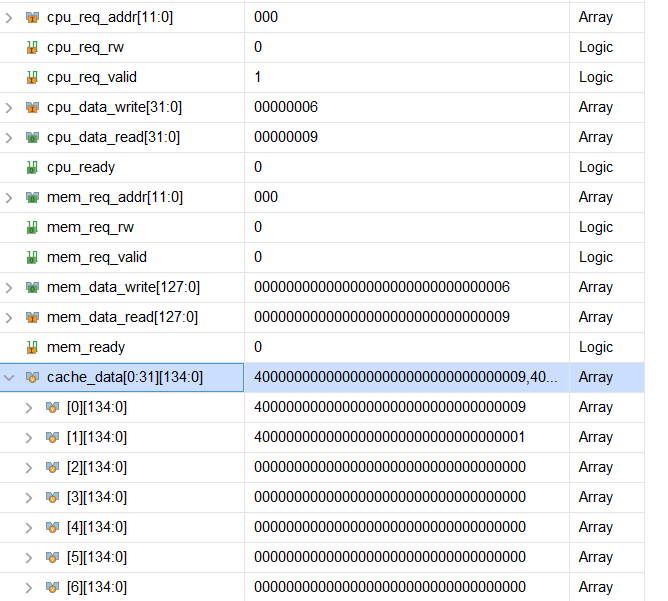
Mem部分：



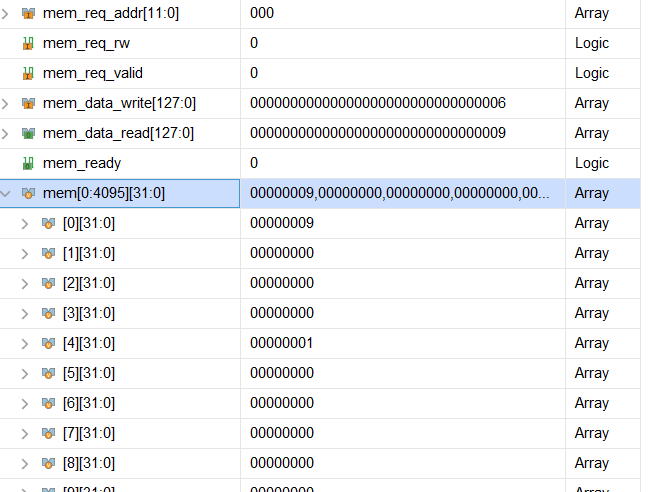
此时cache第一行的数据写回了mem中，第一个字的数据为9，且cache第一行的数据为6，为新写入的数据，此时写入的cache第一行的第一个字的数据对应于mem[128]。

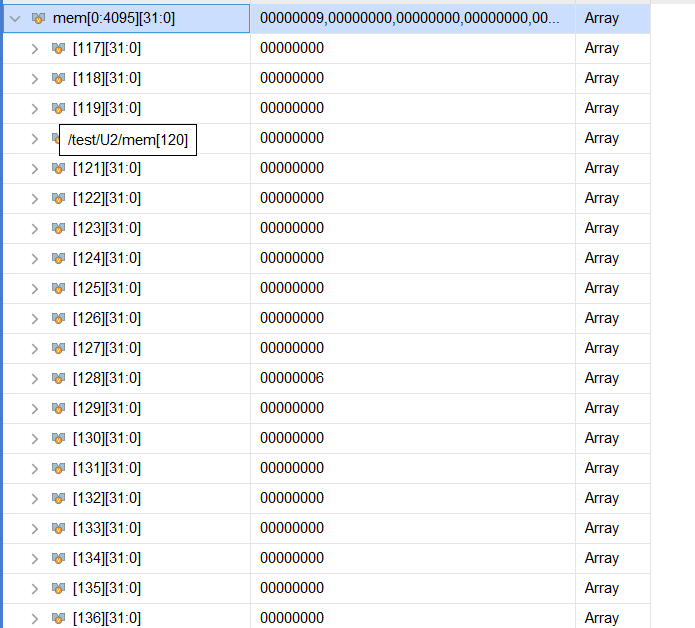
最后是readdirtymiss部分：

Cache部分：



Mem部分：





首先此时cache第一行写的数据对应于mem[128]，由于我们此时读取的数据地址为12’d0即mem[0]的数据，且上一次即在writedirtymiss那写入过，此时修改位为1，所以得需要将cache第一行写入到mem[128:131],再从内存里面取出mem[0]=9,然后将它放到cache第一行，并将修改位设置为1，所以cache第一行此时的数据中4表示只有有效位为1,9表示将mem[0]取出。mem[128]=6表示已经将数据写入到内存中。

**5 组内成员主要工作及贡献比例**

刘伟浩 负责实现并集成多周期华莱士树乘法器 20%

秦铭壕 负责实现并集成迭代除法器模块 20%

郝鑫 负责研究并实现cache模块，完成相关测试 20%

罗奇峰 负责实现CP0协处理器及相关指令 20%

吴宇航 负责实现特权指令及扩展所有异常指令 20%

**四：结论（讨论）**

（对照课设内容，简要总结课设期间完成的主要工作：对照课设目的，着重说明通过课程设计过程所取得的收获和能力的达成情况；存在的问题及可能的改进方向；其它需要说明的问题）

1. **结论**（实验总结）

刘伟浩：通过本次实验，我对于乘法器的实现有了比较大的认识并进行了实践。首先是了解了华莱士树乘法器的基本原理，并自己实现了一个多周期的华莱士树乘法。同时也探索出了控制CPU读取乘法器结果的实现方式。

郝鑫：通过本次计组设计实验，实现了hilo寄存器的相关指令以及其数据前递，深刻理解了数据前递，且通过实现cache控制部分以及测试仿真过程中遇到问题并解决，也加深了我对于cache的理解，也听了同组成员对于各自工作的汇总，虽然具体细节不是完全懂，但是大致的原理和流程是了解了，提高了我对于计算机运行机制的理解。

秦铭壕：由除法器的仿真分析以及goldtrace的测试可知，当前基于试商法的除法器实现成功。但是仍有较大的改进空间，例如当前除法器只能用于整数除法，并且只能得到整数商与余数的形式，更高级的除法器能够实现浮点数的除法，以后我们小组将尝试Goldschmidt方法、泰勒级数展开等更高级的算法实现更全面的除法功能

罗奇峰：协处理器部分完成了对多个状态寄存器CP0的定义书写，主要寄存器有BadVaddr、Count、Status、Cause、EPC等。在实现五个主要寄存器之后，完成对协处理器访问指令mtc0和mfc0，以及为解决数据相关冲突而设计的数据前递部件cp0ReadProxy，在此基础上开展对异常处理部分的拓展。

吴宇航：通过本次实验，我了解了异常判断和处理的具体流程，对异常类型有了清楚的认知，并对于不同类型的异常分别在哪判断、如何判断是否发生有了深刻的理解。此外，对于精确异常、异常发生后的流水线操作等内容也有了较深的感悟。在小组实验的过程中，我们也通过向其他小组成员阐述的方式来判断自己是否完全明白了自己负责的部分，而在这一过程中，我也对他人所负责的部分有了一些了解。总的来说，这次实验让我受益良多，对cpu的运行也有了一些自己的体悟。路漫漫其修远兮，我将继续努力学习相关知识，积极探索，加深对cpu的了解和理解。

**2、讨论**（问题归纳，课程建议等）

刘伟浩：对于本课程，希望能够多在课堂上学习一些指示，而不是让助教用仅有的4节课的时间把计算机组成原理这一门庞大的课程的知识灌输给我们。（必修课计算机组成原理课程中讲到的内容实在太少，满足不了课设的需求）

郝鑫：本次实验在cache部分只完成了控制器部分，并未整合到工程中，存在以下问题：1.cache得单独设立一个cache寄存器模块，用于进行数据和信号交换。2.主存地址在项目中并不是12位，需要进行转换，否则无法通过store和load指令进行对cache的存取。3.cache里面的那些从cpu发出的信号如主存地址还有数据等还需要与项目进行衔接和转换。只有cache在整个框架下运行指令并执行的情况才是完整的cache。

吴宇航：异常部分的难点主要为内容多且杂，分布在各个不同的文件中，相互调用其他文件的output。对于各个文件的端口与数据的传递，我仍有一些困惑的地方。不过，感谢于小组成员以及其他组同学的热心帮助，我能够完成这一部分的任务，回想起来仍感到不胜感激。

**附:**

**TinyMIPS实现的MIPS指令:**

表1-1 算术运算指令

|  |  |
| --- | --- |
| 指令助记格式 | 指令功能简述 |
| ADDU rd,rs,rt | 无符号加（无溢出异常） |
| ADDIU rt,rs,imm | 无符号加立即数（无溢出异常） |
| SUBU rd,rs,rt | 无符号减（无溢出异常） |
| SLT rd,rs,rt | 有符号小于置1 |
| SLTU rd,rs,rt | 无符号小于置1 |

表1-2 逻辑运算指令

|  |  |
| --- | --- |
| 指令助记格式 | 指令功能简述 |
| AND rd,rs,rt | 按位与 |
| LUI rt,imm | 寄存器高位置立即数 |
| OR rd,rs,rt | 按位或 |
| XOR rd,rs,rt | 按位异或 |

表1-3 移位指令

|  |  |
| --- | --- |
| 指令助记格式 | 指令功能简述 |
| SLL rd,rt,sa | 立即数逻辑左移 |
| SLLV rd,rs,rt | 寄存器逻辑左移 |
| SRAV rd,rs,rt | 寄存器算术右移 |
| SRLV rd,rt,sa | 寄存器逻辑右移 |

表1-4 分支跳转指令

|  |  |
| --- | --- |
| 指令助记格式 | 指令功能简述 |
| BEQ rs,rt,offest | 相等时分支转移 |
| BNE rs,rt,offest | 不等时分支转移 |
| JAL target | 无条件直接跳转，并保存返回地址 |
| JALR rd,rs | 无条件寄存器跳转，并保存返回地址 |

表1-5 访存指令

|  |  |
| --- | --- |
| 指令助记格式 | 指令功能简述 |
| LB rt,offest(base) | 访存读字节（8位），有符号扩展 |
| LBU rt,offest(base) | 访存读字节（8位），无符号扩展 |
| LW rt,offest(base) | 访存读字（32位） |
| SB rt,offest(base) | 访存写字节（8位） |
| SW rt,offest(base) | 访存写字（32位） |

**可选的MIPS指令见附件A02:**

**北京科技大学实验报告**

学院： 专业： 班级：

姓名： 学号： 实验日期： 年 月 日

**五、教师评审**

|  |  |
| --- | --- |
| **教师评语** | **实验成绩** |
| （虽然课设主要侧重于验证问题，但是建议各位老师从解决“工程技术问题”，特别是“复杂工程问题”的角度去评审学生课设过程及代码阅读报告，主要内容包括提出问题、分析问题、解决问题及验证问题。**要有较详细的评审意见**。）  签名：  日期： |  |