石 家 庄 铁 道 大 学

**实 验 报 告**

课程名称计算机组成原理课程设计 计算机科学与技术 系 3 班

实验组成员：姓名 崔金泽 学号 20192163

姓名 苗春雨 学号 20193985

姓名 李子怡 学号 20193984

实验日期 2022 年 10 月 10 日 指导教师 赵文彬

**目 录**

[实验一 数据运算：定点加法 3](#_Toc116115187)

[一、实验分析说明 3](#_Toc116115188)

[二、实验设计思路 3](#_Toc116115189)

[三、实验源代码 4](#_Toc116115190)

[adder.v 4](#_Toc116115191)

[adder\_display.v 5](#_Toc116115192)

[四、实验实现截图 8](#_Toc116115193)

[五、收获心得体会 10](#_Toc116115194)

[实验二 数据运算：定点乘法 11](#_Toc116115195)

[一、实验分析说明 11](#_Toc116115196)

[二、实验设计思路 11](#_Toc116115197)

[三、实验源代码 12](#_Toc116115198)

[multiply.v 13](#_Toc116115199)

[四、实验实现截图 15](#_Toc116115200)

[五、收获心得体会 18](#_Toc116115201)

[实验三 寄存器堆实现 19](#_Toc116115202)

[一、实验分析说明 19](#_Toc116115203)

[二、实验设计思路 19](#_Toc116115204)

[三、实验源代码 20](#_Toc116115205)

[regfile.v 20](#_Toc116115206)

[四、实验实现截图 23](#_Toc116115207)

[五、收获心得体会 25](#_Toc116115208)

[实验四 ALU模块实现 26](#_Toc116115209)

[一、实验分析说明 26](#_Toc116115210)

[二、实验设计思路 26](#_Toc116115211)

[三、实验源代码 28](#_Toc116115212)

[alu.v 28](#_Toc116115213)

[四、实验实现截图 32](#_Toc116115214)

[加法 33](#_Toc116115215)

[减法 33](#_Toc116115216)

[按位与 34](#_Toc116115217)

[异或 34](#_Toc116115218)

[按位或 35](#_Toc116115219)

[或非 35](#_Toc116115220)

[逻辑左移 36](#_Toc116115221)

[五、收获心得体会 37](#_Toc116115222)

[实验五 存储器实现 37](#_Toc116115223)

[一、实验分析说明 37](#_Toc116115224)

[二、实验设计思路 37](#_Toc116115225)

[三、实验源代码 38](#_Toc116115226)

[data\_ram\_display.v 38](#_Toc116115227)

[inst\_rom\_display.v 42](#_Toc116115228)

[四、实验实现截图 44](#_Toc116115229)

[五、收获心得体会 47](#_Toc116115230)

[实验六 单周期CPU实现 48](#_Toc116115231)

[一、实验分析说明 48](#_Toc116115232)

[二、实验设计思路 48](#_Toc116115233)

[三、实验源代码 52](#_Toc116115234)

[inst\_rom.v 52](#_Toc116115235)

[regfile.v 54](#_Toc116115236)

[alu.v 57](#_Toc116115237)

[data\_mem.v 61](#_Toc116115238)

[四、实验实现截图 63](#_Toc116115239)

[五、收获心得体会 65](#_Toc116115240)

[实验七 多周期CPU实现 66](#_Toc116115241)

[一、实验分析说明 66](#_Toc116115242)

[二、实验设计思路 66](#_Toc116115243)

[三、实验源代码 70](#_Toc116115244)

[multi\_cycle\_cpu.v 70](#_Toc116115245)

[四、实验实现截图 76](#_Toc116115246)

[五、收获心得体会 78](#_Toc116115247)

[实验八 静态5级流水线CPU实现 78](#_Toc116115248)

[一、实验分析说明 78](#_Toc116115249)

[二、实验设计思路 78](#_Toc116115250)

[三、实验源代码 82](#_Toc116115251)

[pipeline\_cpu.v 82](#_Toc116115252)

[四、实验实现截图 89](#_Toc116115253)

[五、收获心得体会 89](#_Toc116115254)

[实验九 优化CPU系统 90](#_Toc116115255)

[一、实验分析说明 90](#_Toc116115256)

[二、实验设计思路 90](#_Toc116115257)

[三、实验源代码 90](#_Toc116115258)

[define.v 90](#_Toc116115259)

[mycpu\_top 94](#_Toc116115260)

[thinpad\_top 113](#_Toc116115261)

[Pc 123](#_Toc116115262)

[pre\_icache 124](#_Toc116115263)

[line2\_pre\_decoder 125](#_Toc116115264)

[line2\_if\_id 144](#_Toc116115265)

[line2\_if\_ex 146](#_Toc116115266)

[line2\_dcache\_mem 150](#_Toc116115267)

[line2\_mem\_wb 151](#_Toc116115268)

[四、实验实现截图 152](#_Toc116115269)

[五、收获心得体会 152](#_Toc116115270)

|  |  |  |
| --- | --- | --- |
| 实验一 数据运算：定点加法一、实验分析说明 1.熟悉 LS-CPU-EXB-002 实验箱和软件平台 。  2.掌握利用该实验箱各项功能开发组成原理和体系结构实验的方法 。  3.理解并掌握加法器的原理和设计 。  4.熟悉并运用 verilog 语言进行电路设计 。  5.为后续设计 cpu 的实验打下基础 。 二、实验设计思路 计算机中加减法利用补码来实现，公式为**[X+Y]补 = [X]补 + [Y]补，[X-Y]补 = [X]补 + [-Y]补。**  eg:1.x=+1001,y=+0101,求x+y 2.x+1011,y=-0101,求x+y.  01001 01011  + 00101 + 11011  01110 00110  所以x=y=+1110 所以x+y=+0110  在定点整数运算过程中如出现大于字长绝对值的现象称为“溢出”，在定点机中出现溢出时其结果是不正确的，故运算器必须能检测出溢出。遵循规则：1.两个符号位都看作数码一样参加运算  **add.v原理:** 实验中有 2 个32 位数的输入和1个进位输入，产生 1个 32 位的加法和 结果和 1 个向高位的进位。本实验提供的参考设计是直接写“+”号实现加法功能的， 这样的写法综合工具会调用内部的模块库的加法器来实现，往往会比自行设计的加法 模块更高效和省资源。  **testbench.v原理**：实验中需要产生的输入激励就是2个加数和 1个低位进位信号，在该激励输入到加法功能模块中后，会输出加法结果和向高位的进位信号。仿真的过程中会产生波形文件，可以通过观察波形文件确定功能的正确性，在出错的情况下可以定位错误位置。  1 .阅读 LS-CPU-EXB-002 实验箱相关文档，熟悉硬件平台,特别需要掌握利用显示屏观察特定信号的方法 。学习软件平台和设计流程 。  2 .熟悉计算机中加法器的原理 。  3 .设计本次实验的方案，画出结构框图 。外围模块 adder\_display 内部调用了 adder 和 lcd\_module 模块。 在外部除了时钟和复位信号外，还有 cin 和 input\_sel 通过拨码开关输入，以及 cout 输出到 led 灯上  4 .根据设计的实验方案，使用 verilog 编写相应代码 。  5 .对编写的代码进行仿真，得到正确的波形图 。  6 .将以上设计作为一个单独的模块，设计一个外围模块去调用该模块，外围模块中需调用封装好的触摸屏模块，显示两个加数和加法结果，且需要利用触摸功能输入两个加数 。  7. 将编写的代码进行综合布局布线，并下载到实验箱中的 FPGA 板上进行演示。  8.在LCD 触摸屏上会分别显示2 个加数和加法结果，最右侧的led 灯为向高位的进位，输入为 0 是亮，为 1 是不亮。拨码开关最右侧的开关用来选择触摸屏输入的数据为加数1 还是加数 2。 三、实验源代码adder.v  |  | | --- | | `timescale 1ns / 1ps  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //   > 文件名: adder.v  //   > 描述  ：加法器，直接使用"+"，会自动调用库里的加法器  //   > 作者  : 崔金泽 苗春雨 李子怡  //   > 日期  : 2022-10-04  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  module adder(      input  [31:0] operand1,      input  [31:0] operand2,      input         cin,      output [31:0] result,      output        cout      );      assign {cout,result} = operand1 + operand2 + cin;  endmodule |  adder\_display.v  |  | | --- | | //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //   > 文件名: adder\_display.v  //   > 描述  ：加法器显示模块，调用FPGA板上的IO接口和触摸屏  //   > 作者  : 崔金泽 苗春雨 李子怡  //   > 日期  : 2022-10-04  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  module adder\_display(      //时钟与复位信号       input clk,      input resetn,    //后缀"n"代表低电平有效      //拨码开关，用于选择输入数和产生cin      input input\_sel, //0:输入为加数1(add\_operand1);1:输入为加数2(add\_operand2)      input sw\_cin,      //led灯，用于显示cout      output led\_cout,        //触摸屏相关接口，不需要更改      output lcd\_rst,      output lcd\_cs,      output lcd\_rs,      output lcd\_wr,      output lcd\_rd,      inout[15:0] lcd\_data\_io,      output lcd\_bl\_ctr,      inout ct\_int,      inout ct\_sda,      output ct\_scl,      output ct\_rstn      );  //-----{调用加法模块}begin      reg  [31:0] adder\_operand1;      reg  [31:0] adder\_operand2;      wire        adder\_cin;      wire [31:0] adder\_result  ;      wire        adder\_cout;      adder adder\_module(          .operand1(adder\_operand1),          .operand2(adder\_operand2),          .cin     (adder\_cin     ),          .result  (adder\_result  ),          .cout    (adder\_cout    )      );      assign adder\_cin = sw\_cin;      assign led\_cout  = ~adder\_cout;  //-----{调用加法模块}end  //---------------------{调用触摸屏模块}begin--------------------//  //-----{实例化触摸屏}begin  //此小节不需要更改      reg         display\_valid;      reg  [39:0] display\_name;      reg  [31:0] display\_value;      wire [5 :0] display\_number;      wire        input\_valid;      wire [31:0] input\_value;      lcd\_module lcd\_module(          .clk            (clk           ),   //10Mhz          .resetn         (resetn        ),          //调用触摸屏的接口          .display\_valid  (display\_valid ),          .display\_name   (display\_name  ),          .display\_value  (display\_value ),          .display\_number (display\_number),          .input\_valid    (input\_valid   ),          .input\_value    (input\_value   ),          //lcd触摸屏相关接口，不需要更改          .lcd\_rst        (lcd\_rst       ),          .lcd\_cs         (lcd\_cs        ),          .lcd\_rs         (lcd\_rs        ),          .lcd\_wr         (lcd\_wr        ),          .lcd\_rd         (lcd\_rd        ),          .lcd\_data\_io    (lcd\_data\_io   ),          .lcd\_bl\_ctr     (lcd\_bl\_ctr    ),          .ct\_int         (ct\_int        ),          .ct\_sda         (ct\_sda        ),          .ct\_scl         (ct\_scl        ),          .ct\_rstn        (ct\_rstn       )      );  //-----{实例化触摸屏}end  //-----{从触摸屏获取输入}begin  //根据实际需要输入的数修改此小节，  //建议对每一个数的输入，编写单独一个always块      //当input\_sel为0时，表示输入数为加数1，即operand1      always @(posedge clk)      begin          if (!resetn)          begin              adder\_operand1 <= 32'd0;          end          else if (input\_valid && !input\_sel)          begin              adder\_operand1 <= input\_value;          end      end        //当input\_sel为1时，表示输入数为加数2，即operand2      always @(posedge clk)      begin          if (!resetn)          begin              adder\_operand2 <= 32'd0;          end          else if (input\_valid && input\_sel)          begin              adder\_operand2 <= input\_value;          end      end  //-----{从触摸屏获取输入}end  //-----{输出到触摸屏显示}begin  //根据需要显示的数修改此小节，  //触摸屏上共有44块显示区域，可显示44组32位数据  //44块显示区域从1开始编号，编号为1~44，      always @(posedge clk)      begin          case(display\_number)              6'd1 :              begin                  display\_valid <= 1'b1;                  display\_name  <= "ADD\_1";                  display\_value <= adder\_operand1;              end              6'd2 :              begin                  display\_valid <= 1'b1;                  display\_name  <= "ADD\_2";                  display\_value <= adder\_operand2;              end              6'd3 :              begin                  display\_valid <= 1'b1;                  display\_name  <= "RESUL";                  display\_value <= adder\_result;              end              default :              begin                  display\_valid <= 1'b0;                  display\_name  <= 40'd0;                  display\_value <= 32'd0;              end          endcase      end  //-----{输出到触摸屏显示}end  //----------------------{调用触摸屏模块}end---------------------//  endmodule |  四、实验实现截图  五、收获心得体会 在本次实验中我学会了如何掌握利用该实验箱各项功能开发组成原理和体系结构实验的方法 。也理解并掌握了加法器的原理和设计 。学会了调用定点加法模块的外围模块的设计方法。熟悉了实验室实验箱和所用的软件。理解了加法器的原理和设计两个相同符号的数相加会导致溢出,明白了 verilog 语言的电路设计比较难。为后续设计 cpu 的实验打下基础。明白了各个文件的作用以及如何去插入调用，如何去上板分析，如何在仿真波形图上进行演算。  在进一步熟悉试验箱与计算机的联合使用的基础之上，也学会了在建立的测试平台上进行功能仿真，以此验证功能的正确性。目前只是会使用老师给出的代码，还欠缺自己动手设计代码和设计结构框图的能力，在后续试验中需要巩固加强。 |

|  |  |
| --- | --- |
| 实验二 数据运算：定点乘法一、实验分析说明 1.理解定点乘法的不同实现算法的原理，掌握基本实现算法 。  2.熟悉并运用 verilog 语言进行电路设计 。  3.为后续设计 cpu 的实验打下基础 。 二、实验设计思路 假设现在有两个32位带符号定点整数x和y，我们现在要让x和y相乘，然后把乘积存放在z中，大家知道，两个32位数相乘，结果不会超过64位，因此z的长度应该为64位。  z = x \* y中，x是被乘数，在Verilog代码中 multiplicand表示，y是乘数，在代码中用multiplier表示。因为x和y都是带符号数，所以应该是用补码乘法，但是如果对x和y求绝对值，让两个绝对值相乘，然后再判断正负，效果和补码乘法是相同。后面给出的Verilog代码就是基于这种思路编写的。两个32位整数相乘，实际上是进行了32次加法操作。下面以两个4位二进制数相乘来说明乘法实现的过程。    从上图中可以看到，被乘数x为1000，乘数y为1001，上面的乘法过程是手工运算的一个步骤，而计算机在做乘法时就是模拟上述手工运算的执行过程。因为是两个4位数相乘，所以结果应该是四个数加和得到的。先判断y的最低位是0还是1，如果是1，则需要把x加到部分积上，若为0，则需要把0加到部分积上（实际上加0的这个过程计算机并不执行，因为加0对部分积没有任何影响），x左移一位，之后再让y右移一位，若y为0，则循环结束，否则继续此循环过程。    1.学习并理解计算机中定点乘法器的多种实现算法的原理，重点掌握迭代乘法的实现算法 。  2.自行设计本次实验的方案，画出结构框图，详细标出输入输出端口，本次实验的乘法器建议采用迭代的方式实现，如果能力有余的，也可以采用其他效率更高的算法实现 。本次实验要求实现的乘法为有符号乘法，因此需要注意计算机存储的有符号数都是补码的形式，设计方案传递进来的数也需是补码 。  3.根据设计的实验方案，使用 verilog 编写相应代码 。  4.对编写的代码进行仿真，得到正确的波形图 。  5.将以上设计作为一个单独的模块，设计一个外围模块去调用该模块，外围模块中需调用封装好的 LCD 触摸屏模块，显示两个乘数和乘法结果，且需要利用触摸功能输入两个乘数 。 三、实验源代码multiply.v  |  | | --- | | `timescale 1ns / 1ps  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //   > 文件名: multiply.v  //   > 描述  ：乘法器模块，低效率的迭代乘法算法，使用两个乘数绝对值参与运算  //   > 作者  : 崔金泽 苗春雨 李子怡  //   > 日期  : 2022-10-04  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  module multiply(              // 乘法器      input         clk,        // 时钟      input         mult\_begin, // 乘法开始信号      input  [31:0] mult\_op1,   // 乘法源操作数1      input  [31:0] mult\_op2,   // 乘法源操作数2      output [63:0] product,    // 乘积      output        mult\_end    // 乘法结束信号  );      //乘法正在运算信号和结束信号      reg mult\_valid;      assign mult\_end = mult\_valid & ~(|multiplier); //乘法结束信号：乘数全0      always @(posedge clk)      begin          if (!mult\_begin || mult\_end)          begin              mult\_valid <= 1'b0;          end          else          begin              mult\_valid <= 1'b1;          end      end      //两个源操作取绝对值，正数的绝对值为其本身，负数的绝对值为取反加1      wire        op1\_sign;      //操作数1的符号位      wire        op2\_sign;      //操作数2的符号位      wire [31:0] op1\_absolute;  //操作数1的绝对值      wire [31:0] op2\_absolute;  //操作数2的绝对值      assign op1\_sign = mult\_op1[31];      assign op2\_sign = mult\_op2[31];      assign op1\_absolute = op1\_sign ? (~mult\_op1+1) : mult\_op1;      assign op2\_absolute = op2\_sign ? (~mult\_op2+1) : mult\_op2;      //加载被乘数，运算时每次左移一位      reg  [63:0] multiplicand;      always @ (posedge clk)      begin          if (mult\_valid)          begin    // 如果正在进行乘法，则被乘数每时钟左移一位              multiplicand <= {multiplicand[62:0],1'b0};          end          else if (mult\_begin)          begin   // 乘法开始，加载被乘数，为乘数1的绝对值              multiplicand <= {32'd0,op1\_absolute};          end      end      //加载乘数，运算时每次右移一位      reg  [31:0] multiplier;      always @ (posedge clk)      begin          if (mult\_valid)          begin   // 如果正在进行乘法，则乘数每时钟右移一位              multiplier <= {1'b0,multiplier[31:1]};          end          else if (mult\_begin)          begin   // 乘法开始，加载乘数，为乘数2的绝对值              multiplier <= op2\_absolute;          end      end        // 部分积：乘数末位为1，由被乘数左移得到；乘数末位为0，部分积为0      wire [63:0] partial\_product;      assign partial\_product = multiplier[0] ? multiplicand : 64'd0;        //累加器      reg [63:0] product\_temp;      always @ (posedge clk)      begin          if (mult\_valid)          begin              product\_temp <= product\_temp + partial\_product;          end          else if (mult\_begin)          begin              product\_temp <= 64'd0;  // 乘法开始，乘积清零          end      end        //乘法结果的符号位和乘法结果      reg product\_sign;      always @ (posedge clk)  // 乘积      begin          if (mult\_valid)          begin                product\_sign <= op1\_sign ^ op2\_sign;          end      end      //若乘法结果为负数，则需要对结果取反+1      assign product = product\_sign ? (~product\_temp+1) : product\_temp;  endmodule |  四、实验实现截图    五、收获心得体会 迭代乘法是在模拟我们人算乘法的过程，乘数每次右移一位，根据最低位，判断 是加被乘数移位后的值还是加 0，不停地累加最后就得到乘积了 。可以看到迭代乘法 是用多次加法完成乘法操作的，故需要多拍时间，其结束标志为乘数移位后为 0，故 对于 32 位乘法，最多需要 32 拍才能完成一次乘法。在本次实验中我学习并理解了计算机中定点乘法器的多种实现算法的原理，重点掌握迭代乘法的实现算法 。  程序仿真开始时，bestbench.v文件会对输入信号进行初始化。使得mult\_begin为1，并且给出两个操作数mult\_op1和mult\_op2分别作为乘数和被乘数。时钟信号clk每5ns变化一次，也就是说五个always块每隔10ns被触发一次。对mult\_op1和mult\_op2进行分解，分解出他们的符号和绝对值，后面的运算是让mult\_op1和mult\_op2的绝对值进行运算，相当于是两个无符号数的乘法。当乘法信号有效后，也就是说乘法开始之后，把x的绝对值赋值给一个64位的reg型变量multiplicand，把y的绝对值赋值给一个32位reg型变量multiplier，根绝multiplier最低位是0还是1，决定着64位wire型变量partial\_product赋值0还是赋值multiplicand。临时结果product\_temp加上部分积之后再把加的结果赋值给自己，根据mult\_op1和mult\_op2的符号计算乘积结果的符号。最终的乘积结果（product）是wire型变量，用assign赋值，每当临时结果（product\_temp）发生改变时，product也立即发生变化。 |

|  |  |
| --- | --- |
| 实验三 寄存器堆实现一、实验分析说明 1.熟悉并掌握 MIPS 计算机中寄存器堆的原理和设计方法 。  2.初步了解 MIPS 指令结构和源操作数/目的操作数的概念 。  3.熟悉并运用 verilog 语言进行电路设计 。  4.为后续设计 cpu 的实验打下基础 。 二、实验设计思路 这次要做的是用Verilog代码写一个寄存器堆，此寄存器堆共有32个寄存器，每个寄存器可存储32个二进制位。要求有一个写端口，两个读端口，本次实验设计为异步读同步写的寄存器堆，即读寄存器不需要时钟控制，但写寄存器需时钟控制。  整个代码的逻辑还是比较简单的。定义一个reg型数组REG\_Files来充当寄存器堆，此数组共有32个元素，每一个元素的大小为32个二进制位。在initial块中，用for循环对寄存器堆的内容初始化为0。当时钟信号clk上跳沿时触发always语句的执行，如果写使能信号wen为1，则把数据写入寄存器堆中。因为是异步读，所以只要是输入寄存器的地址，应能够立刻得到寄存器的内容。用assign语句对读数据的输出端口rdata1和rdata2进行赋值，其中，数组的下标相当于寄存器的地址，因此可以写成REG\_Files[raddr1] 这种形式。  1.学习MIPS 计算机中寄存器堆的设计及原理，如 ：有多少个寄存器，有无特殊设置的寄存器，mips 指令如何去索引寄存器的等 。  2.自行设计本次实验的方案，画出结构框图，详细标出输入输出端口，本次实验建议设计为异步读同步写的寄存器堆，即读寄存器不需要时钟控制，但写寄存器需时钟控制 。  3 .寄存器堆设计为 1 个写端口和 2 个读端口，后续 CPU 实验用到的寄存器堆需要 1 个写端口和 2 个读端口 。  4.根据设计的实验方案，使用 verilog 编写相应代码 。  5.对编写的代码进行仿真，得到正确的波形图 。  6.将以上设计作为一个单独的模块，设计一个外围模块去调用该模块。外围模块中需调用封装好的 LCD 触摸屏模块，显示寄存器堆的读写端口地址和数  据，最好能扫描出所有寄存器的值显示在 LCD 触摸屏上，并且需要利用触摸功能输入寄存器堆的读写地址和写数据 。  7.将编写的代码进行综合布局布线，并下载到实验箱中的 FPGA 板子上进行演示。    regfile模块结构框图 三、实验源代码regfile.v  |  | | --- | | `timescale 1ns / 1ps  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //   > 文件名: regfile.v  //   > 描述  ：寄存器堆模块，同步写，异步读  //   > 作者  : 崔金泽 苗春雨 李子怡  //   > 日期  : 2022-10-05  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  module regfile(      input             clk,      input             wen,      input      [4 :0] raddr1,      input      [4 :0] raddr2,      input      [4 :0] waddr,      input      [31:0] wdata,      output reg [31:0] rdata1,      output reg [31:0] rdata2,      input      [4 :0] test\_addr,      output reg [31:0] test\_data      );      reg [31:0] rf[31:0];        // three ported register file      // read two ports combinationally      // write third port on rising edge of clock      // register 0 hardwired to 0      always @(posedge clk)      begin          if (wen)          begin              rf[waddr] <= wdata;          end      end        //读端口1      always @(\*)      begin          case (raddr1)              5'd1 : rdata1 <= rf[1 ];              5'd2 : rdata1 <= rf[2 ];              5'd3 : rdata1 <= rf[3 ];              5'd4 : rdata1 <= rf[4 ];              5'd5 : rdata1 <= rf[5 ];              5'd6 : rdata1 <= rf[6 ];              5'd7 : rdata1 <= rf[7 ];              5'd8 : rdata1 <= rf[8 ];              5'd9 : rdata1 <= rf[9 ];              5'd10: rdata1 <= rf[10];              5'd11: rdata1 <= rf[11];              5'd12: rdata1 <= rf[12];              5'd13: rdata1 <= rf[13];              5'd14: rdata1 <= rf[14];              5'd15: rdata1 <= rf[15];              5'd16: rdata1 <= rf[16];              5'd17: rdata1 <= rf[17];              5'd18: rdata1 <= rf[18];              5'd19: rdata1 <= rf[19];              5'd20: rdata1 <= rf[20];              5'd21: rdata1 <= rf[21];              5'd22: rdata1 <= rf[22];              5'd23: rdata1 <= rf[23];              5'd24: rdata1 <= rf[24];              5'd25: rdata1 <= rf[25];              5'd26: rdata1 <= rf[26];              5'd27: rdata1 <= rf[27];              5'd28: rdata1 <= rf[28];              5'd29: rdata1 <= rf[29];              5'd30: rdata1 <= rf[30];              5'd31: rdata1 <= rf[31];              default : rdata1 <= 32'd0;          endcase      end      //读端口2      always @(\*)      begin          case (raddr2)              5'd1 : rdata2 <= rf[1 ];              5'd2 : rdata2 <= rf[2 ];              5'd3 : rdata2 <= rf[3 ];              5'd4 : rdata2 <= rf[4 ];              5'd5 : rdata2 <= rf[5 ];              5'd6 : rdata2 <= rf[6 ];              5'd7 : rdata2 <= rf[7 ];              5'd8 : rdata2 <= rf[8 ];              5'd9 : rdata2 <= rf[9 ];              5'd10: rdata2 <= rf[10];              5'd11: rdata2 <= rf[11];              5'd12: rdata2 <= rf[12];              5'd13: rdata2 <= rf[13];              5'd14: rdata2 <= rf[14];              5'd15: rdata2 <= rf[15];              5'd16: rdata2 <= rf[16];              5'd17: rdata2 <= rf[17];              5'd18: rdata2 <= rf[18];              5'd19: rdata2 <= rf[19];              5'd20: rdata2 <= rf[20];              5'd21: rdata2 <= rf[21];              5'd22: rdata2 <= rf[22];              5'd23: rdata2 <= rf[23];              5'd24: rdata2 <= rf[24];              5'd25: rdata2 <= rf[25];              5'd26: rdata2 <= rf[26];              5'd27: rdata2 <= rf[27];              5'd28: rdata2 <= rf[28];              5'd29: rdata2 <= rf[29];              5'd30: rdata2 <= rf[30];              5'd31: rdata2 <= rf[31];              default : rdata2 <= 32'd0;          endcase      end       //调试端口，读出寄存器值显示在触摸屏上      always @(\*)      begin          case (test\_addr)              5'd1 : test\_data <= rf[1 ];              5'd2 : test\_data <= rf[2 ];              5'd3 : test\_data <= rf[3 ];              5'd4 : test\_data <= rf[4 ];              5'd5 : test\_data <= rf[5 ];              5'd6 : test\_data <= rf[6 ];              5'd7 : test\_data <= rf[7 ];              5'd8 : test\_data <= rf[8 ];              5'd9 : test\_data <= rf[9 ];              5'd10: test\_data <= rf[10];              5'd11: test\_data <= rf[11];              5'd12: test\_data <= rf[12];              5'd13: test\_data <= rf[13];              5'd14: test\_data <= rf[14];              5'd15: test\_data <= rf[15];              5'd16: test\_data <= rf[16];              5'd17: test\_data <= rf[17];              5'd18: test\_data <= rf[18];              5'd19: test\_data <= rf[19];              5'd20: test\_data <= rf[20];              5'd21: test\_data <= rf[21];              5'd22: test\_data <= rf[22];              5'd23: test\_data <= rf[23];              5'd24: test\_data <= rf[24];              5'd25: test\_data <= rf[25];              5'd26: test\_data <= rf[26];              5'd27: test\_data <= rf[27];              5'd28: test\_data <= rf[28];              5'd29: test\_data <= rf[29];              5'd30: test\_data <= rf[30];              5'd31: test\_data <= rf[31];              default : test\_data <= 32'd0;          endcase      end  endmodule |  四、实验实现截图    五、收获心得体会 在本次实验中我熟悉并掌握了 MIPS 计算机中寄存器堆的原理和设计方法 。初步了解 MIPS 指令结构和源操作数/目的操作数的概念 。 寄存器的功能是存储二进制代码，它是由具有存储功能的触发器组合起来构成的。一个触发器可以存储1位二进制代码，故存放n位二进制代码的寄存器，需用n个触发器来构成。  一开始我的实验中的寄存器堆实验中无法在显示屏上显示寄存器内容，因为regfile模块的输入输出端口是提前写好的，我需要做的是把寄存器堆的逻辑功能实现。有一个输入端口test\_addr和一个输出端口test\_data我没有用到，因为我在补充代码时以为这两个端口是仿真时用的，所以在上板时就没有管它。用上面的代码来上板验证，寄存器的内容始终为0，感觉代码的逻辑也没有问题，而且也能对寄存器进行正确的读操作，对此问题感觉百思不得其解。之后我发现在regfile.v的代码中，用一个含有32个32位长度的数组来表示寄存器，但这个数组的名称是自己定义的，每个人在补充代码时定义的数组名称可能会不同，触摸屏总不能智能到辨识出哪个是你定义的寄存器。因此初步分析应该是没有给触摸屏传数据，所以才导致触摸屏上无法显示寄存器的内容。  我打开了regfile\_display.v文件，查看寄存器堆的显示模块。此模块代码有二百多行，虽然有些看不懂的地方，但是能够理解个大概。编号为7~38的显示块用来显示32个寄存器的值。有一个赋值语句， display\_value <= test\_data; 虽然不是很明白这行代码什么意思，但我猜想display\_value是要显示到屏幕上的值。test\_addr的值等于display\_number - 7。display\_number是显示屏上块的编号，因为7~38号是显示寄存器内容的块，所以display\_number - 7的范围就是0~31，这恰好是寄存器地址的范围。又由于是assign赋值语句，所以display\_number的变化会立刻反映到test\_addr上。我没能从代码中找到display\_number是如何变化的，但由于它是用module模块来实例化的，所以我猜想它的值一定会发生变化，可能是会从0开始，每次自加。  以display\_number等于25为例来解释一下这两行代码，此时test\_addr等于18（表示现在是第18个寄存器），换算成二进制就是1 0010。最终diaplay\_name的十六位是 0011 0001 0011 0010，它对应着两个字符，这两个字符分别是’1’和’2’，那么display\_name所对应的字符串就是’‘REG12’’，它表示这是寄存器12h，能够与test\_addr等于18（这里的18是十进制数）对应起来。 |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 实验四 ALU模块实现一、实验分析说明 1.熟悉 MIPS 指令集中的运算指令，学会对这些指令进行归纳分类 。  2.了解 MIPS 指令结构 。  3.熟悉并掌握 ALU 的原理 、功能和设计 。  4.进一步加强运用 verilog 语言进行电路设计的能力 。  5.掌握算术逻辑运算单元( ALU)的工作原理。  6.熟悉简单运算器的数据传送通路。  7.验证16位运算器（用4位74181级联而成）的组合功能。  8.按给定数据，完成几种指定的算术和逻辑运算。 二、实验设计思路 用Verilog代码实现以下12个ALU功能  加法  减法  有符号比较，小于置位  无符号比较，小于置位  按位与  按位或非  按位或  按位异或  逻辑左移  逻辑右移  算术右移  高位加载  8位字长的ALU由2片74181构成。2片 74273构成两个操作数寄存器 DRl和 DR2，用来保存参与运算的数据，这样可以构成一个8位的ALU，再将两个这样的ALU级联便可构成一个16位的ALU。DRl接 ALU的A低八位数据输入端口，DR2接ALU的B低八位数据输入端口，DR3接A高八位数据输入端口，DR4接B高八位数据输入端口。最后计算结果由四片74181的四位输出端组成即FO-F15。  1.学习MIPS 指令集，熟知指令类型，了解指令功能和编码，归纳基础的ALU 运 算指令 。  2.归纳确定自己本次实验中准备实现的 ALU 运算，要求不实现定点乘除指令和浮 点运算指令，要求至少实现 5 种 ALU 运算，其中要包含加减运算，其中减法在内部要 转换为加法，与加法运算共同调用实验一里自己完成的加法模块去做 。  3.自行设计本次实验的方案，画出结构框图，大致结构框图如图 。中 的操作码位数和类型请自行设计，可以设计为独热码（一位有效编码）或二进制编 码 。 比如，设计方案中预定实现 7 种 ALU 运算，则操作码采用独热码，则需 7bit 数 据，每位单独指示一种运算；若采用二进制编码，则只用 3bit 数据位即可，但在需 ALU 内部先进行解码，才能确定 ALU 作何种运算。 根据设计的实验方案，使用 verilog 编写相应代码 。  4. 对编写的代码进行仿真，得到正确的波形图 。  5.将以上设计作为一个单独的模块，设计一个外围模块去调用该模块，。外围模块中需调用封装好的 LCD 触摸屏模块，显示 ALU 的两个源操作数 、操作码和运算结果，并且需要利用触摸功能输入源操作数 。操作码可以考虑用LCD 触摸屏输 入，也可以用拨码开关输入 。  6.将编写的代码进行综合布局布线，并下载到试验箱中的 FPGA 板子上进行演 示 。    功能表   |  |  |  | | --- | --- | --- | | ALU\_OP[3:0] | ALU功能 | 功能说明 | | 0000 | and | 按位与运算 | | 0001 | or | 按位或运算 | | 0010 | xor | 按位异或运算 | | 0011 | nor | 按位或非运算 | | 0100 | add | 算术加运算 | | 0101 | sub | 算术减运算 | | 0110 | slt | 若A<B，则输出1;否则输出0 | | 0111 | sll | B逻辑左移A所指定的位数 |  三、实验源代码alu.v  |  | | --- | | `timescale 1ns / 1ps  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  // > 文件名: alu.v  // > 描述 ：ALU模块，可做12种操作  //   > 作者  : 崔金泽 苗春雨 李子怡  //   > 日期  : 2022-10-05  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  module alu(      input  [11:0] alu\_control,  // ALU控制信号      input  [31:0] alu\_src1,     // ALU操作数1,为补码      input  [31:0] alu\_src2,     // ALU操作数2，为补码      output [31:0] alu\_result    // ALU结果      );      // ALU控制信号，独热码      wire alu\_add;   //加法操作      wire alu\_sub;   //减法操作      wire alu\_slt;   //有符号比较，小于置位，复用加法器做减法      wire alu\_sltu;  //无符号比较，小于置位，复用加法器做减法      wire alu\_and;   //按位与      wire alu\_nor;   //按位或非      wire alu\_or;    //按位或      wire alu\_xor;   //按位异或      wire alu\_sll;   //逻辑左移      wire alu\_srl;   //逻辑右移      wire alu\_sra;   //算术右移      wire alu\_lui;   //高位加载      assign alu\_add  = alu\_control[11];      assign alu\_sub  = alu\_control[10];      assign alu\_slt  = alu\_control[ 9];      assign alu\_sltu = alu\_control[ 8];      assign alu\_and  = alu\_control[ 7];      assign alu\_nor  = alu\_control[ 6];      assign alu\_or   = alu\_control[ 5];      assign alu\_xor  = alu\_control[ 4];      assign alu\_sll  = alu\_control[ 3];      assign alu\_srl  = alu\_control[ 2];      assign alu\_sra  = alu\_control[ 1];      assign alu\_lui  = alu\_control[ 0];      wire [31:0] add\_sub\_result;      wire [31:0] slt\_result;      wire [31:0] sltu\_result;      wire [31:0] and\_result;      wire [31:0] nor\_result;      wire [31:0] or\_result;      wire [31:0] xor\_result;      wire [31:0] sll\_result;      wire [31:0] srl\_result;      wire [31:0] sra\_result;      wire [31:0] lui\_result;      assign and\_result = alu\_src1 & alu\_src2;      // 与结果为两数按位与      assign or\_result  = alu\_src1 | alu\_src2;      // 或结果为两数按位或      assign nor\_result = ~or\_result;               // 或非结果为或结果按位取反      assign xor\_result = alu\_src1 ^ alu\_src2;      // 异或结果为两数按位异或      assign lui\_result = {alu\_src2[15:0], 16'd0};  // 立即数装载结果为立即数移位至高半字节  //-----{加法器}begin  //add,sub,slt,sltu均使用该模块      wire [31:0] adder\_operand1;      wire [31:0] adder\_operand2;      wire        adder\_cin     ;      wire [31:0] adder\_result  ;      wire        adder\_cout    ;      assign adder\_operand1 = alu\_src1;      assign adder\_operand2 = alu\_add ? alu\_src2 : ~alu\_src2;      assign adder\_cin      = ~alu\_add; //减法需要cin      adder adder\_module(      .operand1(adder\_operand1),      .operand2(adder\_operand2),      .cin     (adder\_cin     ),      .result  (adder\_result  ),      .cout    (adder\_cout    )      );      //加减结果      assign add\_sub\_result = adder\_result;      //slt结果      //adder\_src1[31] adder\_src2[31] adder\_result[31]      //       0             1           X(0或1)       "正-负"，显然小于不成立      //       0             0             1           相减为负，说明小于      //       0             0             0           相减为正，说明不小于      //       1             1             1           相减为负，说明小于      //       1             1             0           相减为正，说明不小于      //       1             0           X(0或1)       "负-正"，显然小于成立      assign slt\_result[31:1] = 31'd0;      assign slt\_result[0]    = (alu\_src1[31] & ~alu\_src2[31]) | (~(alu\_src1[31]^alu\_src2[31]) & adder\_result[31]);      //sltu结果      //对于32位无符号数比较，相当于33位有符号数（{1'b0,src1}和{1'b0,src2}）的比较，最高位0为符号位      //故，可以用33位加法器来比较大小，需要对{1'b0,src2}取反,即需要{1'b0,src1}+{1'b1,~src2}+cin      //但此处用的为32位加法器，只做了运算:                             src1   +    ~src2   +cin      //32位加法的结果为{adder\_cout,adder\_result},则33位加法结果应该为{adder\_cout+1'b1,adder\_result}      //对比slt结果注释，知道，此时判断大小属于第二三种情况，即源操作数1符号位为0，源操作数2符号位为0      //结果的符号位为1，说明小于，即adder\_cout+1'b1为2'b01，即adder\_cout为0      assign sltu\_result = {31'd0, ~adder\_cout};  //-----{加法器}end  //-----{移位器}begin      // 移位分三步进行，      // 第一步根据移位量低2位即[1:0]位做第一次移位，      // 第二步在第一次移位基础上根据移位量[3:2]位做第二次移位，      // 第三步在第二次移位基础上根据移位量[4]位做第三次移位。      wire [4:0] shf;      assign shf = alu\_src1[4:0];      wire [1:0] shf\_1\_0;      wire [1:0] shf\_3\_2;      assign shf\_1\_0 = shf[1:0];      assign shf\_3\_2 = shf[3:2];         // 逻辑左移      wire [31:0] sll\_step1;      wire [31:0] sll\_step2;      assign sll\_step1 = {32{shf\_1\_0 == 2'b00}} & alu\_src2                   // 若shf[1:0]="00",不移位                       | {32{shf\_1\_0 == 2'b01}} & {alu\_src2[30:0], 1'd0}     // 若shf[1:0]="01",左移1位                       | {32{shf\_1\_0 == 2'b10}} & {alu\_src2[29:0], 2'd0}     // 若shf[1:0]="10",左移2位                       | {32{shf\_1\_0 == 2'b11}} & {alu\_src2[28:0], 3'd0};    // 若shf[1:0]="11",左移3位      assign sll\_step2 = {32{shf\_3\_2 == 2'b00}} & sll\_step1                  // 若shf[3:2]="00",不移位                       | {32{shf\_3\_2 == 2'b01}} & {sll\_step1[27:0], 4'd0}    // 若shf[3:2]="01",第一次移位结果左移4位                       | {32{shf\_3\_2 == 2'b10}} & {sll\_step1[23:0], 8'd0}    // 若shf[3:2]="10",第一次移位结果左移8位                       | {32{shf\_3\_2 == 2'b11}} & {sll\_step1[19:0], 12'd0};  // 若shf[3:2]="11",第一次移位结果左移12位      assign sll\_result = shf[4] ? {sll\_step2[15:0], 16'd0} : sll\_step2;     // 若shf[4]="1",第二次移位结果左移16位      // 逻辑右移      wire [31:0] srl\_step1;      wire [31:0] srl\_step2;      assign srl\_step1 = {32{shf\_1\_0 == 2'b00}} & alu\_src2                   // 若shf[1:0]="00",不移位                       | {32{shf\_1\_0 == 2'b01}} & {1'd0, alu\_src2[31:1]}     // 若shf[1:0]="01",右移1位,高位补0                       | {32{shf\_1\_0 == 2'b10}} & {2'd0, alu\_src2[31:2]}     // 若shf[1:0]="10",右移2位,高位补0                       | {32{shf\_1\_0 == 2'b11}} & {3'd0, alu\_src2[31:3]};    // 若shf[1:0]="11",右移3位,高位补0      assign srl\_step2 = {32{shf\_3\_2 == 2'b00}} & srl\_step1                  // 若shf[3:2]="00",不移位                       | {32{shf\_3\_2 == 2'b01}} & {4'd0, srl\_step1[31:4]}    // 若shf[3:2]="01",第一次移位结果右移4位,高位补0                       | {32{shf\_3\_2 == 2'b10}} & {8'd0, srl\_step1[31:8]}    // 若shf[3:2]="10",第一次移位结果右移8位,高位补0                       | {32{shf\_3\_2 == 2'b11}} & {12'd0, srl\_step1[31:12]}; // 若shf[3:2]="11",第一次移位结果右移12位,高位补0      assign srl\_result = shf[4] ? {16'd0, srl\_step2[31:16]} : srl\_step2;    // 若shf[4]="1",第二次移位结果右移16位,高位补0        // 算术右移      wire [31:0] sra\_step1;      wire [31:0] sra\_step2;      assign sra\_step1 = {32{shf\_1\_0 == 2'b00}} & alu\_src2                                 // 若shf[1:0]="00",不移位                       | {32{shf\_1\_0 == 2'b01}} & {alu\_src2[31], alu\_src2[31:1]}           // 若shf[1:0]="01",右移1位,高位补符号位                       | {32{shf\_1\_0 == 2'b10}} & {{2{alu\_src2[31]}}, alu\_src2[31:2]}      // 若shf[1:0]="10",右移2位,高位补符号位                       | {32{shf\_1\_0 == 2'b11}} & {{3{alu\_src2[31]}}, alu\_src2[31:3]};     // 若shf[1:0]="11",右移3位,高位补符号位      assign sra\_step2 = {32{shf\_3\_2 == 2'b00}} & sra\_step1                                // 若shf[3:2]="00",不移位                       | {32{shf\_3\_2 == 2'b01}} & {{4{sra\_step1[31]}}, sra\_step1[31:4]}    // 若shf[3:2]="01",第一次移位结果右移4位,高位补符号位                       | {32{shf\_3\_2 == 2'b10}} & {{8{sra\_step1[31]}}, sra\_step1[31:8]}    // 若shf[3:2]="10",第一次移位结果右移8位,高位补符号位                       | {32{shf\_3\_2 == 2'b11}} & {{12{sra\_step1[31]}}, sra\_step1[31:12]}; // 若shf[3:2]="11",第一次移位结果右移12位,高位补符号位      assign sra\_result = shf[4] ? {{16{sra\_step2[31]}}, sra\_step2[31:16]} : sra\_step2;    // 若shf[4]="1",第二次移位结果右移16位,高位补符号位  //-----{移位器}end      // 选择相应结果输出      assign alu\_result = (alu\_add|alu\_sub) ? add\_sub\_result[31:0] :                          alu\_slt           ? slt\_result :                          alu\_sltu          ? sltu\_result :                          alu\_and           ? and\_result :                          alu\_nor           ? nor\_result :                          alu\_or            ? or\_result  :                          alu\_xor           ? xor\_result :                          alu\_sll           ? sll\_result :                          alu\_srl           ? srl\_result :                          alu\_sra           ? sra\_result :                          alu\_lui           ? lui\_result :                          32'd0;  endmodule |  四、实验实现截图   加法 减法   按位与  异或  按位或  或非  逻辑左移 五、收获心得体会 在本次试验中，我学会了ALU的原理与功能，算术逻辑单元（Arithmetic&logical Unit）是中央处理器(CPU)的执行单元，是所有中央处理器的核心组成部分，由"And Gate"（与门） 和"Or Gate"（或门）构成的算术逻辑单元，主要功能是进行二位元的算术运算，如加减乘(不包括整数除法)。基本上，在所有现代CPU体系结构中，二进制都以补码的形式来表示。总的来说实验四不是很难，书上的要求是做OP[2:0]三位8中运算，标志位也没有那么多。因为上次做龙芯实验是1年半之前了，很多Verilog的语法都忘记了，但是通过磕磕碰碰的和组员讨论，慢慢debug最后还是完成了实验。本次实验暴露出了一些自身存在的问题，对一些复用器功能的不熟悉，设计逻辑电路的方法与步骤不够科学，只会生搬硬套。但通过本次实验，我理解了ALU内部的原理，并且对于部分复用器的使用更加熟练，优化了逻辑电路的设计的方法。本次实验使我收益匪浅。 |

|  |  |  |
| --- | --- | --- |
| 实验五 存储器实现一、实验分析说明 1.了解只读存储器 ROM 和随机存取存储器 RAM 的原理 。  2.理解 ROM 读取数据及 RAM 读取 、写入数据的过程 。  3.理解计算机中存储器地址编址和数据索引方法 。  4.理解同步 RAM 和异步 RAM 的区别 。  5.掌握调用 xilinx 库 IP 实例化 RAM 的设计方法 。  6.熟悉并运用 verilog 语言进行电路设计 。  7.为后续设计 cpu 的实验打下基础 。 二、实验设计思路 CPU中的重要部件，如RAM、ROM可直接调用他们构成，因此在FPGA中利用嵌入式阵列块EAB可以构成各种结构的存储器，ROM是其中的一种。由于ROM是只读存储器，所以它的数据口是单向的输出端口，ROM中的数据是在对FPGA现场配置时，通过配置文件一起写入存储单元的。  本次实验使用Vivado的Block Memory Generator模拟数据在存储器中的存取过程。实验使用单端口ROM。初始化ROM存储器中的内容，通过开关选择相应的地址,将对应的存储器中内容读出来，并通过七段数码管显示。实验原理如图所示:    1.学习存储器的设计及原理，如 ：ROM 读地址索引读取数据过程及时序，RAM 读 写时序，同步和异步的区别等 。  2.学习计算机中内存地址编址和数据索引方法 。  3.自行设计本次实验的方案，画出结构框图，详细标出输入输出端口，确定存储 器宽度 、深度和写使能位数 。  4.学习 ISE 工具中调用库 IP 的方法 。  5.本次实验要求调用 xilinx 库 IP 实例化一块 RAM 。实例化的 RAM 选择为同步 RAM 。本次实验的 RAM 建议设置为两个端口，一个端口用来正常的读写，另一个端口作为调试端口只使用读功能用于观察存储器内部数据 。  6.调用 xilinx 库 IP 实例化一块 RAM，并进行仿真，得到正确的波形图 。7.将以上设计作为一个单独的模块，设计一个外围模块去调用该模块。外围模块中需调用封装好的 LCD 触摸屏模块，显示 RAM 的正常端口的地址 、待写 入的数据和读出的数据，显示调试端口的地址和读出的数据 。并且需要利用触摸功能 输入正常端口的地址和写数据，以及调试端口的地址。8.将编写的代码进行综合布局布线，并下载到实验箱中的 FPGA 板子上进行演示。  存储器深度不要过大，避免耗费过多的 FPGA 上的资源 。本次实验要求实现 同步的存储器 。而异步存储器的搭建方法同寄存器堆的搭建，但不同的是，寄存器堆 中读写端口是分开的，但对于异步 RAM 要求读写共用一个端口，只是会增加一个写使 能信号 。可以自行尝试搭建异步的ROM 和 RAM，在单周期 CPU 实验中会用到异步的 ROM 作为指令存储器，而异步 RAM 作为数据存储器 。 三、实验源代码data\_ram\_display.v  |  | | --- | | `timescale 1ns / 1ps  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //   > 文件名: data\_ram\_display.v  //   > 描述  ：数据存储器模块显示模块，调用FPGA板上的IO接口和触摸屏  //   > 作者  : 崔金泽 苗春雨 李子怡  //   > 日期  : 2022-10-05  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  module data\_ram\_display(      //时钟与复位信号      input clk,      input resetn,    //后缀"n"代表低电平有效      //拨码开关，用于产生写使能和选择输入数      input [3:0] wen,      input [1:0] input\_sel,      //led灯，用于指示写使能信号，和正在输入什么数据      output [3:0] led\_wen,      output led\_addr,      //指示输入读写地址      output led\_wdata,     //指示输入写数据      output led\_test\_addr, //指示输入test地址      //触摸屏相关接口，不需要更改      output lcd\_rst,      output lcd\_cs,      output lcd\_rs,      output lcd\_wr,      output lcd\_rd,      inout[15:0] lcd\_data\_io,      output lcd\_bl\_ctr,      inout ct\_int,      inout ct\_sda,      output ct\_scl,      output ct\_rstn      );  //-----{LED显示}begin      assign led\_wen       = wen;      assign led\_addr      = (input\_sel==2'd0);      assign led\_wdata     = (input\_sel==2'd1);      assign led\_test\_addr = (input\_sel==2'd2);  //-----{LED显示}end  //-----{调用数据储存器模块}begin      //数据存储器多增加一个读端口，用于读出特定内存地址显示在触摸屏上      reg  [31:0] addr;      reg  [31:0] wdata;      wire [31:0] rdata;      reg  [31:0] test\_addr;      wire [31:0] test\_data;      data\_ram data\_ram\_module(          .clka  (clk           ),          .wea   (wen           ),          .addra (addr[9:2]     ),          .dina  (wdata         ),          .douta (rdata         ),          .clkb  (clk           ),          .web   (4'd0          ),          .addrb (test\_addr[9:2]),          .doutb (test\_data     ),          .dinb  (32'd0         )      );  //-----{调用寄存器堆模块}end  //---------------------{调用触摸屏模块}begin--------------------//  //-----{实例化触摸屏}begin  //此小节不需要更改      reg         display\_valid;      reg  [39:0] display\_name;      reg  [31:0] display\_value;      wire [5 :0] display\_number;      wire        input\_valid;      wire [31:0] input\_value;      lcd\_module lcd\_module(          .clk            (clk           ),   //10Mhz          .resetn         (resetn        ),          //调用触摸屏的接口          .display\_valid  (display\_valid ),          .display\_name   (display\_name  ),          .display\_value  (display\_value ),          .display\_number (display\_number),          .input\_valid    (input\_valid   ),          .input\_value    (input\_value   ),          //lcd触摸屏相关接口，不需要更改          .lcd\_rst        (lcd\_rst       ),          .lcd\_cs         (lcd\_cs        ),          .lcd\_rs         (lcd\_rs        ),          .lcd\_wr         (lcd\_wr        ),          .lcd\_rd         (lcd\_rd        ),          .lcd\_data\_io    (lcd\_data\_io   ),          .lcd\_bl\_ctr     (lcd\_bl\_ctr    ),          .ct\_int         (ct\_int        ),          .ct\_sda         (ct\_sda        ),          .ct\_scl         (ct\_scl        ),          .ct\_rstn        (ct\_rstn       )      );  //-----{实例化触摸屏}end  //-----{从触摸屏获取输入}begin  //根据实际需要输入的数修改此小节，  //建议对每一个数的输入，编写单独一个always块      //当input\_sel为2'b00时，表示输入数为读写地址，即addr      always @(posedge clk)      begin          if (!resetn)          begin              addr <= 32'd0;          end          else if (input\_valid &&  input\_sel==2'd0)          begin              addr[31:2] <= input\_value[31:2];          end      end        //当input\_sel为2'b01时，表示输入数为写数据，即wdata      always @(posedge clk)      begin          if (!resetn)          begin              wdata <= 32'd0;          end          else if (input\_valid && input\_sel==2'd1)          begin              wdata <= input\_value;          end      end        //当input\_sel为2'b10时，表示输入数为test地址，即test\_addr      always @(posedge clk)      begin          if (!resetn)          begin              test\_addr  <= 32'd0;          end          else if (input\_valid && input\_sel==2'd2)          begin              test\_addr[31:2] <= input\_value[31:2];          end      end  //-----{从触摸屏获取输入}end  //-----{输出到触摸屏显示}begin  //根据需要显示的数修改此小节，  //触摸屏上共有44块显示区域，可显示44组32位数据  //44块显示区域从1开始编号，编号为1~44，      always @(posedge clk)      begin         case(display\_number)             6'd1:             begin                 display\_valid <= 1'b1;                 display\_name  <= "ADDR ";                 display\_value <= addr;             end             6'd2:             begin                 display\_valid <= 1'b1;                 display\_name  <= "WDATA";                 display\_value <= wdata;             end             6'd3:             begin                 display\_valid <= 1'b1;                 display\_name  <= "RDATA";                 display\_value <= rdata;             end             6'd5:             begin                 display\_valid <= 1'b1;                 display\_name  <= "T\_ADD";                 display\_value <= test\_addr;             end             6'd6:             begin                 display\_valid <= 1'b1;                 display\_name  <= "T\_DAT";                 display\_value <= test\_data;             end             default :             begin                 display\_valid <= 1'b0;                 display\_name  <= 40'd0;                 display\_value <= 32'd0;             end         endcase      end  //-----{输出到触摸屏显示}end  //----------------------{调用触摸屏模块}end---------------------//  endmodule |  inst\_rom\_display.v  |  | | --- | | `timescale 1ns / 1ps  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //   > 文件名: inst\_rom\_display.v  //   > 描述  ：异步指令存储器显示模块，调用FPGA板上的IO接口和触摸屏  //   > 作者  : 崔金泽 苗春雨 李子怡  //   > 日期  : 2022-10-05  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  module inst\_rom\_display(      //时钟与复位信号      input clk,      input resetn,    //后缀"n"代表低电平有效      //触摸屏相关接口，不需要更改      output lcd\_rst,      output lcd\_cs,      output lcd\_rs,      output lcd\_wr,      output lcd\_rd,      inout[15:0] lcd\_data\_io,      output lcd\_bl\_ctr,      inout ct\_int,      inout ct\_sda,      output ct\_scl,      output ct\_rstn      );  //-----{调用数据储存器模块}begin      //数据存储器多增加一个读端口，用于读出特定内存地址显示在触摸屏上      reg  [31:0] addr;      wire [31:0] inst;      inst\_rom inst\_rom\_module(          .clka   (clk       ),          .addra  (addr[9:2] ),          .douta  (inst      )      );  //-----{调用寄存器堆模块}end  //---------------------{调用触摸屏模块}begin--------------------//  //-----{实例化触摸屏}begin  //此小节不需要更改      reg         display\_valid;      reg  [39:0] display\_name;      reg  [31:0] display\_value;      wire [5 :0] display\_number;      wire        input\_valid;      wire [31:0] input\_value;      lcd\_module lcd\_module(          .clk            (clk           ),   //10Mhz          .resetn         (resetn        ),          //调用触摸屏的接口          .display\_valid  (display\_valid ),          .display\_name   (display\_name  ),          .display\_value  (display\_value ),          .display\_number (display\_number),          .input\_valid    (input\_valid   ),          .input\_value    (input\_value   ),          //lcd触摸屏相关接口，不需要更改          .lcd\_rst        (lcd\_rst       ),          .lcd\_cs         (lcd\_cs        ),          .lcd\_rs         (lcd\_rs        ),          .lcd\_wr         (lcd\_wr        ),          .lcd\_rd         (lcd\_rd        ),          .lcd\_data\_io    (lcd\_data\_io   ),          .lcd\_bl\_ctr     (lcd\_bl\_ctr    ),          .ct\_int         (ct\_int        ),          .ct\_sda         (ct\_sda        ),          .ct\_scl         (ct\_scl        ),          .ct\_rstn        (ct\_rstn       )      );  //-----{实例化触摸屏}end  //-----{从触摸屏获取输入}begin  //根据实际需要输入的数修改此小节，  //建议对每一个数的输入，编写单独一个always块      always @(posedge clk)      begin          if (!resetn)          begin              addr <= 32'd0;          end          else if (input\_valid)          begin              addr[31:2] <= input\_value[31:2];          end      end  //-----{从触摸屏获取输入}end  //-----{输出到触摸屏显示}begin  //根据需要显示的数修改此小节，  //触摸屏上共有44块显示区域，可显示44组32位数据  //44块显示区域从1开始编号，编号为1~44，      always @(posedge clk)      begin         case(display\_number)             6'd1:             begin                 display\_valid <= 1'b1;                 display\_name  <= "ADDR ";                 display\_value <= addr;             end             6'd2:             begin                 display\_valid <= 1'b1;                 display\_name  <= "INST ";                 display\_value <= inst;             end             default :             begin                 display\_valid <= 1'b0;                 display\_name  <= 40'd0;                 display\_value <= 32'd0;             end         endcase      end  //-----{输出到触摸屏显示}end  //----------------------{调用触摸屏模块}end---------------------//  endmodule |  四、实验实现截图    五、收获心得体会 在本次试验中，我了解到了只读存储器（Read-Only Memory，ROM）以非破坏性读出方式工作，只能读出无法写入信息。信息一旦写入后就固定下来，即使切断电源，信息也不会丢失，所以又称为固定存储器。ROM所存数据通常是装入整机前写入的，整机工作过程中只能读出，不像随机存储器能快速方便地改写存储内容。ROM所存数据稳定，断电后所存数据也不会改变，并且结构较简单，使用方便，因而常用于存储各种固定程序和数据。  随机存取存储器（英语：Random Access Memory，缩写：RAM），也叫主存，是与CPU直接交换数据的内部存储器。它可以随时读写（刷新时除外），而且速度很快，通常作为操作系统或其他正在运行中的程序的临时数据存储介质。RAM工作时可以随时从任何一个指定的地址写入（存入）或读出（取出）信息。它与ROM的最大区别是数据的易失性，即一旦断电所存储的数据将随之丢失。RAM在计算机和数字系统中用来暂时存储程序、数据和中间结果  Vivado软件自带了BMG IP核(Block Memory Generator，块RAM生成器)，可以配置成RAM或者ROM。这两者的区别是RAM是一种随机存取存储器，不仅仅可以存储数据，同时支持对存储的数据进行修改；而ROM是一种只读存储器，也就是说，在正常工作时只能读出数据，而不能写入数据。  实验越来越难，参考设计的顶层模块框图也越来越复杂，更加考验小组的合作能力，需要不断地交流沟通，在指导书的配合下完成，最后要结合之前的实验，为了下一个单周期CPU实验打好基础。 |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 实验六 单周期CPU实现一、实验分析说明 1.理解 MIPS 指令结构，理解 MIPS 指令集中常用指令的功能和编码，学会对这些  指令进行归纳分类 。  2.了解熟悉 MIPS 体系的处理器结构，如延迟槽，哈佛结构的概念 。  3.熟悉并掌握单周期 CPU 的原理和设计 。  4.进一步加强运用 verilog 语言进行电路设计的能力 。  5.为后续设计多周期 cpu 的实验打下基础 。 二、实验设计思路 1.学习MIPS 指令集，深入理解常用指令的功能和编码，并进行归纳确定处理器 各部件的控制码，比如使用何种 ALU 运算，是否写寄存器堆等 。  2.确定自己本次实验中的准备实现的 MIPS 指令，要求至少实现一条 load 指令 、 一条 store 指令 、 10 条基础运算指令 、一条跳转指令 。其中基础运算指令最好包含多 种类型的操作，必须包含一条加法和一条减法指令 。不考虑指令可能产生异常的情 况 。单周期 CPU 的实验重点是搭建出一个 CPU 架构，为避免被繁琐的指令所困惑，建 议在单周期 CPU 实验中只实现十几条指令 。  3.对准备实现的指令进行分析  mips 基础指令特性归纳表   |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | 指令 类型 | 汇编指令 | 指令码 | 源操作  数 **1** | 源操作  数 **2** | 目的寄  存器 | 功能描述 | | R 型  指令 | addu rd , rs , rt | 000000 rs | rt | rd |00000|100001 | [rs] | [rt] | rd | GPR[rd]= GPR [rs]+ GPR [rt] | |  |  |  |  |  |  | |  |  |  |  |  |  | | I 型  指令 | addiu rt,rs,imm | 001001 rs | rt  |imm | [rs] | sign\_ext(imm) | rt | GPR [rt]= GPR  [rs]+sign\_ext(imm) | |  |  |  |  |  |  | | J 型  指令 | j target | 000010|target | PC | target |  | 跳转, PC= {PC[31:28],target,2’b00} |   4.自行设计本次实验的方案，画出结构框图，大致结构框图如图 7.1 。 图 7.1 中 粗线表示接口位数和种类不定，需要在自己的结构框图中详细给出 。从图 7.1 中可以 看出，本次实验是需要用到之前实验的结果的，比如 ALU 模块 、寄存器堆模块 、指令 ROM 模块和数据 RAM 模块，其中ROM 和 RAM 要使用自行搭建的异步存储器 。单周期 CPU 是指一条指令的所有操作在一个时钟周期内执行完 。设计中所有寄存器和存储器都是异步读同步写的，即读出数据不需要时钟控制，但写入数据需时钟控。故单周期 CPU 的运作即 ：在一个时钟周期内，根据 PC 值从指令 ROM 中读出相应的指令，将指令译码后从寄存器堆中读出需要的操作数，送往 ALU 模块，ALU 模块运算得到结果 。如果是 store 指令，则 ALU 运算结果为数据存储的地址，就向数据 RAM 发出写请求，在下一个时钟上升沿真正写入到数据存储器 。如果是 load 指令，则 ALU 运算结果为数据存储的地址，根据该值从数据存 RAM 中读出数据，送往寄存器堆根据目的寄存器发出写请求，在下一个时钟上升沿真正写入到寄存器堆中 。如果非 load/store 操作，若有写寄存器堆的操作，则直接将 ALU 运算结果送往寄存器堆根据目的寄存器发出写请求，在下一个时钟上升沿真正写入到寄存器堆中 。如果是分支跳转指令，则是需要将结果写入到 pc 寄存器中的 。  5.根据设计的实验方案，使用 verilog 编写相应代码 。    6.依据自己设计中实现的指令，编写一段不少于 20 行的汇编程序，力求验证到所有实现的指令 。该段汇编程序是需要内嵌到自行搭建的异步指令 ROM 中的 。  测试所用汇编程序详述   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 指令 地址 | 汇编指令 | 结 果描 述 | 机器指令的机器码 | | | **16** 进制 | 二进制 | | 00H | addiu ,$1, $0,#1 | [$1] = 0000\_0001H | 24010001 | 0010 0100 0000 0001 0000 0000 0000 0001  \_ \_ \_ \_\_ \_ \_ \_ | |  |  |  |  |  | |  |  |  |  |  |   7.对编写的代码进行仿真，得到正确的波形图 。  8.将以上设计作为一个单独的模块，设计一个外围模块去调用该模块 。外围模块中需调用封装好的 LCD 触摸屏模块，观察单周期 CPU 的内部状态，比如32 个寄存器的值，PC 的值等 。并且需要利用触摸功能输入特定数据 RAM 地址，从该RAM 的调试端口读出数据显示在屏上，以达到实时观察数据存储器内部数据变化的效果 。通过这些手段，可以在板上充分验证 CPU 的正确性 。  9.将编写的代码进行综合布局布线，并下载到实验箱中的 FPGA 板子上进行演 示 。  单周期 CPU 实现的指令   |  |  |  | | --- | --- | --- | | 指令名称 | 汇编指令 | 功能描述 | | 无符号加法 | addu rd,rs,rt | GPR[rd] = GPR[rs] + GPR[rt] | | 无符号减法 | subu rd,rs,rt | GPR[rd] = GPR[rs] - GPR[rt] | | 有符号比较 ，小于置位 | slt rd,rs,rt | GPR[rd] = (sign(GPR[rs]) < sign(GPR[rt])) | | 按位与 | and rd,rs,rt | GPR[rd] = GPR[rs] & GPR[rt] | | 按位或非 | nor rd,rs,rt | GPR[rd] = ~(GPR[rs] | GPR[rt]) | | 按位或 | or rd,rs,rt | GPR[rd] = GPR[rs] | GPR[rt] | | 按位异或 | xor rd,rs,rt | GPR[rd] = GPR[rs] ^ GPR[rt] | | 逻辑左移 | sll rd,rt,shf | GPR[rd] = zero(GPR[rt]) << shf | | 逻辑右移 | srl rd,rt,shf | GPR[rd] = zero(GPR[rt]) >> shf | | 立即数无符号加法 | addiu rt,rs,imm | GPR[rt] = GPR[rs] + sign\_ext(imm) | | 立即数装载高位 | lui rt,imm | GPR[rt] = {imm, 16'd0} | | 从内存装载字 | lw rt,offset(base) | GPR[rt] = Mem[GPR[base] + sign\_ext (offset)] | | 向内存存储字 | sw rt,offset(base) | Mem[GPR[base]+sign\_ext (offset)] = GPR[rt] | | 判断相等跳转 | beq rs,rt,offset | if GPR[rs] = GPR[rt] then PC=PC+ sign\_ext (offset)<<2 | | 判断不等跳转 | bne rs,rt,offset | if GPR[rs] ≠ GPR[rt] then PC=PC+ sign\_ext (offset)<<2 | | 直接跳转 | j target | PC = {PC[31:28], target<<2} |   单周期 CPU 实现的 mips 指令特性归纳   |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 指令 类型 | 汇编  指令 | 指令码 | 源操作  数 **1** | 源操作  数 **2** | 源操作  数 **3** | 目的寄  存器 | 功能描述 | | R 型  指令 | addu  rd,rs,rt | 000000|rs|rt|rd|00  000|100001 | [rs] | [rt] |  | rd | GPR[rd]=GPR[rs]+GPR[rt] | | subu  rd,rs,rt | 000000|rs|rt|rd|00  000|100011 | [rs] | [rt] |  | rd | GPR[rd]=GPR[rs]-GPR[rt] | | slt rd,rs,rt | 000000|rs|rt|rd|00  000|101010 | [rs] | [rt] |  | rd | GPR[rd]=(sign(GPR[rs])<sign(GPR[rt])) | | and rd,rs,rt | 000000|rs|rt|rd|00  000|100100 | [rs] | [rt] |  | rd | GPR[rd]=GPR[rs]&GPR[rt] | | nor rd,rs,rt | 000000|rs|rt|rd|00  000|100111 | [rs] | [rt] |  | rd | GPR[rd]=~(GPR[rs]|GPR[rt]) | | or rd,rs,rt | 000000|rs|rt|rd|00  000|100101 | [rs] | [rt] |  | rd | GPR[rd]=GPR[rs]|GPR[rt] | | xor rd,rs,rt | 000000|rs|rt|rd|00  000|100110 | [rs] | [rt] |  | rd | GPR[rd]=GPR[rs]^GPR[rt] | | sll rd,rt,shf | 000000|00000|rt|r  d|shf|000000 |  | [rt] |  | rd | GPR[rd]=zero(GPR[rt])<<shf | | srl rd,rt,shf | 000000|00000|rt|r  d|shf|000010 |  | [rt] |  | rd | GPR[rd]=zero(GPR[rt])>>shf | | I 型  指令 | addiu rt,rs,imm | 001001|rs|rt|imm | [rs] | sign\_ext  (imm) |  | rt | GPR[rt]=GPR[rs]+ sign\_ext (imm) | | beq rs,rt,offset | 000100|rs|rt|offset | [rs] | [rt] |  |  | if GPR[rs]=GPR[rt] then PC=PC+  sign\_ext (offset)<<2 | | bne  rs,rt,offset | 000101|rs|rt|offset | [rs] | [rt] |  |  | if GPR[rs]≠GPR[rt] then PC=PC+  sign\_ext (offset)<<2 | |  | lw  rt,offset(b) | 100011|b|rt|offset | [b] | sign\_ext (offset) |  | rt | GPR[rt]=Mem[GPR[b]+sign\_ext(offset)] | |  | sw  rt,offset(b) | 101011|b|rt|offset | [b] | sign\_ext (offset) | [rt] |  | Mem[GPR[b]+sign\_ext(offset)]=GPR[rt] | |  | lui rt,imm | 001111|00000|rt|i  mm |  | {imm,  16'd0} |  | rt | GPR[rt]= {imm, 16'd0} | | J 型  指令 | j target | 000010|target |  |  |  |  | PC= {PC[31:28],target<<2} |   单周期 CPU 测试所用汇编程序详述   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 指令  地址 | 汇编指令 | 结果描述 | 机器指令的机器码 | | | 16 进制 | 二进制 | | 00H | addiu $1,$0,#1 | [$1] = 0000\_0001H | 24010001 | 0010 0100 0000 0001 0000 0000 0000 0001  \_ \_ \_ \_ \_ \_ \_ | | 04H | sll $2,$1,#4 | [$2] = 0000\_0010H | 00011100 | 0000 0000 0000 0001 0001 0001 0000 0000  \_ \_ \_ \_ \_ \_ \_ | | 08H | addu $3,$2,$1 | [$3] = 0000\_0011H | 00411821 | 0000 0000 0100 0001 0001 1000 0010 0001  \_ \_ \_ \_ \_ \_ \_ | | 0CH | srl $4,$2,#2 | [$4] = 0000\_0004H | 00022082 | 0000 0000 0000 0010 0010 0000 1000 0010  \_ \_ \_ \_ \_ \_ \_ | | 10H | subu $5,$3,$4 | [$5] = 0000\_000DH | 00642823 | 0000 0000 0110 0100 0010 1000 0010 0011  \_ \_ \_ \_ \_ \_ \_ | | 14H | sw $5,#19($1) | Mem[0000\_0014H] =  0000 000DH  \_ | AC250013 | 1010 1100 0010 0101 0000 0000 0001 0011  \_ \_ \_ \_ \_ \_ \_ | | 18H | nor $6,$5,$2 | [$6] = FFFF\_FFE2H | 00A23027 | 0000 0000 1010 0010 0011 0000 0010 0111  \_ \_ \_ \_ \_ \_ \_ | | 1CH | or $7,$6,$3 | [$7] = FFFF\_FFF3H | 00C33825 | 0000 0000 1100 0011 0011 1000 0010 0101  \_ \_ \_ \_ \_ \_ \_ | | 20H | xor $8,$7,$6 | [$8] = 0000\_0011H | 00E64026 | 0000 0000 1110 0110 0100 0000 0010 0110  \_ \_ \_ \_ \_ \_ \_ | | 24H | sw $8,#28($0) | Mem[0000\_001CH] =  0000 0011H  \_ | AC08001C | 1010 1100 0000 1000 0000 0000 0001 1100  \_ \_ \_ \_ \_ \_ \_ | | 28H | slt $9,$6,$7 | [$9] = 0000\_0001H | 00C7482A | 0000 0000 1100 0111 0100 1000 0010 1010  \_ \_ \_ \_ \_ \_ \_ | | 2CH | beq $9,$1,#2 | 跳转到指令 34H | 11210002 | 0001 0001 0010 0001 0000 0000 0000 0010  \_ \_ \_ \_ \_ \_ \_ | | 30H | addiu $1,$0,#4 | 不执行 | 24010004 | 0010 0100 0000 0001 0000 0000 0000 0100  \_ \_ \_ \_ \_ \_ \_ | | 34H | lw $10,#19($1) | [$10] = 0000\_000DH | 8C2A0013 | 1000 1100 0010 1010 0000 0000 0001 0011  \_ \_ \_ \_ \_ \_ \_ | | 38H | bne $10,$5,#3 | 不跳转 | 15450003 | 0001 0101 0100 0101 0000 0000 0000 0011  \_ \_ \_ \_ \_ \_ \_ | | 3CH | and $11,$2,$1 | [$11] = 0000\_0000H | 00415824 | 0000 0000 0100 0001 0101 1000 0010 0100  \_ \_ \_ \_ \_ \_ \_ | | 40H | sw $11,#28($0) | Men[0000\_001CH] =  0000 0000H  \_ | AC0B001C | 1010 1100 0000 1011 0000 0000 0001 1100  \_ \_ \_ \_ \_ \_ \_ | | 44H | sw $4,#16($0) | Mem[0000\_0010H] =  0000 0004H  \_ | AC040010 | 1010 1100 0000 0100 0000 0000 0001 0000  \_ \_ \_ \_ \_ \_ \_ | | 48H | lui $12,#12 | [$12] = 000C\_0000H | 3C0C000C | 0011 1100 0000 1100 0000 0000 0000 1100  \_ \_ \_ \_ \_ \_ \_ | | 4CH | j 00H | 跳转指令 00H | 08000000 | 0000 1000 0000 0000 0000 0000 0000 0000  \_ \_ \_ \_ \_ \_ \_ |  三、实验源代码inst\_rom.v  |  | | --- | | //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  // > 文件名: inst\_rom.v  // > 描述 ：异步指令存储器模块，采用寄存器搭建而成，类似寄存器堆  // > 内嵌好指令，只读，异步读  //   > 作者  : 崔金泽 苗春雨 李子怡  //   > 日期  : 2022-10-06  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  module inst\_rom(      input      [4 :0] addr, // 指令地址      output reg [31:0] inst       // 指令      );      wire [31:0] inst\_rom[19:0];  // 指令存储器，字节地址7'b000\_0000~7'b111\_1111      //------------- 指令编码 ---------|指令地址|--- 汇编指令 -----|- 指令结果 -----//      assign inst\_rom[ 0] = 32'h24010001; // 00H: addiu $1 ,$0,#1   | $1 = 0000\_0001H      assign inst\_rom[ 1] = 32'h00011100; // 04H: sll   $2 ,$1,#4   | $2 = 0000\_0010H      assign inst\_rom[ 2] = 32'h00411821; // 08H: addu  $3 ,$2,$1   | $3 = 0000\_0011H      assign inst\_rom[ 3] = 32'h00022082; // 0CH: srl   $4 ,$2,#2   | $4 = 0000\_0004H      assign inst\_rom[ 4] = 32'h00642823; // 10H: subu  $5 ,$3,$4   | $5 = 0000\_000DH      assign inst\_rom[ 5] = 32'hAC250013; // 14H: sw    $5 ,#19($1) | Mem[0000\_0014H] = 0000\_000DH      assign inst\_rom[ 6] = 32'h00A23027; // 18H: nor   $6 ,$5,$2   | $6 = FFFF\_FFE2H      assign inst\_rom[ 7] = 32'h00C33825; // 1CH: or    $7 ,$6,$3   | $7 = FFFF\_FFF3H      assign inst\_rom[ 8] = 32'h00E64026; // 20H: xor   $8 ,$7,$6   | $8 = 0000\_0011H      assign inst\_rom[ 9] = 32'hAC08001C; // 24H: sw    $8 ,#28($0) | Mem[0000\_001CH] = 0000\_0011H      assign inst\_rom[10] = 32'h00C7482A; // 28H: slt   $9 ,$6,$7   | $9 = 0000\_0001H      assign inst\_rom[11] = 32'h11210002; // 2CH: beq   $9 ,$1,#2   | 跳转到指令34H      assign inst\_rom[12] = 32'h24010004; // 30H: addiu $1 ,$0,#4   | 不执行      assign inst\_rom[13] = 32'h8C2A0013; // 34H: lw    $10,#19($1) | $10 = 0000\_000DH      assign inst\_rom[14] = 32'h15450003; // 38H: bne   $10,$5,#3   | 不跳转      assign inst\_rom[15] = 32'h00415824; // 3CH: and   $11,$2,$1   | $11 = 0000\_0000H      assign inst\_rom[16] = 32'hAC0B001C; // 40H: sw    $11,#28($0) | Men[0000\_001CH] = 0000\_0000H      assign inst\_rom[17] = 32'hAC040010; // 44H: sw    $4 ,#16($0) | Mem[0000\_0010H] = 0000\_0004H      assign inst\_rom[18] = 32'h3C0C000C; // 48H: lui   $12,#12     | [R12] = 000C\_0000H      assign inst\_rom[19] = 32'h08000000; // 4CH: j     00H         | 跳转指令00H      //读指令,取4字节      always @(\*)      begin          case (addr)              5'd0 : inst <= inst\_rom[0 ];              5'd1 : inst <= inst\_rom[1 ];              5'd2 : inst <= inst\_rom[2 ];              5'd3 : inst <= inst\_rom[3 ];              5'd4 : inst <= inst\_rom[4 ];              5'd5 : inst <= inst\_rom[5 ];              5'd6 : inst <= inst\_rom[6 ];              5'd7 : inst <= inst\_rom[7 ];              5'd8 : inst <= inst\_rom[8 ];              5'd9 : inst <= inst\_rom[9 ];              5'd10: inst <= inst\_rom[10];              5'd11: inst <= inst\_rom[11];              5'd12: inst <= inst\_rom[12];              5'd13: inst <= inst\_rom[13];              5'd14: inst <= inst\_rom[14];              5'd15: inst <= inst\_rom[15];              5'd16: inst <= inst\_rom[16];              5'd17: inst <= inst\_rom[17];              5'd18: inst <= inst\_rom[18];              5'd19: inst <= inst\_rom[19];              default: inst <= 32'd0;          endcase      end  endmodule |  regfile.v  |  | | --- | | `timescale 1ns / 1ps  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //   > 文件名: regfile.v  //   > 描述  ：寄存器堆模块，同步写，异步读  //   > 作者  : 崔金泽 苗春雨 李子怡  //   > 日期  : 2022-10-06  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  module regfile(      input             clk,      input             wen,      input      [4 :0] raddr1,      input      [4 :0] raddr2,      input      [4 :0] waddr,      input      [31:0] wdata,      output reg [31:0] rdata1,      output reg [31:0] rdata2,      input      [4 :0] test\_addr,      output reg [31:0] test\_data      );      reg [31:0] rf[31:0];        // three ported register file      // read two ports combinationally      // write third port on rising edge of clock      // register 0 hardwired to 0      always @(posedge clk)      begin          if (wen)          begin              rf[waddr] <= wdata;          end      end        //读端口1      always @(\*)      begin          case (raddr1)              5'd1 : rdata1 <= rf[1 ];              5'd2 : rdata1 <= rf[2 ];              5'd3 : rdata1 <= rf[3 ];              5'd4 : rdata1 <= rf[4 ];              5'd5 : rdata1 <= rf[5 ];              5'd6 : rdata1 <= rf[6 ];              5'd7 : rdata1 <= rf[7 ];              5'd8 : rdata1 <= rf[8 ];              5'd9 : rdata1 <= rf[9 ];              5'd10: rdata1 <= rf[10];              5'd11: rdata1 <= rf[11];              5'd12: rdata1 <= rf[12];              5'd13: rdata1 <= rf[13];              5'd14: rdata1 <= rf[14];              5'd15: rdata1 <= rf[15];              5'd16: rdata1 <= rf[16];              5'd17: rdata1 <= rf[17];              5'd18: rdata1 <= rf[18];              5'd19: rdata1 <= rf[19];              5'd20: rdata1 <= rf[20];              5'd21: rdata1 <= rf[21];              5'd22: rdata1 <= rf[22];              5'd23: rdata1 <= rf[23];              5'd24: rdata1 <= rf[24];              5'd25: rdata1 <= rf[25];              5'd26: rdata1 <= rf[26];              5'd27: rdata1 <= rf[27];              5'd28: rdata1 <= rf[28];              5'd29: rdata1 <= rf[29];              5'd30: rdata1 <= rf[30];              5'd31: rdata1 <= rf[31];              default : rdata1 <= 32'd0;          endcase      end      //读端口2      always @(\*)      begin          case (raddr2)              5'd1 : rdata2 <= rf[1 ];              5'd2 : rdata2 <= rf[2 ];              5'd3 : rdata2 <= rf[3 ];              5'd4 : rdata2 <= rf[4 ];              5'd5 : rdata2 <= rf[5 ];              5'd6 : rdata2 <= rf[6 ];              5'd7 : rdata2 <= rf[7 ];              5'd8 : rdata2 <= rf[8 ];              5'd9 : rdata2 <= rf[9 ];              5'd10: rdata2 <= rf[10];              5'd11: rdata2 <= rf[11];              5'd12: rdata2 <= rf[12];              5'd13: rdata2 <= rf[13];              5'd14: rdata2 <= rf[14];              5'd15: rdata2 <= rf[15];              5'd16: rdata2 <= rf[16];              5'd17: rdata2 <= rf[17];              5'd18: rdata2 <= rf[18];              5'd19: rdata2 <= rf[19];              5'd20: rdata2 <= rf[20];              5'd21: rdata2 <= rf[21];              5'd22: rdata2 <= rf[22];              5'd23: rdata2 <= rf[23];              5'd24: rdata2 <= rf[24];              5'd25: rdata2 <= rf[25];              5'd26: rdata2 <= rf[26];              5'd27: rdata2 <= rf[27];              5'd28: rdata2 <= rf[28];              5'd29: rdata2 <= rf[29];              5'd30: rdata2 <= rf[30];              5'd31: rdata2 <= rf[31];              default : rdata2 <= 32'd0;          endcase      end       //调试端口，读出寄存器值显示在触摸屏上      always @(\*)      begin          case (test\_addr)              5'd1 : test\_data <= rf[1 ];              5'd2 : test\_data <= rf[2 ];              5'd3 : test\_data <= rf[3 ];              5'd4 : test\_data <= rf[4 ];              5'd5 : test\_data <= rf[5 ];              5'd6 : test\_data <= rf[6 ];              5'd7 : test\_data <= rf[7 ];              5'd8 : test\_data <= rf[8 ];              5'd9 : test\_data <= rf[9 ];              5'd10: test\_data <= rf[10];              5'd11: test\_data <= rf[11];              5'd12: test\_data <= rf[12];              5'd13: test\_data <= rf[13];              5'd14: test\_data <= rf[14];              5'd15: test\_data <= rf[15];              5'd16: test\_data <= rf[16];              5'd17: test\_data <= rf[17];              5'd18: test\_data <= rf[18];              5'd19: test\_data <= rf[19];              5'd20: test\_data <= rf[20];              5'd21: test\_data <= rf[21];              5'd22: test\_data <= rf[22];              5'd23: test\_data <= rf[23];              5'd24: test\_data <= rf[24];              5'd25: test\_data <= rf[25];              5'd26: test\_data <= rf[26];              5'd27: test\_data <= rf[27];              5'd28: test\_data <= rf[28];              5'd29: test\_data <= rf[29];              5'd30: test\_data <= rf[30];              5'd31: test\_data <= rf[31];              default : test\_data <= 32'd0;          endcase      end  endmodule |  alu.v  |  | | --- | | `timescale 1ns / 1ps  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //   > 文件名: alu.v  //   > 描述  ：ALU模块，可做12种操作  //   > 作者  : 崔金泽 苗春雨 李子怡  //   > 日期  : 2022-10-06  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  module alu(      input  [11:0] alu\_control,  // ALU控制信号      input  [31:0] alu\_src1,     // ALU操作数1,为补码      input  [31:0] alu\_src2,     // ALU操作数2，为补码      output [31:0] alu\_result    // ALU结果      );      // ALU控制信号，独热码      wire alu\_add;   //加法操作      wire alu\_sub;   //减法操作      wire alu\_slt;   //有符号比较，小于置位，复用加法器做减法      wire alu\_sltu;  //无符号比较，小于置位，复用加法器做减法      wire alu\_and;   //按位与      wire alu\_nor;   //按位或非      wire alu\_or;    //按位或      wire alu\_xor;   //按位异或      wire alu\_sll;   //逻辑左移      wire alu\_srl;   //逻辑右移      wire alu\_sra;   //算术右移      wire alu\_lui;   //高位加载      assign alu\_add  = alu\_control[11];      assign alu\_sub  = alu\_control[10];      assign alu\_slt  = alu\_control[ 9];      assign alu\_sltu = alu\_control[ 8];      assign alu\_and  = alu\_control[ 7];      assign alu\_nor  = alu\_control[ 6];      assign alu\_or   = alu\_control[ 5];      assign alu\_xor  = alu\_control[ 4];      assign alu\_sll  = alu\_control[ 3];      assign alu\_srl  = alu\_control[ 2];      assign alu\_sra  = alu\_control[ 1];      assign alu\_lui  = alu\_control[ 0];      wire [31:0] add\_sub\_result;      wire [31:0] slt\_result;      wire [31:0] sltu\_result;      wire [31:0] and\_result;      wire [31:0] nor\_result;      wire [31:0] or\_result;      wire [31:0] xor\_result;      wire [31:0] sll\_result;      wire [31:0] srl\_result;      wire [31:0] sra\_result;      wire [31:0] lui\_result;      assign and\_result = alu\_src1 & alu\_src2;      // 与结果为两数按位与      assign or\_result  = alu\_src1 | alu\_src2;      // 或结果为两数按位或      assign nor\_result = ~or\_result;               // 或非结果为或结果按位取反      assign xor\_result = alu\_src1 ^ alu\_src2;      // 异或结果为两数按位异或      assign lui\_result = {alu\_src2[15:0], 16'd0};  // 立即数装载结果为立即数移位至高半字节  //-----{加法器}begin  //add,sub,slt,sltu均使用该模块      wire [31:0] adder\_operand1;      wire [31:0] adder\_operand2;      wire        adder\_cin     ;      wire [31:0] adder\_result  ;      wire        adder\_cout    ;      assign adder\_operand1 = alu\_src1;      assign adder\_operand2 = alu\_add ? alu\_src2 : ~alu\_src2;      assign adder\_cin      = ~alu\_add; //减法需要cin      adder adder\_module(      .operand1(adder\_operand1),      .operand2(adder\_operand2),      .cin     (adder\_cin     ),      .result  (adder\_result  ),      .cout    (adder\_cout    )      );      //加减结果      assign add\_sub\_result = adder\_result;      //slt结果      //adder\_src1[31] adder\_src2[31] adder\_result[31]      //       0             1           X(0或1)       "正-负"，显然小于不成立      //       0             0             1           相减为负，说明小于      //       0             0             0           相减为正，说明不小于      //       1             1             1           相减为负，说明小于      //       1             1             0           相减为正，说明不小于      //       1             0           X(0或1)       "负-正"，显然小于成立      assign slt\_result[31:1] = 31'd0;      assign slt\_result[0]    = (alu\_src1[31] & ~alu\_src2[31]) | (~(alu\_src1[31]^alu\_src2[31]) & adder\_result[31]);      //sltu结果      //对于32位无符号数比较，相当于33位有符号数（{1'b0,src1}和{1'b0,src2}）的比较，最高位0为符号位      //故，可以用33位加法器来比较大小，需要对{1'b0,src2}取反,即需要{1'b0,src1}+{1'b1,~src2}+cin      //但此处用的为32位加法器，只做了运算:                             src1   +    ~src2   +cin      //32位加法的结果为{adder\_cout,adder\_result},则33位加法结果应该为{adder\_cout+1'b1,adder\_result}      //对比slt结果注释，知道，此时判断大小属于第二三种情况，即源操作数1符号位为0，源操作数2符号位为0      //结果的符号位为1，说明小于，即adder\_cout+1'b1为2'b01，即adder\_cout为0      assign sltu\_result = {31'd0, ~adder\_cout};  //-----{加法器}end  //-----{移位器}begin      // 移位分三步进行，      // 第一步根据移位量低2位即[1:0]位做第一次移位，      // 第二步在第一次移位基础上根据移位量[3:2]位做第二次移位，      // 第三步在第二次移位基础上根据移位量[4]位做第三次移位。      wire [4:0] shf;      assign shf = alu\_src1[4:0];      wire [1:0] shf\_1\_0;      wire [1:0] shf\_3\_2;      assign shf\_1\_0 = shf[1:0];      assign shf\_3\_2 = shf[3:2];         // 逻辑左移      wire [31:0] sll\_step1;      wire [31:0] sll\_step2;      assign sll\_step1 = {32{shf\_1\_0 == 2'b00}} & alu\_src2                   // 若shf[1:0]="00",不移位                       | {32{shf\_1\_0 == 2'b01}} & {alu\_src2[30:0], 1'd0}     // 若shf[1:0]="01",左移1位                       | {32{shf\_1\_0 == 2'b10}} & {alu\_src2[29:0], 2'd0}     // 若shf[1:0]="10",左移2位                       | {32{shf\_1\_0 == 2'b11}} & {alu\_src2[28:0], 3'd0};    // 若shf[1:0]="11",左移3位      assign sll\_step2 = {32{shf\_3\_2 == 2'b00}} & sll\_step1                  // 若shf[3:2]="00",不移位                       | {32{shf\_3\_2 == 2'b01}} & {sll\_step1[27:0], 4'd0}    // 若shf[3:2]="01",第一次移位结果左移4位                       | {32{shf\_3\_2 == 2'b10}} & {sll\_step1[23:0], 8'd0}    // 若shf[3:2]="10",第一次移位结果左移8位                       | {32{shf\_3\_2 == 2'b11}} & {sll\_step1[19:0], 12'd0};  // 若shf[3:2]="11",第一次移位结果左移12位      assign sll\_result = shf[4] ? {sll\_step2[15:0], 16'd0} : sll\_step2;     // 若shf[4]="1",第二次移位结果左移16位      // 逻辑右移      wire [31:0] srl\_step1;      wire [31:0] srl\_step2;      assign srl\_step1 = {32{shf\_1\_0 == 2'b00}} & alu\_src2                   // 若shf[1:0]="00",不移位                       | {32{shf\_1\_0 == 2'b01}} & {1'd0, alu\_src2[31:1]}     // 若shf[1:0]="01",右移1位,高位补0                       | {32{shf\_1\_0 == 2'b10}} & {2'd0, alu\_src2[31:2]}     // 若shf[1:0]="10",右移2位,高位补0                       | {32{shf\_1\_0 == 2'b11}} & {3'd0, alu\_src2[31:3]};    // 若shf[1:0]="11",右移3位,高位补0      assign srl\_step2 = {32{shf\_3\_2 == 2'b00}} & srl\_step1                  // 若shf[3:2]="00",不移位                       | {32{shf\_3\_2 == 2'b01}} & {4'd0, srl\_step1[31:4]}    // 若shf[3:2]="01",第一次移位结果右移4位,高位补0                       | {32{shf\_3\_2 == 2'b10}} & {8'd0, srl\_step1[31:8]}    // 若shf[3:2]="10",第一次移位结果右移8位,高位补0                       | {32{shf\_3\_2 == 2'b11}} & {12'd0, srl\_step1[31:12]}; // 若shf[3:2]="11",第一次移位结果右移12位,高位补0      assign srl\_result = shf[4] ? {16'd0, srl\_step2[31:16]} : srl\_step2;    // 若shf[4]="1",第二次移位结果右移16位,高位补0        // 算术右移      wire [31:0] sra\_step1;      wire [31:0] sra\_step2;      assign sra\_step1 = {32{shf\_1\_0 == 2'b00}} & alu\_src2                                 // 若shf[1:0]="00",不移位                       | {32{shf\_1\_0 == 2'b01}} & {alu\_src2[31], alu\_src2[31:1]}           // 若shf[1:0]="01",右移1位,高位补符号位                       | {32{shf\_1\_0 == 2'b10}} & {{2{alu\_src2[31]}}, alu\_src2[31:2]}      // 若shf[1:0]="10",右移2位,高位补符号位                       | {32{shf\_1\_0 == 2'b11}} & {{3{alu\_src2[31]}}, alu\_src2[31:3]};     // 若shf[1:0]="11",右移3位,高位补符号位      assign sra\_step2 = {32{shf\_3\_2 == 2'b00}} & sra\_step1                                // 若shf[3:2]="00",不移位                       | {32{shf\_3\_2 == 2'b01}} & {{4{sra\_step1[31]}}, sra\_step1[31:4]}    // 若shf[3:2]="01",第一次移位结果右移4位,高位补符号位                       | {32{shf\_3\_2 == 2'b10}} & {{8{sra\_step1[31]}}, sra\_step1[31:8]}    // 若shf[3:2]="10",第一次移位结果右移8位,高位补符号位                       | {32{shf\_3\_2 == 2'b11}} & {{12{sra\_step1[31]}}, sra\_step1[31:12]}; // 若shf[3:2]="11",第一次移位结果右移12位,高位补符号位      assign sra\_result = shf[4] ? {{16{sra\_step2[31]}}, sra\_step2[31:16]} : sra\_step2;    // 若shf[4]="1",第二次移位结果右移16位,高位补符号位  //-----{移位器}end      // 选择相应结果输出      assign alu\_result = (alu\_add|alu\_sub) ? add\_sub\_result[31:0] :                          alu\_slt           ? slt\_result :                          alu\_sltu          ? sltu\_result :                          alu\_and           ? and\_result :                          alu\_nor           ? nor\_result :                          alu\_or            ? or\_result  :                          alu\_xor           ? xor\_result :                          alu\_sll           ? sll\_result :                          alu\_srl           ? srl\_result :                          alu\_sra           ? sra\_result :                          alu\_lui           ? lui\_result :                          32'd0;  endmodule |  data\_mem.v  |  | | --- | | `timescale 1ns / 1ps  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //   > 文件名: data\_mem.v  //   > 描述  ：异步数据存储器模块，采用寄存器搭建而成，类似寄存器堆  //   >         同步写，异步读  //   > 作者  : 崔金泽 苗春雨 李子怡  //   > 日期  : 2022-10-06  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  module data\_ram(      input         clk,         // 时钟      input  [3:0]  wen,         // 字节写使能      input  [4:0] addr,        // 地址      input  [31:0] wdata,       // 写数据      output reg [31:0] rdata,       // 读数据        //调试端口，用于读出数据显示      input  [4 :0] test\_addr,      output reg [31:0] test\_data  );      reg [31:0] DM[31:0];  //数据存储器，字节地址7'b000\_0000~7'b111\_1111      //写数据      always @(posedge clk)    // 当写控制信号为1，数据写入内存      begin          if (wen[3])          begin              DM[addr][31:24] <= wdata[31:24];          end      end      always @(posedge clk)      begin          if (wen[2])          begin              DM[addr][23:16] <= wdata[23:16];          end      end      always @(posedge clk)      begin          if (wen[1])          begin              DM[addr][15: 8] <= wdata[15: 8];          end      end      always @(posedge clk)      begin          if (wen[0])          begin              DM[addr][7 : 0] <= wdata[7 : 0];          end      end        //读数据,取4字节      always @(\*)      begin          case (addr)              5'd0 : rdata <= DM[0 ];              5'd1 : rdata <= DM[1 ];              5'd2 : rdata <= DM[2 ];              5'd3 : rdata <= DM[3 ];              5'd4 : rdata <= DM[4 ];              5'd5 : rdata <= DM[5 ];              5'd6 : rdata <= DM[6 ];              5'd7 : rdata <= DM[7 ];              5'd8 : rdata <= DM[8 ];              5'd9 : rdata <= DM[9 ];              5'd10: rdata <= DM[10];              5'd11: rdata <= DM[11];              5'd12: rdata <= DM[12];              5'd13: rdata <= DM[13];              5'd14: rdata <= DM[14];              5'd15: rdata <= DM[15];              5'd16: rdata <= DM[16];              5'd17: rdata <= DM[17];              5'd18: rdata <= DM[18];              5'd19: rdata <= DM[19];              5'd20: rdata <= DM[20];              5'd21: rdata <= DM[21];              5'd22: rdata <= DM[22];              5'd23: rdata <= DM[23];              5'd24: rdata <= DM[24];              5'd25: rdata <= DM[25];              5'd26: rdata <= DM[26];              5'd27: rdata <= DM[27];              5'd28: rdata <= DM[28];              5'd29: rdata <= DM[29];              5'd30: rdata <= DM[30];              5'd31: rdata <= DM[31];          endcase      end      //调试端口，读出特定内存的数据      always @(\*)      begin          case (test\_addr)              5'd0 : test\_data <= DM[0 ];              5'd1 : test\_data <= DM[1 ];              5'd2 : test\_data <= DM[2 ];              5'd3 : test\_data <= DM[3 ];              5'd4 : test\_data <= DM[4 ];              5'd5 : test\_data <= DM[5 ];              5'd6 : test\_data <= DM[6 ];              5'd7 : test\_data <= DM[7 ];              5'd8 : test\_data <= DM[8 ];              5'd9 : test\_data <= DM[9 ];              5'd10: test\_data <= DM[10];              5'd11: test\_data <= DM[11];              5'd12: test\_data <= DM[12];              5'd13: test\_data <= DM[13];              5'd14: test\_data <= DM[14];              5'd15: test\_data <= DM[15];              5'd16: test\_data <= DM[16];              5'd17: test\_data <= DM[17];              5'd18: test\_data <= DM[18];              5'd19: test\_data <= DM[19];              5'd20: test\_data <= DM[20];              5'd21: test\_data <= DM[21];              5'd22: test\_data <= DM[22];              5'd23: test\_data <= DM[23];              5'd24: test\_data <= DM[24];              5'd25: test\_data <= DM[25];              5'd26: test\_data <= DM[26];              5'd27: test\_data <= DM[27];              5'd28: test\_data <= DM[28];              5'd29: test\_data <= DM[29];              5'd30: test\_data <= DM[30];              5'd31: test\_data <= DM[31];          endcase      end  endmodule |  四、实验实现截图    五、收获心得体会 通过本次实验，掌握了单周期cpu数据通路图的构成、原理及其设计方法，利用ISE将每一个模块实现了一遍，对于每一个模块的实现和运作有了更深的理解，掌握了对于cpu的检测方法，熟悉了从cpu的组合到测试到结果的每一个过程。 单周期 CPU 的 inst\_rom .v 和 data\_ram .v 均为自行搭建的异步存储器 ，也是在实验 五中的源代码 。单周期 CPU 的源代码还缺少regfile .v 和 alu .v ， 这 两个源文件都已在实验三和实验四中给出 ，本实验中直接通过“Add Copy ofSource”添加到工程中即可 。其中 alu .v 还调用了实验一中的 adder .v 。  在实验过程中，遇到了在always@()中括号里的值的问题，出现了突然触发效果或者一直没有触发效果的情况。对于这个问题，认真的查找了网上的资料，并通过不断的更改其中的值做尝试，解决了问题。单周期CPU用到了之前实验里的代码，在这里直接调用即可，由此可见基础的重要性，要是前边的实验没有打好基础，在这个实验中我们会遇到不少的问题。 |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 实验七 多周期CPU实现一、实验分析说明 1.在单周期 CPU 实验完成的提前下，理解多周期的概念 。  2.熟悉并掌握多周期 CPU 的原理和设计 。  3.进一步提升运用 verilog 语言进行电路设计的能力 。  4.为后续实现流水线 cpu 的课程设计打下基础 。 二、实验设计思路 1.本次实验是对单周期 CPU 实验的拔高，也是为流水线 CPU 打下基础 。前期的实 验准备同单周期 CPU 的实验，在单周期 CPU 中只要求实现了十几条指令，但此处要求 扩展到 30 多条指令 。多周期 CPU 是指，一条指令需要花费多个周期才能完成所有操作，在每个周期内 只做一部分操作，比如 ：取指 、译码 、执行 、访存 、写回，此时，一条指令执行完，共需 5 个周期，每个周期只做一部分操作 。将 CPU 划分为多周期的优势在于，每个时钟周期内CPU 需要做的工作就变少，因 此频率可以更高，且每个部件做的事情单一了，比如取指部件只负责从指令存储器中 取出指令，因此 CPU 可以进行流水工作，也相当于一个时钟周期完成一条指令 。频率 更高，依然相当于是一个周期完成一条指令，因此 CPU 可以运行的更快 。本次实验就是将实验六所实现的单周期 CPU 划分为多周期的，并扩展指令到 30 多条 。  2.依据单周期实验的设计框图，将其划分为多个功能块，每个功能块占用一个周 期完成，即每个功能块从上一个功能块获取信息，作相关动作，完成后将结果锁存到 寄存器中，作为下一个功能块的输入 。建议划分为教课书上的 5 个功能块 ：取指 、译 码 、执行 、访存 、写回，将理论与实践相结合 。  3.画出划分后的多周期 CPU 的框图，大致框图如图 8.1 。从图中可以看出指令每 个周期走完一个功能块，进入下一个功能块 。标注的 clk 箭头是去往相邻模块的中间 锁存器，是因为每个模块的输出需要锁存到寄存器中，下一个模块会从该寄存器中读 出数据作为自己的输入，寄存器的锁存是需要时钟控制的 。值得注意的是，写回模块 所做的就是从访存模块获取要写入寄存器堆的数据和目的寄存器，送往寄存器堆，其 所需的 clk 信号是最终发生在寄存器堆的写操作上的 。 自己设计的框图中要求力求精 细，可参考教科书上的 5 级流水的框图 。    4.本次实验是需要用到之前实验的结果的，比如 ALU 模块 、寄存器堆模块 、指令 ROM 模块和数据 RAM 模块，其中ROM 和 RAM 建议使用调用库 IP 实例化的同步存储器，因为存储器在实际应用中基本都是同步读写的，为了更贴近真实情况，此处建议使用 同步 RAM 和 ROM 。  5.在实验五中生成的同步 RAM 和 ROM，都是在发送地址后的下一拍才能获得对应 数据的 。故在在使用同步存储器时，从指令和数据存储器中读取数据就需要等待一拍 时钟了，即取指令需要两拍时间，load 操作也需要两拍时间 。在真实的处理器系统 中，取指令和访存其实都是需要多拍时钟的 。   |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | 指令 类型 | 汇编指令 | 指令码 | 源操作  数 **1** | 源操作  数 **2** | 目的寄  存器 | 功能描述 | | R 型  指令 | addu rd , rs , rt | 000000 rs | rt | rd |00000|100001 | [rs] | [rt] | rd | GPR[rd]=GPR[rs]+GPR[rt] | |  |  |  |  |  |  | |  |  |  |  |  |  | | I 型  指令 | addiu rt,rs,imm | 001001 rs | rt  |imm | [rs] | sign\_ext(imm) | rt | GPR[rt]=GPR[rs]+sign\_ext(i  mm) | |  |  |  |  |  |  | | J 型  指令 | j target | 000010|target | PC | target |  | 跳转,  PC= PC[31:28],target,2’b00} |  |  |  |  |  |  | | --- | --- | --- | --- | --- | | 指令 地址 | 汇编指令 | 结 果描 述 | 机器指令的机器码 | | | **16** 进制 | 二进制 | | 00H | addiu ,$1, $0,#1 | [$1] = 0000\_0001H | 24010001 | 0010 0100 0000 0001 0000 0000 0000 0001  \_ \_ \_ \_\_ \_ \_ \_ | |  |  |  |  |  | |  |  |  |  |  |   7.根据设计的实验方案，使用 verilog 编写相应代码 。  8.对编写的代码进行仿真，得到正确的波形图 。  9.将以上设计作为一个单独的模块，设计一个外围模块去调用该模块，见图外围模块中需调用封装好的 LCD 触摸屏模块，观察多周期 CPU 的内部状态，比如32 个寄存器的值，各模块 PC 的值等 。并且需要利用触摸功能输入特定数据 RAM 地址，从该 RAM 的调试端口读出数据显示在屏上，以达到实时观察数据存储器内部数据变化的效果 。通过这些手段，可以在板上充分验证 CPU 的正确性 。  10.将编写的代码进行综合布局布线，并下载到实验箱中的 FPGA 板子上进行演 示 。  多周期 CPU 新增的指令   |  |  |  | | --- | --- | --- | | 指令名称 | 汇编指令 | 功能描述 | | 无符号小于置位 | sltu rd,rs,rt | GPR[rd] = (zero(GPR[rs]) < zero(GPR[rt])) | | 跳转和链接寄存器 | jalr rs | GPR[31] = PC, PC = GPR[rs] | | 跳转寄存器 | r rs  j | PC = GPR[rs] | | 向量逻辑左移字 | sllv rd,rt,rs | GPR[rd] = zero(GPR[rt]) << (GPR[rs]%32) | | 算术右移字 | sra rd,rt,shf | GPR[rd] = sign(GPR[rt]) >> shf | | 向量算术右移字 | srav rd,rt,rs | GPR[rd] = sign(GPR[rt]) >> (GPR[rs]%32) | | 向量逻辑右移字 | srlv rd,rt,rs | GPR[rd] = zero(GPR[rt]) >> GPR[rs] | | 立即数小于置位 | slti rt,rs,imm | GPR[rt] = (sign(GPR[rs]) < sign\_ext(imm)) | | 立即数无符号小于置位 | sltiu rt,rs,imm | GPR[rt] = (zero(GPR[rs]) < sign\_ext (imm)) | | 大于或等于零跳转 | bgez rs,offset | if GPR[rs] ≥ 0 then PC=PC+ sign\_ext (offset)<<2 | | 大于零跳转 | bgtz rs,offset | if GPR[rs] > 0 then PC=PC+ sign\_ext (offset)<<2 | | 小于或等于零跳转 | blez rs,offset | if GPR[rs] ≤ 0 then PC=PC+ sign\_ext (offset)<<2 | | 小于零跳转 | bltz rs,offset | if GPR[rs] < 0 then PC=PC+ sign\_ext (offset)<<2 | | 装载字节 | lb rt,offset(base) | GPR[rt] = sign(Mem[GPR[base]+sign\_ext(offset)]) | | 无符号装载字节 | lbu rt,offset(base) | GPR[rt] = zero(Mem[GPR[base]+sign\_ext (offset)]) | | 存储字节 | sb rt,offset(base) | Mem[GPR[base]+sign\_ext (offset)] = GPR[rt] | | 立即数按位与 | andi rt,rs,imm | GPR[rt] = GPR[rs] & zero\_ext (imm) | | 立即数按位或 | ori rt,rs,imm | GPR[rt] = GPR[rs] | zero\_ext (imm) | | 立即数按位异或 | xori rt,rs,imm | GPR[rt] = GPR[rs] ^ zero\_ext (imm) | | 跳转和链接 | jal target | GPR[31] = PC, PC = {PC[31:28], target<<2} |   多周期 CPU 实现的 mips 指令特性归纳   |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 指令 类型 | 汇编  指令 | 指令码 | 源操作  数 **1** | 源操作  数 **2** | 源操作  数 **3** | 目的寄  存器 | 功能描述 | | R 型  指令 | addu  rd,rs,rt | 000000|rs|rt|rd|0000  0|100001 | [rs] | [rt] |  | rd | GPR[rd]=GPR[rs]+GPR[rt] | | subu  rd,rs,rt | 000000|rs|rt|rd|0000  0|100011 | [rs] | [rt] |  | rd | GPR[rd]=GPR[rs]-GPR[rt] | | slt rd,rs,rt | 000000|rs|rt|rd|0000  0|101010 | [rs] | [rt] |  | rd | GPR[rd]=(sign(GPR[rs])<sign(GPR[rt])) | | sltu rd,rs,rt | 000000|rs|rt|rd|0000  0|101011 | [rs] | [rt] |  | rd | GPR[rd]=(zero(GPR[rs])<zero(GPR[rt])) | | jalr rs | 000000|rs|00000|111  11|00000|001001 | [rs] |  |  | 31 | GPR[31]=PC,PC=GPR[rs] | | jr rs | 000000|rs|00000000  00|00000|001000 | [rs] |  |  |  | PC=GPR[rs] | |  | and rd,rs,rt | 000000|rs|rt|rd|0000  0|100100 | [rs] | [rt] |  | rd | GPR[rd]=GPR[rs]&GPR[rt] | | nor rd,rs,rt | 000000|rs|rt|rd|0000  0|100111 | [rs] | [rt] |  | rd | GPR[rd]=~(GPR[rs]|GPR[rt]) | | or rd,rs,rt | 000000|rs|rt|rd|0000  0|100101 | [rs] | [rt] |  | rd | GPR[rd]=GPR[rs]|GPR[rt] | | xor rd,rs,rt | 000000|rs|rt|rd|0000  0|100110 | [rs] | [rt] |  | rd | GPR[rd]=GPR[rs]^GPR[rt] | | sll rd,rt,shf | 000000|00000|rt|rd|s  hf|000000 |  | [rt] |  | rd | GPR[rd]=zero(GPR[rt])<<shf | | sllv rd,rt,rs | 000000|rs|rt|rd|0000  0|000100 | [rs] | [rt] |  | rd | GPR[rd]=zero(GPR[rt])<<(GPR[rs]%32) | | sra  rd,rt,shf | 000000|00000|rt|rd|s  hf|000011 |  | [rt] |  | rd | GPR[rd]=sign(GPR[rt])>>shf | | srav  rd,rt,rs | 000000|rs|rt|rd|0000  0|000111 | [rs] | [rt] |  | rd | GPR[rd]=sign(GPR[rt])>>(GPR[rs]%32) | | srl rd,rt,shf | 000000|00000|rt|rd|s  hf|000010 |  | [rt] |  | rd | GPR[rd]=zero(GPR[rt])>>shf | | srlv rd,rt,rs | 000000|rs|rt|rd|0000  0|000110 | [rs] | [rt] |  | rd | GPR[rd]=zero(GPR[rt])>>GPR[rs] | |  | addiu  rt,rs,imm | 001001|rs|rt|imm | [rs] | sign\_ext  (imm) |  | rt | GPR[rt]=GPR[rs]+ sign\_ext (imm) | |  | slti rt,rs,imm | 001010|rs|rt|imm | [rs] | sign\_ext  (imm) |  | rt | GPR[rt]=(sign(GPR[rs])< sign\_ext  (imm)) | |  | sltiu rt,rs,imm | 001011|rs|rt|imm | [rs] | sign\_ext  (imm) |  | rt | GPR[rt]=(zero(GPR[rs])< sign\_ext  (imm)) | |  | beq rs,rt,offset | 000100|rs|rt|offset | [rs] | [rt] |  |  | if GPR[rs]=GPR[rt] then PC=PC+  sign\_ext (offset)<<2 | |  | bgez  rs,offset | 000001|rs|00001|off  set | [rs] |  |  |  | if GPR[rs]≥0 then PC=PC+  sign ext (offset)<<2 | |  | bgtz  rs,offset | 000111|rs|00000|off  set | [rs] |  |  |  | if GPR[rs]>0 then PC=PC+  sign\_ext (offset)<<2 | | I 型  指令 | blez  rs,offset | 000110|rs|00000|off  set | [rs] |  |  |  | if GPR[rs]≤0 then PC=PC+  sign ext (offset)<<2 | |  | bltz  rs,offset | 000001|rs|00000|off  set | [rs] |  |  |  | if GPR[rs]<0 thenPC=PC+  sign ext (offset)<<2 | |  | bne rs,rt,offset | 000101|rs|rt|offset | [rs] | [rt] |  |  | if GPR[rs]≠GPR[rt] then PC=PC+  sign\_ext (offset)<<2 | |  | lw  rt,offset(b) | 100011|b|rt|offset | [b] | sign\_ext (offset) |  | rt | GPR[rt]=Mem[GPR[b]+sign\_ext  (offset)] | |  | sw  rt,offset(b) | 101011|b|rt|offset | [b] | sign\_ext (offset) | [rt] |  | Mem[GPR[b]+sign\_ext  (offset)]=GPR[rt] | |  | lb  rt,offset(b) | 100000|b|rt|offset | [b] | sign\_ext  (offset) |  | rt | GPR[rt]=sign(Mem[GPR[b]+sign\_ext  (offset)]) | |  | lbu  rt,offset(b) | 100100|b|rt|offset | [b] | sign\_ext  (offset) |  | rt | GPR[rt]=zero(Mem[GPR[b]+sign\_ext  (offset)]) | |  | sb  rt,offset(b) | 101000|b|rt|offset | [b] | sign\_ext  (offset) | [rt] |  | Mem[GPR[b]+sign\_ext(offset)]=GPR[rt] |  三、实验源代码multi\_cycle\_cpu.v `timescale 1ns / 1ps  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //   > 文件名: multi\_cycle\_cpu.v  //   > 描述  :多周期CPU模块，共实现36条指令  //   >        指令rom和数据ram均实例化xilinx IP得到，为同步读写  //   > 作者  : 崔金泽 苗春雨 李子怡  //   > 日期  : 2022-10-04  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*  module multi\_cycle\_cpu(  // 多周期cpu      input clk,           // 时钟      input resetn,        // 复位信号，低电平有效        //display data      input  [ 4:0] rf\_addr,      input  [31:0] mem\_addr,      output [31:0] rf\_data,      output [31:0] mem\_data,      output [31:0] IF\_pc,      output [31:0] IF\_inst,      output [31:0] ID\_pc,      output [31:0] EXE\_pc,      output [31:0] MEM\_pc,      output [31:0] WB\_pc,      output [31:0] display\_state      );  //----------------------{控制多周期的状态机}begin------------------------//      reg [2:0] state;       // 当前状态      reg [2:0] next\_state;  // 下一状态      //展示当前处理器正在执行哪个模块      assign display\_state = {29'd0,state};      // 状态机状态      parameter IDLE   = 3'd0;  // 开始      parameter FETCH  = 3'd1;  // 取指      parameter DECODE = 3'd2;  // 译码      parameter EXE    = 3'd3;  // 执行      parameter MEM    = 3'd4;  // 访存      parameter WB     = 3'd5;  // 写回      always @ (posedge clk)        // 当前状态      begin          if (!resetn) begin        // 如果复位信号有效              state <= IDLE;       // 当前状态为 开始          end          else begin                // 否则              state <= next\_state;  // 为下一状态          end      end      wire IF\_over;     // IF模块已执行完      wire ID\_over;     // ID模块已执行完      wire EXE\_over;    // EXE模块已执行完      wire MEM\_over;    // MEM模块已执行完      wire WB\_over;     // WB模块已执行完      wire jbr\_not\_link;//分支指令(非link类)，只走IF和ID级      always @ (\*)                             // 下一状态      begin          case (state)              IDLE :              begin                  next\_state = FETCH;    // 开始->取指              end              FETCH:              begin                  if (IF\_over)                  begin                      next\_state = DECODE;   // 取指->译码                  end                  else                  begin                      next\_state = FETCH;    // 取指->译码                  end              end              DECODE:              begin                  if (ID\_over)                  begin                      // 译码->执行或取指                      next\_state = jbr\_not\_link ? FETCH : EXE;                  end                  else                  begin                      next\_state = DECODE;   // 取指->译码                  end              end              EXE:              begin                  if (EXE\_over)                  begin                      next\_state = MEM;      // 执行->访存                  end                  else                  begin                      next\_state = EXE;   // 取指->译码                  end              end              MEM:              begin                  if (MEM\_over)                  begin                      next\_state = WB;       // 访存->写回                  end                  else                  begin                      next\_state = MEM;   // 取指->译码                  end              end              WB:              begin                  if (WB\_over)                  begin                      next\_state = FETCH;    // 写回->取指                  end                  else                  begin                      next\_state = WB;   // 取指->译码                  end              end              default : next\_state = IDLE;          endcase      end      //5模块的valid信号      wire IF\_valid;      wire ID\_valid;      wire EXE\_valid;      wire MEM\_valid;      wire WB\_valid;      assign  IF\_valid = (state == FETCH );  // 当前状态为取指时，IF级有效      assign  ID\_valid = (state == DECODE);  // 当前状态为译码时，ID级有效      assign EXE\_valid = (state == EXE   );  // 当前状态为执行时，EXE级有效      assign MEM\_valid = (state == MEM   );  // 当前状态为访存时，MEM级有效      assign  WB\_valid = (state == WB    );  // 当前状态为写回时，WB级有效  //-----------------------{控制多周期的状态机}end-------------------------//  //--------------------------{5级间的总线}begin---------------------------//      wire [ 63:0] IF\_ID\_bus;   // IF->ID级总线      wire [149:0] ID\_EXE\_bus;  // ID->EXE级总线      wire [105:0] EXE\_MEM\_bus; // EXE->MEM级总线      wire [ 69:0] MEM\_WB\_bus;  // MEM->WB级总线        //锁存以上总线信号      reg [ 63:0] IF\_ID\_bus\_r;      reg [149:0] ID\_EXE\_bus\_r;      reg [105:0] EXE\_MEM\_bus\_r;      reg [ 69:0] MEM\_WB\_bus\_r;        //IF到ID的锁存信号      always @(posedge clk)      begin          if(IF\_over)          begin              IF\_ID\_bus\_r <= IF\_ID\_bus;          end      end      //ID到EXE的锁存信号      always @(posedge clk)      begin          if(ID\_over)          begin              ID\_EXE\_bus\_r <= ID\_EXE\_bus;          end      end      //EXE到MEM的锁存信号      always @(posedge clk)      begin          if(EXE\_over)          begin              EXE\_MEM\_bus\_r <= EXE\_MEM\_bus;          end      end      //MEM到WB的锁存信号      always @(posedge clk)      begin          if(MEM\_over)          begin              MEM\_WB\_bus\_r <= MEM\_WB\_bus;          end      end  //---------------------------{5级间的总线}end----------------------------//  //--------------------------{其他交互信号}begin--------------------------//      //跳转总线      wire [ 32:0] jbr\_bus;      //IF与inst\_rom交互      wire [31:0] inst\_addr;      wire [31:0] inst;      //MEM与data\_ram交互      wire [ 3:0] dm\_wen;      wire [31:0] dm\_addr;      wire [31:0] dm\_wdata;      wire [31:0] dm\_rdata;      //ID与regfile交互      wire [ 4:0] rs;      wire [ 4:0] rt;      wire [31:0] rs\_value;      wire [31:0] rt\_value;        //WB与regfile交互      wire        rf\_wen;      wire [ 4:0] rf\_wdest;      wire [31:0] rf\_wdata;  //---------------------------{其他交互信号}end---------------------------//  //-------------------------{各模块实例化}begin---------------------------//      wire next\_fetch; //即将运行取指模块，需要先锁存PC值      //当前状态为decode，且指令为跳转分支指令(非link类)，且decode执行完成      //或者，当前状态为wb，且wb执行完成，则即将进入fetch状态      assign next\_fetch = (state==DECODE & ID\_over & jbr\_not\_link)                        | (state==WB     & WB\_over);      fetch IF\_module(             // 取指级          .clk       (clk       ),  // I, 1          .resetn    (resetn    ),  // I, 1          .IF\_valid  (IF\_valid  ),  // I, 1          .next\_fetch(next\_fetch),  // I, 1          .inst      (inst      ),  // I, 32          .jbr\_bus   (jbr\_bus   ),  // I, 33          .inst\_addr (inst\_addr ),  // O, 32          .IF\_over   (IF\_over   ),  // O, 1          .IF\_ID\_bus (IF\_ID\_bus ),  // O, 64            //展示PC和取出的指令          .IF\_pc     (IF\_pc     ),          .IF\_inst   (IF\_inst   )      );      decode ID\_module(               // 译码级          .ID\_valid    (ID\_valid    ),  // I, 1          .IF\_ID\_bus\_r (IF\_ID\_bus\_r ),  // I, 64          .rs\_value    (rs\_value    ),  // I, 32          .rt\_value    (rt\_value    ),  // I, 32          .rs          (rs          ),  // O, 5          .rt          (rt          ),  // O, 5          .jbr\_bus     (jbr\_bus     ),  // O, 33          .jbr\_not\_link(jbr\_not\_link),  // O, 1          .ID\_over     (ID\_over     ),  // O, 1          .ID\_EXE\_bus  (ID\_EXE\_bus  ),  // O, 150            //展示PC          .ID\_pc      (ID\_pc      )      );      exe EXE\_module(                   // 执行级          .EXE\_valid   (EXE\_valid   ),  // I, 1          .ID\_EXE\_bus\_r(ID\_EXE\_bus\_r),  // I, 150          .EXE\_over    (EXE\_over    ),  // O, 1          .EXE\_MEM\_bus (EXE\_MEM\_bus ),  // O, 106            //展示PC          .EXE\_pc      (EXE\_pc      )      );      mem MEM\_module(                     // 访存级          .clk          (clk          ),  // I, 1          .MEM\_valid    (MEM\_valid    ),  // I, 1          .EXE\_MEM\_bus\_r(EXE\_MEM\_bus\_r),  // I, 106          .dm\_rdata     (dm\_rdata     ),  // I, 32          .dm\_addr      (dm\_addr      ),  // O, 32          .dm\_wen       (dm\_wen       ),  // O, 4          .dm\_wdata     (dm\_wdata     ),  // O, 32          .MEM\_over     (MEM\_over     ),  // O, 1          .MEM\_WB\_bus   (MEM\_WB\_bus   ),  // O, 70            //展示PC          .MEM\_pc       (MEM\_pc       )      );        wb WB\_module(                     // 写回级          .WB\_valid    (WB\_valid    ),  // I, 1          .MEM\_WB\_bus\_r(MEM\_WB\_bus\_r),  // I, 70          .rf\_wen      (rf\_wen      ),  // O, 1          .rf\_wdest    (rf\_wdest    ),  // O, 5          .rf\_wdata    (rf\_wdata    ),  // O, 32          .WB\_over     (WB\_over     ),  // O, 1            //展示PC          .WB\_pc       (WB\_pc       )      );      inst\_rom inst\_rom\_module(         // 指令存储器          .clka       (clk           ),  // I, 1 ,时钟          .addra      (inst\_addr[9:2]),  // I, 8 ,指令地址          .douta      (inst          )   // O, 32,指令      );      regfile rf\_module(        // 寄存器堆模块          .clk    (clk      ),  // I, 1          .wen    (rf\_wen   ),  // I, 1          .raddr1 (rs       ),  // I, 5          .raddr2 (rt       ),  // I, 5          .waddr  (rf\_wdest ),  // I, 5          .wdata  (rf\_wdata ),  // I, 32          .rdata1 (rs\_value ),  // O, 32          .rdata2 (rt\_value ),  // O, 32          //display rf          .test\_addr(rf\_addr),          .test\_data(rf\_data)      );        data\_ram data\_ram\_module(   // 数据存储模块          .clka   (clk         ),  // I, 1,  时钟          .wea    (dm\_wen      ),  // I, 1,  写使能          .addra  (dm\_addr[9:2]),  // I, 8,  读地址          .dina   (dm\_wdata    ),  // I, 32, 写数据          .douta  (dm\_rdata    ),  // O, 32, 读数据          //display mem          .clkb   (clk          ),          .web    (4'd0         ),          .addrb  (mem\_addr[9:2]),          .doutb  (mem\_data     ),          .dinb   (32'd0        )      );  //--------------------------{各模块实例化}end----------------------------//  endmodule 四、实验实现截图    五、收获心得体会 本次实验因为有了单周期CPU的基础，所以在实现的时候对于每条指令的理解更加清晰。  一开始的时候误以为不仅仅要实现多周期而且要用流水线，导致迟迟无法下手，后来发现并不需要实现流水线，相对难度降低了很多。并且一开始一直不理解IR、ADR等的作用，增加IR指令寄存器，目的是使指令代码保持稳定，pc写使能控制信号PCWre，是确保pc适时修改，原因都是和多周期工作的CPU有关。10个基本部件都容易实现，最难的就是多周期的控制器。控制器根据有限状态机生成，其中状态生成逻辑和状态保存逻辑根据状态机的转换图容易实现，状态输出逻辑中的控制信号生成较为复杂。要考虑不同指令的特点，分为3类，存取指令，数据处理指令和跳转指令。同类指令还要考虑Ⅰ型和R型指令的不同，造成输出的控制信号难于准确无误控制。需要对数据通路以及指令十分熟悉并且沿着指令执行的每个阶段去判断应该输出何种控制信号，此处花费功夫较多。 |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 实验八 静态5级流水线CPU实现一、实验分析说明 1 . 在多周期 CPU 实验完成的提前下 ，深入理解 CPU 流水线的概念 。  2 . 熟悉并掌握流水线 CPU 的原理和设计 。  3 . 最终检验运用 verilog 语言进行电路设计的能力 。  4 . 通过亲自设计实现静态 5 级流水线 CPU ，加深对计算机组成原理和体系结构理 论知识的理解 。  5 . 培养对 CPU 设计的兴趣 ，加深对 CPU 现有架构的理解和深思 。 二、实验设计思路 1 . 本课程设计是对之前课程实验的拔高 。前期的课程设计准备同多周期 CPU 的实  验 ，主体部分可以直接使用多周期 CPU 实验的设计方案 ，但在多周期 CPU 中只要求实 现了 30 多条指令 ，此处要求扩展到 40 多条指令 。  多周期 CPU 在单周期基础上提高了时钟频率 ，但并没有改善执行一条指令的时 间 ，且存在资源闲置的问题 ，例如当指令在执行级有效时 ，译码级实际上在空转 。若 每一级都在执行有效的指令 ，将解决资源闲置的问题 。最理想的情况是 ， 当第一条指 令从取指级转换到下一级译码时 ，第二条指令进入取指级 ， 当第一条指令完成译码进 入执行级时 ，第二条指令进入译码级 ，第三条指令进入取指级……静态 5 级流水 CPU 就是基于这样的设计思路 。  在流水 CPU 中 ， 当在一时钟周期内完成了某一条指令的全部执行时（写回级完  成） ，则有望在下一时钟周期内完成下条指令的执行 ， 因此依然相当于是一个周期完 成一条指令 ，而时钟频率更高 ， 因此 CPU 可以运行的更快 。  本次课程就是将上次所实现的多周期 CPU 更改结构为静态 5 级流水线 CPU 。  IM 4724  9 . 1 mips 基础指令特性归纳表   |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | 指令 类型 | 汇编指令 | 指令码 | 源操作  数 **1** | 源操作  数 **2** | 目的寄  存器 | 功能描述 | | R 型  指令 | addu rd , rs , rt | 000000 rs | rt | rd |00000|100001 | [rs] | [rt] | rd | GPR[rd]=GPR[rs]+GPR[rt] | |  |  |  |  |  |  | |  |  |  |  |  |  | | I 型  指令 | addiu rt,rs,imm | 001001 rs | rt  |imm | [rs] | sign\_ext(imm) | rt | GPR[rt]=GPR[rs]+sign\_ext(imm) | |  |  |  |  |  |  | | J 型  指令 | j target | 000010|target | next\_pc | target |  | 跳转,  PC= {next\_pc[31:28],target,2’b00} |   表 9 . 2 测试所用汇编程序描述表   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 指令 地址 | 汇编指令 | 结 果描 述 | 机器指令的机器码 | | | **16** 进制 | 二进制 | | 00H | addiu ,$1, $0,#1 | [$1] = 0000\_0001H | 24010001 | 0010 0100 0000 0001 0000 0000 0000 0001  \_ \_ \_ \_\_ \_ \_ \_ | |  |  |  |  |  | |  |  |  |  |  |   5 级流水线 CPU 新增的指令   |  |  |  | | --- | --- | --- | | 指令名称 | 汇编指令 | 功能描述 | | 有符号乘法 | mult rs,rt | (HI,LO)=sign(GPR[rs])\*sign(GPR[rt]) | | 从 LO 寄存器取值 | mflo rd | GPR[rd]=[LO] | | 从 HI 寄存器取值 | mfhi rd | GPR[rd]=[HI] | | 向LO 寄存器存值 | mtlo rs | [LO]=GPR[rs] | | 向HI 寄存器存值 | mthi rs | [HI]=GPR[rs] | | 从cp0 寄存器取值 | mfc0 rt,cs | GPR[rt]=CPR[cs] | | 向cp0 寄存器存值 | mtc0 rt,cd | CPR[cd]=GPR[rt] | | 系统调用 | syscall | CPR[14 . 0]=PC, CPR[13 . 0][6:2]=01000, CPR[12 . 0][1]=1,  PC=CPR[15 . 1]+0x180 | | 异常返回 | eret | CPR[12 . 0][1]=0, PC=CPR[14 . 0] |   5 级流水线 CPU 实现的 mips 指令特性归纳   |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 指令  类型 | 汇编指令 | 指令码 | 源操作  数 **1** | 源操作  数 **2** | 源操作  数 **3** | 目的寄  存器 | 功能描述 | | R 型  指令 | addu rd,rs,rt | 000000|rs|rt|rd|000  00|100001 | [rs] | [rt] |  | rd | GPR[rd]=GPR[rs]+GPR[rt] | | subu rd,rs,rt | 000000|rs|rt|rd|000  00|100011 | [rs] | [rt] |  | rd | GPR[rd]=GPR[rs]-GPR[rt] | | slt rd,rs,rt | 000000|rs|rt|rd|000  00|101010 | [rs] | [rt] |  | rd | GPR[rd]=(sign(GPR[rs])<sign(GPR[rt])) | | sltu rd,rs,rt | 000000|rs|rt|rd|000  00|101011 | [rs] | [rt] |  | rd | GPR[rd]=(zero(GPR[rs])<zero(GPR[rt])) | | jalr rs | 000000|rs|00000|1  1111|00000|00100  1 | [rs] |  |  | 31 | GPR[31]=PC,PC=GPR[rs] | | r rs  j | 000000|rs|000000  0000|00000|00100  0 | [rs] |  |  |  | PC=GPR[rs] | | and rd,rs,rt | 000000|rs|rt|rd|000  00|100100 | [rs] | [rt] |  | rd | GPR[rd]=GPR[rs]&GPR[rt] | | nor rd,rs,rt | 000000|rs|rt|rd|000  00|100111 | [rs] | [rt] |  | rd | GPR[rd]=~(GPR[rs]|GPR[rt]) | | or rd,rs,rt | 000000|rs|rt|rd|000  00|100101 | [rs] | [rt] |  | rd | GPR[rd]=GPR[rs]|GPR[rt] | | xor rd,rs,rt | 000000|rs|rt|rd|000  00|100110 | [rs] | [rt] |  | rd | GPR[rd]=GPR[rs]^GPR[rt] | | sll rd,rt,shf | 000000|00000|rt|rd  |shf|000000 |  | [rt] |  | rd | GPR[rd]=zero(GPR[rt])<<shf | | sllv rd,rt,rs | 000000|rs|rt|rd|000  00|000100 | [rs] | [rt] |  | rd | GPR[rd]=zero(GPR[rt])<<(GPR[rs]%32) | | sra rd,rt,shf | 000000|00000|rt|rd  |shf|000011 |  | [rt] |  | rd | GPR[rd]=sign(GPR[rt])>>shf | | srav rd,rt,rs | 000000|rs|rt|rd|000  00|000111 | [rs] | [rt] |  | rd | GPR[rd]=sign(GPR[rt])>>(GPR[rs]%32) | | srl rd,rt,shf | 000000|00000|rt|rd  |shf|000010 |  | [rt] |  | rd | GPR[rd]=zero(GPR[rt])>>shf | | srlv rd,rt,rs | 000000|rs|rt|rd|000  00|000110 | [rs] | [rt] |  | rd | GPR[rd]=zero(GPR[rt])>>GPR[rs] | | mult rs,rt | 000000|rs|rt|00000  00000|011000 | [rs] | [rt] |  | (HI,LO) | (HI,LO)=sign(GPR[rs])\*sign(GPR[rt]) | | mflo rd | 000000|00000000  00|rd|00000|01001  0 | [LO] |  |  | rd | GPR[rd]=[LO] | | mfhi rd | 000000|00000000  00|rd|00000|01000  0 | [HI] |  |  | rd | GPR[rd]=[HI] | | mtlo rs | 000000|rs|000000  000000000|01001  1 | [rs] |  |  | LO | [LO]=GPR[rs] | | mthi rs | 000000|rs|000000  000000000|01000  1 | [rs] |  |  | HI | [HI]=GPR[rs] | | I 型  指令 | addiu  rt,rs,imm | 001001|rs|rt|imm | [rs] | sign\_ext  (imm) |  | rt | GPR[rt]=GPR[rs]+sign\_ext (imm) | | slti rt,rs,imm | 001010|rs|rt|imm | [rs] | sign\_ext  (imm) |  | rt | GPR[rt]=(sign(GPR[rs])<sign\_ext (imm)) | | sltiu  rt,rs,imm | 001011|rs|rt|imm | [rs] | sign\_ext  (imm) |  | rt | GPR[rt]=(zero(GPR[rs])<sign\_ext (imm)) | |  | beq rs,rt,offset | 000100|rs|rt|offset | [rs] | [rt] |  |  | if GPR[rs]=GPR[rt] then PC= next\_pc+  sign\_ext (offset)<<2 | |  | bgez  rs,offset | 000001|rs|00001|o  ffset | [rs] |  |  |  | if GPR[rs]≥0 then PC= next\_pc +  sign\_ext (offset)<<2 | |  | bgtz  rs,offset | 000111|rs|00000|of  fset | [rs] |  |  |  | if GPR[rs]>0 then PC= next\_pc +  sign ext (offset)<<2 | |  | blez rs,offset | 000110|rs|00000|o  ffset | [rs] |  |  |  | if GPR[rs]≤0 then PC= next\_pc +  sign ext (offset)<<2 | |  | bltz rs,offset | 000001|rs|00000|o  ffset | [rs] |  |  |  | if GPR[rs]<0 then PC= next\_pc +  sign\_ext (offset)<<2 | |  | bne  rs,rt,offset | 000101|rs|rt|offset | [rs] | [rt] |  |  | if GPR[rs]≠GPR[rt] then PC= next\_pc +  sign\_ext (offset)<<2 | |  | lw  rt,offset(b) | 100011|b|rt|offset | [b] | sign\_ext (offset) |  | rt | GPR[rt]=Mem[GPR[b]+sign\_ext (offset)] | |  | sw  rt,offset(b) | 101011|b|rt|offset | [b] | sign\_ext (offset) | [rt] |  | Mem[GPR[b]+sign\_ext (offset)]=GPR[rt] | |  | lb  rt,offset(b) | 100000|b|rt|offset | [b] | sign\_ext  (offset) |  | rt | GPR[rt]=sign(Mem[GPR[b]+sign\_ext  (offset)]) | |  | lbu  rt,offset(b) | 100100|b|rt|offset | [b] | sign\_ext  (offset) |  | rt | GPR[rt]=zero(Mem[GPR[b]+sign\_ext  (offset)]) | |  | sb  rt,offset(b) | 101000|b|rt|offset | [b] | sign\_ext  (offset) | [rt] |  | Mem[GPR[b]+sign\_ext (offset)]=GPR[rt] | |  | andi  rt,rs,imm | 001100|rs|rt|imm | [rs] | zero ext  \_  (imm) |  | rt | GPR[rt]=GPR[rs]&zero\_ext (imm) | |  | lui rt,imm | 001111|00000|rt|i  mm |  | {imm,  16'd0} |  | rt | GPR[rt]= {imm, 16'd0} | |  | ori rt,rs,imm | 001101|rs|rt|imm | [rs] | zero ext  \_  (imm) |  | rt | GPR[rt]=GPR[rs]|zero\_ext (imm) | |  | xori  rt,rs,imm | 001110|rs|rt|imm | [rs] | zero ext  \_  (imm) |  | rt | GPR[rt]=GPR[rs]^zero\_ext (imm) | | J 型  指令 | j target | 000010|target |  |  |  |  | [PC]= {next\_pc[31:28],target<<2} | |  | jal target | 000011|target |  |  |  |  | GPR[31]=PC,PC= {next\_pc[31:28],target <<2} | | cp0 指  令 | mfc0 rt,cs | 010000|00000|rt|cs |00000000|sel | CPR[cs. sel] |  |  | rt | GPR[rt]=CPR[cs . sel] | |  | mtc0 rt,cd | 010000|00100|rt|c  d|00000000|sel |  | GPR[rt] |  | CPR[cd .sel] | CPR[cd . sel]=GPR[rt] | |  | syscall | 000000|code|0011  00 |  |  |  |  | CPR[14 . 0]=PC,CPR[13 .0][6:2]=01000,C  PR[12 . 0][1]=1,PC=CPR[15 . 1]+0x180 并跳转 | |  | Eret | 010000|1|0000000  000000000000|01  1000 |  |  |  |  | CPR[12 . 0][1]=0,PC=CPR[14 . 0]并跳转 |  三、实验源代码pipeline\_cpu.v  |  | | --- | | `timescale 1ns / 1ps  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //   > 文件名: pipeline\_cpu.v  //   > 描述  :五级流水CPU模块，共实现XX条指令  //   >        指令rom和数据ram均实例化xilinx IP得到，为同步读写  //   > 作者  : 崔金泽 苗春雨 李子怡  //   > 日期  : 2022-10-04  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  module pipeline\_cpu(  // 多周期cpu      input clk,           // 时钟      input resetn,        // 复位信号，低电平有效        //display data      input  [ 4:0] rf\_addr,      input  [31:0] mem\_addr,      output [31:0] rf\_data,      output [31:0] mem\_data,      output [31:0] IF\_pc,      output [31:0] IF\_inst,      output [31:0] ID\_pc,      output [31:0] EXE\_pc,      output [31:0] MEM\_pc,      output [31:0] WB\_pc,        //5级流水新增      output [31:0] cpu\_5\_valid,      output [31:0] HI\_data,      output [31:0] LO\_data      );  //------------------------{5级流水控制信号}begin-------------------------//      //5模块的valid信号      reg IF\_valid;      reg ID\_valid;      reg EXE\_valid;      reg MEM\_valid;      reg WB\_valid;      //5模块执行完成信号,来自各模块的输出      wire IF\_over;      wire ID\_over;      wire EXE\_over;      wire MEM\_over;      wire WB\_over;      //5模块允许下一级指令进入      wire IF\_allow\_in;      wire ID\_allow\_in;      wire EXE\_allow\_in;      wire MEM\_allow\_in;      wire WB\_allow\_in;        // syscall和eret到达写回级时会发出cancel信号，      wire cancel;    // 取消已经取出的正在其他流水级执行的指令        //各级允许进入信号:本级无效，或本级执行完成且下级允许进入      assign IF\_allow\_in  = (IF\_over & ID\_allow\_in) | cancel;      assign ID\_allow\_in  = ~ID\_valid  | (ID\_over  & EXE\_allow\_in);      assign EXE\_allow\_in = ~EXE\_valid | (EXE\_over & MEM\_allow\_in);      assign MEM\_allow\_in = ~MEM\_valid | (MEM\_over & WB\_allow\_in );      assign WB\_allow\_in  = ~WB\_valid  | WB\_over;        //IF\_valid，在复位后，一直有效     always @(posedge clk)      begin          if (!resetn)          begin              IF\_valid <= 1'b0;          end          else          begin              IF\_valid <= 1'b1;          end      end        //ID\_valid      always @(posedge clk)      begin          if (!resetn || cancel)          begin              ID\_valid <= 1'b0;          end          else if (ID\_allow\_in)          begin              ID\_valid <= IF\_over;          end      end        //EXE\_valid      always @(posedge clk)      begin          if (!resetn || cancel)          begin              EXE\_valid <= 1'b0;          end          else if (EXE\_allow\_in)          begin              EXE\_valid <= ID\_over;          end      end        //MEM\_valid      always @(posedge clk)      begin          if (!resetn || cancel)          begin              MEM\_valid <= 1'b0;          end          else if (MEM\_allow\_in)          begin              MEM\_valid <= EXE\_over;          end      end        //WB\_valid      always @(posedge clk)      begin          if (!resetn || cancel)          begin              WB\_valid <= 1'b0;          end          else if (WB\_allow\_in)          begin              WB\_valid <= MEM\_over;          end      end        //展示5级的valid信号      assign cpu\_5\_valid = {12'd0         ,{4{IF\_valid }},{4{ID\_valid}},                            {4{EXE\_valid}},{4{MEM\_valid}},{4{WB\_valid}}};  //-------------------------{5级流水控制信号}end--------------------------//  //--------------------------{5级间的总线}begin---------------------------//      wire [ 63:0] IF\_ID\_bus;   // IF->ID级总线      wire [166:0] ID\_EXE\_bus;  // ID->EXE级总线      wire [153:0] EXE\_MEM\_bus; // EXE->MEM级总线      wire [117:0] MEM\_WB\_bus;  // MEM->WB级总线        //锁存以上总线信号      reg [ 63:0] IF\_ID\_bus\_r;      reg [166:0] ID\_EXE\_bus\_r;      reg [153:0] EXE\_MEM\_bus\_r;      reg [117:0] MEM\_WB\_bus\_r;        //IF到ID的锁存信号      always @(posedge clk)      begin          if(IF\_over && ID\_allow\_in)          begin              IF\_ID\_bus\_r <= IF\_ID\_bus;          end      end      //ID到EXE的锁存信号      always @(posedge clk)      begin          if(ID\_over && EXE\_allow\_in)          begin              ID\_EXE\_bus\_r <= ID\_EXE\_bus;          end      end      //EXE到MEM的锁存信号      always @(posedge clk)      begin          if(EXE\_over && MEM\_allow\_in)          begin              EXE\_MEM\_bus\_r <= EXE\_MEM\_bus;          end      end      //MEM到WB的锁存信号      always @(posedge clk)      begin          if(MEM\_over && WB\_allow\_in)          begin              MEM\_WB\_bus\_r <= MEM\_WB\_bus;          end      end  //---------------------------{5级间的总线}end----------------------------//  //--------------------------{其他交互信号}begin--------------------------//      //跳转总线      wire [ 32:0] jbr\_bus;      //IF与inst\_rom交互      wire [31:0] inst\_addr;      wire [31:0] inst;      //ID与EXE、MEM、WB交互      wire [ 4:0] EXE\_wdest;      wire [ 4:0] MEM\_wdest;      wire [ 4:0] WB\_wdest;        //MEM与data\_ram交互      wire [ 3:0] dm\_wen;      wire [31:0] dm\_addr;      wire [31:0] dm\_wdata;      wire [31:0] dm\_rdata;      //ID与regfile交互      wire [ 4:0] rs;      wire [ 4:0] rt;      wire [31:0] rs\_value;      wire [31:0] rt\_value;        //WB与regfile交互      wire        rf\_wen;      wire [ 4:0] rf\_wdest;      wire [31:0] rf\_wdata;        //WB与IF间的交互信号      wire [32:0] exc\_bus;  //---------------------------{其他交互信号}end---------------------------//  //-------------------------{各模块实例化}begin---------------------------//      wire next\_fetch; //即将运行取指模块，需要先锁存PC值      //IF允许进入时，即锁存PC值，取下一条指令      assign next\_fetch = IF\_allow\_in;      fetch IF\_module(             // 取指级          .clk       (clk       ),  // I, 1          .resetn    (resetn    ),  // I, 1          .IF\_valid  (IF\_valid  ),  // I, 1          .next\_fetch(next\_fetch),  // I, 1          .inst      (inst      ),  // I, 32          .jbr\_bus   (jbr\_bus   ),  // I, 33          .inst\_addr (inst\_addr ),  // O, 32          .IF\_over   (IF\_over   ),  // O, 1          .IF\_ID\_bus (IF\_ID\_bus ),  // O, 64            //5级流水新增接口          .exc\_bus   (exc\_bus   ),  // I, 32            //展示PC和取出的指令          .IF\_pc     (IF\_pc     ),  // O, 32          .IF\_inst   (IF\_inst   )   // O, 32      );      decode ID\_module(               // 译码级          .ID\_valid   (ID\_valid   ),  // I, 1          .IF\_ID\_bus\_r(IF\_ID\_bus\_r),  // I, 64          .rs\_value   (rs\_value   ),  // I, 32          .rt\_value   (rt\_value   ),  // I, 32          .rs         (rs         ),  // O, 5          .rt         (rt         ),  // O, 5          .jbr\_bus    (jbr\_bus    ),  // O, 33  //        .inst\_jbr   (inst\_jbr   ),  // O, 1          .ID\_over    (ID\_over    ),  // O, 1          .ID\_EXE\_bus (ID\_EXE\_bus ),  // O, 167            //5级流水新增          .IF\_over     (IF\_over     ),// I, 1          .EXE\_wdest   (EXE\_wdest   ),// I, 5          .MEM\_wdest   (MEM\_wdest   ),// I, 5          .WB\_wdest    (WB\_wdest    ),// I, 5            //展示PC          .ID\_pc       (ID\_pc       ) // O, 32      );      exe EXE\_module(                   // 执行级          .EXE\_valid   (EXE\_valid   ),  // I, 1          .ID\_EXE\_bus\_r(ID\_EXE\_bus\_r),  // I, 167          .EXE\_over    (EXE\_over    ),  // O, 1          .EXE\_MEM\_bus (EXE\_MEM\_bus ),  // O, 154            //5级流水新增          .clk         (clk         ),  // I, 1          .EXE\_wdest   (EXE\_wdest   ),  // O, 5            //展示PC          .EXE\_pc      (EXE\_pc      )   // O, 32      );      mem MEM\_module(                     // 访存级          .clk          (clk          ),  // I, 1          .MEM\_valid    (MEM\_valid    ),  // I, 1          .EXE\_MEM\_bus\_r(EXE\_MEM\_bus\_r),  // I, 154          .dm\_rdata     (dm\_rdata     ),  // I, 32          .dm\_addr      (dm\_addr      ),  // O, 32          .dm\_wen       (dm\_wen       ),  // O, 4          .dm\_wdata     (dm\_wdata     ),  // O, 32          .MEM\_over     (MEM\_over     ),  // O, 1          .MEM\_WB\_bus   (MEM\_WB\_bus   ),  // O, 118            //5级流水新增接口          .MEM\_allow\_in (MEM\_allow\_in ),  // I, 1          .MEM\_wdest    (MEM\_wdest    ),  // O, 5            //展示PC          .MEM\_pc       (MEM\_pc       )   // O, 32      );        wb WB\_module(                     // 写回级          .WB\_valid    (WB\_valid    ),  // I, 1          .MEM\_WB\_bus\_r(MEM\_WB\_bus\_r),  // I, 118          .rf\_wen      (rf\_wen      ),  // O, 1          .rf\_wdest    (rf\_wdest    ),  // O, 5          .rf\_wdata    (rf\_wdata    ),  // O, 32            .WB\_over     (WB\_over     ),  // O, 1            //5级流水新增接口          .clk         (clk         ),  // I, 1        .resetn      (resetn      ),  // I, 1          .exc\_bus     (exc\_bus     ),  // O, 32          .WB\_wdest    (WB\_wdest    ),  // O, 5          .cancel      (cancel      ),  // O, 1            //展示PC和HI/LO值          .WB\_pc       (WB\_pc       ),  // O, 32          .HI\_data     (HI\_data     ),  // O, 32          .LO\_data     (LO\_data     )   // O, 32      );      inst\_rom inst\_rom\_module(         // 指令存储器          .clka       (clk           ),  // I, 1 ,时钟          .addra      (inst\_addr[9:2]),  // I, 8 ,指令地址          .douta      (inst          )   // O, 32,指令      );      regfile rf\_module(        // 寄存器堆模块          .clk    (clk      ),  // I, 1          .wen    (rf\_wen   ),  // I, 1          .raddr1 (rs       ),  // I, 5          .raddr2 (rt       ),  // I, 5          .waddr  (rf\_wdest ),  // I, 5          .wdata  (rf\_wdata ),  // I, 32          .rdata1 (rs\_value ),  // O, 32          .rdata2 (rt\_value ),  // O, 32          //display rf          .test\_addr(rf\_addr),  // I, 5          .test\_data(rf\_data)   // O, 32      );        data\_ram data\_ram\_module(   // 数据存储模块          .clka   (clk         ),  // I, 1,  时钟          .wea    (dm\_wen      ),  // I, 1,  写使能          .addra  (dm\_addr[9:2]),  // I, 8,  读地址          .dina   (dm\_wdata    ),  // I, 32, 写数据          .douta  (dm\_rdata    ),  // O, 32, 读数据          //display mem          .clkb   (clk          ),  // I, 1,  时钟          .web    (4'd0         ),  // 不使用端口2的写功能          .addrb  (mem\_addr[9:2]),  // I, 8,  读地址          .doutb  (mem\_data     ),  // I, 32, 写数据          .dinb   (32'd0        )   // 不使用端口2的写功能      );  //--------------------------{各模块实例化}end----------------------------//  endmodule |  四、实验实现截图五、收获心得体会 经过各模块和整体程序的仿真，达到了设计的要求。MIPS 五级流水线的设计思路基本实现，从程序测试指令的译码（从文本文件中读入）到指令的译码、执行，从仿真图上可以清楚的的值数据运算的准确性和有效性，同时具备一定的异常处理。使得整个程序具有很好的冗错性。指令完成执行阶段后进入访存和回写阶段，从最后的运行状态来看未发现明显异常状况。  在五级流水线的设计过程中遇到的最具代表性的问题就是数据相关性的处理，所谓数据相关性指的是在流水线中执行的几条指令中，一条指令的执行依赖前一条或几条指令的执行结果，通过查阅资料等途径，本程序中采用的解决措施：在 MIPS 中利用前推的方法解决流水线中数据相关性问题，主要是将执行阶段的结，访存阶段的结果前推到译码阶段。 具体来看就是：  1）将处于流水线执行阶段的指令的运算结果，包括：是否需要写入目的寄存器 wreg\_o、要写入的目的寄存器的地址 wd\_o,要写入目的寄存器的数据 wdata\_o 等信息送到译码阶段。  2）将处于流水线访存阶段的指令的运算结果，包括：是否要写入目的寄存器 wreg\_o，要写入的目的寄存器的地址 wd\_o,要写入目的寄存器的数据 wdata\_o 等信息送到译码阶段。  在此次的数字钟设计过程中，我更进一步地熟悉有关数字电路的知识和具体应用。硬件描述语言 verilog 的编写，程序的仿真等工作。并能根据仿真结果分析设计的存在的问题和缺陷，从而进行程序的调试和完善,同时，虽然自己未承担应用程序程序的开发任务，但是也学习不少相关方面的知识，以及下板过程中一些常见的问题，学会了如何使用 vivado。 |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 实验九 优化CPU系统一、实验分析说明 1 . 加深对计算机组成原理和体系结构理论知识的理解 。  2 . 培养对 CPU 设计的兴趣 ，在理解现有 CPU 架构的基础上 ， 引发对体系结构的思 考和创新 。  3 . 培养创新思维能力 ，并通过实践验证新想法 。 二、实验设计思路 实验9是对之前课程设计的拔高 ，是为大家在学有余力的时候设定的 ， 旨在发挥 大家的创新思维能力 ，为大家提供一个学习反思验证的环境 ，可以尽情地提出自己关 于计算机组成原理和体系结构的想法 ，大胆地对书中的知识提出疑问 ，通过设计实验 方案验证自己的想法 ，从而获得对课本知识的升华 。  1 . 分析静态 5 级流水 CPU 中的流水线阻塞情况 ，包括数据相关 、控制相关 、结构 相关等 ，优化流水线设计 ，尽可能减少流水线阻塞情况 ， 比如前递技术等 。  2 . 对于分支跳转指令 ，mips 架构中有延迟槽指令的设定 ，利用这一点 ，在静态 5 级流水 CPU 中 ，可实现分支指令永远不阻塞后续指令 ，大家可以检查自己的流水线设 定 ，进行优化实现这一点 。  3 . 针对第 2 点 ，此时 ，分支跳转指令就不需要进行转移猜测了 ，但大家可以将流 水线结构改为 x86 中无延迟槽技术的设定 ，此时分支跳转指令与后续真正需要执行的 指令至少会堵塞一拍 ，此时可以考虑实现转移预测技术 ，提升流水线结构 。  4 . 在学习的过程中 ，大家一定会有很多自己的想法 ， 比如 ，为什么取指 、译码 、 执行 、访存 、写回称为经典的 5 级流水结构 ，可以实现 3 级 、4 级 、6 级流水结构的 CPU 吗?答案肯定是可以的 ，甚至在各类产品的 CPU 中采用经典 5 级流水结构的都很 少 。所以希望大家尽情地发挥自己的想法 ，大改流水结构 ，验证自己关于流水结构的 想法 ，您可以实现 3 级（比如 ：取指 、译码 、执行） 、4 级 、6 级等等各类流水结构的 CPU 。  5 . 课程设计要求大家实现的 mips 指令有限 ，大家可以分析其余 mips 指令 ，加以  实现 。  6 . 目前课程设计实现的指令存储器和数据存储器是同步读的机制的 ，故在当前拍 数（时钟周期）发出读数据的地址请求时 ，在下一拍才能获得读的数据 ， 因此取值级 和访存级的 load 都需要两拍时间 。其实发地址请求在下一拍获得读的数据 ， 明显也是 一个可以流水做的工作 ，故可以考虑对流水线设计方案作稍微修改 ，使得取指级和访 存级的 load 不需要多等一拍 。 三、实验源代码define.v  |  | | --- | | //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //   > 文件名: define.v  //   > 作者  : 崔金泽 苗春雨 李子怡  //   > 日期  : 2022-10-04  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //inside\_op 处理器内部指令识别编码  // arithmetic 算术  `define ADD   6'd0  `define ADDI  6'd1  `define ADDU  6'd2  `define ADDIU 6'd3  `define SUB   6'd4  `define SUBU  6'd5  `define SLT   6'd6  `define SLTI  6'd7  `define SLTU  6'd8  `define SLTIU 6'd9  // div 除法（通过hilo寄存器输出数据）  `define DIV   6'd10  `define DIVU  6'd11  // mul 乘法  `define MULT  6'd12  `define MULTU 6'd13  // logic 逻辑  `define AND   6'd14  `define ANDI  6'd15  `define LUI   6'd16  `define NOR   6'd17  `define OR    6'd18  `define ORI   6'd19  `define XOR   6'd20  `define XORI  6'd21  // shift 移位  `define SLLV  6'd22  `define SLL   6'd23  `define SRAV  6'd24  `define SRA   6'd25  `define SRLV  6'd26  `define SRL   6'd27  // branch & jump 分支跳转  `define BEQ   6'd28  `define BNE   6'd29  `define BGEZ  6'd30  `define BGTZ  6'd31  `define BLEZ  6'd32  `define BLTZ  6'd33  `define BGEZAL  6'd34  `define BLTZAL  6'd35  `define J     6'd36  `define JAL   6'd37  `define JR    6'd38  `define JALR  6'd39  // move 数据移动  `define MFHI  6'd40  `define MFLO  6'd41  `define MTHI  6'd42  `define MTLO  6'd43  // 自陷指令  `define BREAK 6'd44  `define SYSCALL 6'd45  // load & store 访存  `define LB    6'd46  `define LBU   6'd47  `define LH    6'd48  `define LHU   6'd49  `define LW    6'd50  `define SB    6'd51  `define SH    6'd52  `define SW    6'd53  // special 特权  `define ERET  6'd54  `define MFC0  6'd55  `define MTC0  6'd56  // NOP 空指令  `define NOP 6'd57  // MUL  `define MUL 6'd58  // `define 6'd59  // `define 6'd60  // `define 6'd61  // `define 6'd62  // `define 6'd63  // `define 6'd64  //AluSel  `define EXE\_NOP 3'b000  `define EXE\_LOGIC 3'b001  `define EXE\_SHIFT 3'b010  `define EXE\_MOVE 3'b011  `define EXE\_ARITHMETIC 3'b100  `define EXE\_MUL 3'b101  `define EXE\_JUMP\_BRANCH 3'b110  `define EXE\_LOAD\_STORE 3'b111    // 全局  `define RstEnable 1'b1  `define RstDisable 1'b0  `define ZeroWord 32'h00000000  `define WriteEnable 1'b1  `define WriteDisable 1'b0  `define ReadEnable 1'b1  `define ReadDisable 1'b0  `define AluOpBus 5:0  `define AluSelBus 2:0  `define InstValid 1'b0  `define InstInvalid 1'b1  `define Stop 1'b1  `define NoStop 1'b0  `define InDelaySlot 1'b1  `define NotInDelaySlot 1'b0  `define Branch 1'b1  `define NotBranch 1'b0  `define InterruptAssert 1'b1  `define InterruptNotAssert 1'b0  `define TrapAssert 1'b1  `define TrapNotAssert 1'b0  `define True\_v 1'b1  `define False\_v 1'b0  `define ChipEnable 1'b1  `define ChipDisable 1'b0  `define Cache 1'b1  `define UnCache 1'b0  `define StallBus 8:0  //指令存储器inst\_rom  `define InstAddrBus 31:0  `define InstBus 31:0  `define InstMemNum 131071  `define InstMemNumLog2 17  //数据存储器data\_ram  `define DataAddrBus 31:0  `define DataBus 31:0  `define DataMemNum 131071  `define DataMemNumLog2 17  `define ByteWidth 7:0  //通用寄存器regfile  `define RegAddrBus 4:0  `define RegBus 31:0  `define RegWidth 32  `define DoubleRegWidth 64  `define DoubleRegBus 63:0  `define RegNum 32  `define RegNumLog2 5  `define NOPRegAddr 5'b00000  //除法div  `define DivFree 2'b00  `define DivByZero 2'b01  `define DivOn 2'b10  `define DivEnd 2'b11  `define DivResultReady 1'b1  `define DivResultNotReady 1'b0  `define DivStart 1'b1  `define DivStop 1'b0  //axi 状态机  `define TEST1 4'd0  `define TEST2 4'd1  `define TEST3 4'd2  `define READ1 4'd3  `define READ2 4'd4  `define READ3 4'd5  `define READ4 4'd6  `define READ5 4'd7  `define READ6 4'd8  `define READ7 4'd9  `define WRITE1 4'd10  `define WRITE2 4'd11  `define WRITE3 4'd12  `define WRITE4 4'd13  `define WRITE5 4'd14  //CP0寄存器地址  `define CP0\_REG\_COUNT    5'b01001        //可读写  `define CP0\_REG\_COMPARE    5'b01011      //可读写  `define CP0\_REG\_STATUS    5'b01100       //可读写  `define CP0\_REG\_CAUSE    5'b01101        //只读  `define CP0\_REG\_EPC    5'b01110          //可读写  `define CP0\_REG\_CONFIG    5'b10000       //只读  `define CP0\_REG\_BADADDR    5'b01000 |  mycpu\_top  |  | | --- | | `include "defines.v"  module mycpu\_top(    input wire clk,    input wire resetn,    input wire [5:0] int,      output wire inst\_sram\_en,    output wire [3:0] inst\_sram\_wen,    output wire [31:0] inst\_sram\_addr,    output wire [31:0] inst\_sram\_wdata,    input wire [31:0] inst\_sram\_rdata,      output wire data\_sram\_en,    output wire [3:0] data\_sram\_wen,    output wire [31:0] data\_sram\_addr,    output wire [31:0] data\_sram\_wdata,    input wire [31:0] data\_sram\_rdata,      output wire [31:0] debug\_wb\_pc,    output wire [3:0] debug\_wb\_rf\_wen,    output wire [4:0] debug\_wb\_rf\_wnum,    output wire [31:0] debug\_wb\_rf\_wdata  );    wire rst;    assign rst = ~resetn;  // pc - pre\_icache    wire [`InstAddrBus] pc\_pc\_o;    wire pc\_ce\_o;  // pre\_icache    wire [`InstAddrBus] icache\_pc\_i;    wire icache\_ce\_i;  // icache - if\_id    wire [`InstAddrBus] if\_pc\_o;    wire [`InstBus] if\_inst\_o;  // if\_id - decoder    wire [`InstAddrBus] id\_pc\_i;    wire [`InstBus] id\_inst\_i;  // decoder - id\_ex    wire [`AluOpBus] id\_aluop\_o;    wire [`AluSelBus] id\_alusel\_o;    wire [`RegBus] id\_reg1\_o;    wire [`RegBus] id\_reg2\_o;    wire [`RegAddrBus] id\_waddr\_o;    wire id\_wreg\_o;    wire [`RegBus] id\_inst\_o;    wire [`InstAddrBus] id\_pc\_o;  // decoder - regfile    wire reg1\_read;    wire reg2\_read;    wire [`RegBus] reg1\_data;    wire [`RegBus] reg2\_data;    wire [`RegAddrBus] reg1\_addr;    wire [`RegAddrBus] reg2\_addr;  // id\_ex - ex    wire [`AluOpBus] ex\_aluop\_i;    wire [`AluSelBus] ex\_alusel\_i;    wire [`RegBus] ex\_reg1\_i;    wire [`RegBus] ex\_reg2\_i;    wire [`RegAddrBus] ex\_waddr\_i;    wire ex\_wreg\_i;    wire [`RegBus] ex\_inst\_i;    wire [`InstAddrBus] ex\_pc\_i;  // ex - ex\_premem    wire [`RegAddrBus] ex\_waddr\_o;    wire ex\_wreg\_o;    wire [`RegBus] ex\_wdata\_o;    wire [`AluOpBus] ex\_aluop\_o;    wire [`RegBus] ex\_mem\_addr\_o;    wire [`RegBus] ex\_reg2\_o;    wire [`RegBus] ex\_hi\_o;    wire [`RegBus] ex\_lo\_o;    wire ex\_whilo\_o;    wire ex\_cp0\_reg\_we\_o;    wire [4:0] ex\_cp0\_reg\_write\_addr\_o;    wire [`RegBus] ex\_cp0\_reg\_data\_o;    wire [`InstAddrBus] ex\_pc\_o;  // ex\_premem - premem    wire [`AluOpBus] premem\_aluop;    wire [`RegBus] premem\_mem\_addr\_i;    wire [`RegBus] premem\_reg2\_i;    wire [`InstAddrBus] premem\_pc\_i;  // ex\_premem - pre\_dcache    wire [`RegAddrBus] premem\_waddr;    wire premem\_wreg;    wire [`RegBus] premem\_wdata;    wire [`RegBus] premem\_hi;    wire [`RegBus] premem\_lo;    wire premem\_whilo;    wire premem\_cp0\_reg\_we;    wire [4:0] premem\_cp0\_reg\_write\_addr;    wire [`RegBus] premem\_cp0\_reg\_data;  // premem - pre\_dcache    // wire [`RegAddrBus] premem\_waddr\_o;    // wire premem\_wreg\_o;    // wire [`RegBus] premem\_wdata\_o;    // wire [`RegBus] premem\_hi\_o;    // wire [`RegBus] premem\_lo\_o;    // wire premem\_whilo\_o;    // wire [`AluOpBus] premem\_aluop\_o;    wire premem\_we\_o;    wire [3:0] premem\_sel\_o;    wire [`RegBus] premem\_data\_o;    wire premem\_ce\_o;    wire [`InstAddrBus] premem\_pc\_o;  // premem - mmu    wire [`RegBus] premem\_addr\_un\_mmu;    wire [`RegBus] premem\_addr\_mmu;  // mmu - pre\_dcache    wire mmu\_cache\_o;  // pre\_dcache - dcache    wire [`RegBus] dcache\_addr\_i;    wire dcache\_we\_i;    wire [3:0] dcache\_sel\_i;    wire [`RegBus] dcache\_data\_i;    wire dcache\_ce\_i;    wire dcache\_cache\_i;  // dcache - dcache\_mem    wire [`RegBus] dcache\_data\_o;  // pre\_dcache (- dcache) - dcache\_mem    wire [`RegAddrBus] dcache\_waddr;    wire dcache\_wreg;    wire [`RegBus] dcache\_wdata;    wire [`RegBus] dcache\_hi;    wire [`RegBus] dcache\_lo;    wire dcache\_whilo;    wire [`AluOpBus] dcache\_aluop;    wire [`RegBus] dcache\_addr;    wire dcache\_cp0\_reg\_we;    wire [4:0] dcache\_cp0\_reg\_write\_addr;    wire [`RegBus] dcache\_cp0\_reg\_data;      wire [`RegBus] dcache\_pc;  // dcache\_mem - mem    wire [`RegAddrBus] mem\_waddr\_i;    wire mem\_wreg\_i;    wire [`RegBus] mem\_wdata\_i;    wire [`RegBus] mem\_hi\_i;    wire [`RegBus] mem\_lo\_i;    wire mem\_whilo\_i;    wire [`AluOpBus] mem\_aluop\_i;    wire [`RegBus] mem\_addr\_i;    wire [`RegBus] mem\_data\_i;    wire [`RegBus] mem\_pc\_i;  // dcache\_mem - mem\_wb    wire mem\_cp0\_reg\_we;    wire [4:0] mem\_cp0\_reg\_write\_addr;    wire [`RegBus] mem\_cp0\_reg\_data;  // mem - mem\_wb    wire [`RegAddrBus] mem\_waddr\_o;    wire mem\_wreg\_o;    wire [`RegBus] mem\_wdata\_o;    wire [`RegBus] mem\_hi\_o;    wire [`RegBus] mem\_lo\_o;    wire mem\_whilo\_o;    wire [`RegBus] mem\_pc\_o;  // mem\_wb - regfile    wire [`RegAddrBus] wb\_waddr\_i;    wire wb\_wreg\_i;    wire [`RegBus] wb\_wdata\_i;  // hilo\_reg - ex    wire [`RegBus] hi\_i;    wire [`RegBus] lo\_i;  // mem\_wb - HILO    wire [`RegBus] wb\_hi\_i;    wire [`RegBus] wb\_lo\_i;    wire wb\_whilo\_i;  // mem\_wb - cp0\_reg    wire cp0\_reg\_we\_i;    wire [4:0] cp0\_reg\_write\_addr\_i;    wire [`RegBus] cp0\_reg\_data\_i;  // cp0\_reg - ex    wire [`RegBus] cp0\_reg\_data\_o;  // ex - cp0\_reg    wire [4:0] cp0\_reg\_read\_addr\_i;  // stall    wire [`StallBus] stall;    wire stallreq\_from\_id;    wire stallreq\_from\_ex;    wire stallreq\_from\_icache;    wire stallreq\_from\_dcache;  // excepttype    wire flush;    wire [`RegBus] new\_pc;    wire [31:0] pc\_excepttype\_o;    wire [31:0] icache\_excepttype;    wire [31:0] id\_excepttype\_i;    wire [31:0] id\_excepttype\_o;    wire [31:0] ex\_excepttype\_i;    wire [31:0] ex\_excepttype\_o;    wire ex\_is\_in\_delayslot\_o;    wire [`RegBus] ex\_badvaddr\_o;    wire [31:0] premem\_excepttype;    wire premem\_is\_in\_delayslot;    wire [`RegBus] premem\_badvaddr;    wire [31:0] dcache\_excepttype;    wire dcache\_is\_in\_delayslot;    wire [`RegBus] dcache\_badvaddr;    wire [31:0] mem\_excepttype\_i;    wire mem\_is\_in\_delayslot\_i;    wire [`RegBus] mem\_badvaddr\_i;    wire [31:0] mem\_excepttype\_o;    wire [`RegBus] mem\_cp0\_epc\_o;    wire mem\_is\_in\_delayslot\_o;    wire [`RegBus] mem\_badvaddr\_o;    wire [`RegBus] mem\_cp0\_status\_i;    wire [`RegBus] mem\_cp0\_cause\_i;    wire [`RegBus] mem\_cp0\_epc\_i;  // ex - mul    wire mul\_start;    wire [`RegBus] mul\_op1;    wire [`RegBus] mul\_op2;    wire mul\_signed;    wire [`DoubleRegBus] result\_mul;    wire result\_is\_ok;  // ex - div    wire div\_start;      wire[`RegBus] div\_op1;      wire[`RegBus] div\_op2;      wire div\_signed;      wire[`DoubleRegBus] div\_result;      wire div\_ready;  // jump & branch    wire id\_pc\_branch\_flag;                     // id -> pc    wire [`RegBus] id\_pc\_branch\_target\_address; // id -> pc    wire next\_inst\_in\_delayslot;                // id -> id\_ex    wire [`RegBus] id\_link\_address\_o;           // id -> id\_ex    wire id\_is\_in\_delayslot\_o;                  // id -> id\_ex    wire id\_is\_in\_delayslot\_i;                  // id\_ex -> id    wire [`RegBus] ex\_link\_addrss\_i;            // id\_ex -> ex    wire ex\_is\_in\_delayslot\_i;                  // id\_ex -> ex  // axi & cache    // icache to axi    wire [`InstAddrBus] icache\_pc\_o;    wire isMiss\_from\_icache;    // axi to icache    wire icache\_we\_w;    wire [`InstAddrBus] icache\_pc\_w;    wire [`InstBus] icache\_inst\_w;    wire last\_for\_icache;    // dcache to axi    wire [`RegBus] dcache\_mem\_addr\_i;    wire dcache\_mem\_we\_i;    wire [3:0] dcache\_mem\_sel\_i;    wire [`RegBus] dcache\_mem\_data\_i;    wire dcache\_mem\_ce\_i;    wire isMiss\_from\_dcache;    wire dcache\_cache\_o;    // axi to dcache    wire [`RegBus] dcache\_mem\_addr\_w;    wire dcache\_mem\_we\_w;    wire [`RegBus] dcache\_mem\_data\_w;    wire dcache\_cache\_w;    wire dcache\_last\_w;  // debug    assign debug\_wb\_rf\_wen = {4{wb\_wreg\_i}};    assign debug\_wb\_rf\_wnum = wb\_waddr\_i;    assign debug\_wb\_rf\_wdata = wb\_wdata\_i;  // LINE2    wire l2\_pc\_plus\_4\_req;    // l2: icache - pre\_decoder      wire [31:0] l1\_icache\_decoder\_pc;      wire [31:0] l1\_icache\_decoder\_inst;      wire l1\_is\_ok;      wire [31:0] l2\_icache\_decoder\_pc;      wire [31:0] l2\_icache\_decoder\_inst;      wire l2\_is\_ok;    // l2: pre\_decoder - if\_id      wire [`AluSelBus] l1\_decoder\_alusel\_o;      wire [`RegAddrBus] l1\_decoder\_waddr\_o;      wire l1\_decoder\_wreg\_o;      wire [`InstAddrBus] l2\_decoder\_pc;      wire l2\_decoder\_reg1\_read;      wire l2\_decoder\_reg2\_read;      wire [`RegAddrBus] l2\_decoder\_reg1\_addr;      wire [`RegAddrBus] l2\_decoder\_reg2\_addr;      wire [`AluOpBus] l2\_decoder\_aluop;      wire [`AluSelBus] l2\_decoder\_alusel;      wire [`RegAddrBus] l2\_decoder\_waddr;      wire l2\_decoder\_wreg;      wire [`RegBus] l2\_decoder\_imm;      wire [`InstAddrBus] l2\_decoder\_pc\_o;      wire l2\_decoder\_reg1\_read\_o;      wire l2\_decoder\_reg2\_read\_o;      wire [`RegAddrBus] l2\_decoder\_reg1\_addr\_o;      wire [`RegAddrBus] l2\_decoder\_reg2\_addr\_o;      wire [`AluOpBus] l2\_decoder\_aluop\_o;      wire [`AluSelBus] l2\_decoder\_alusel\_o;      wire [`RegAddrBus] l2\_decoder\_waddr\_o;      wire l2\_decoder\_wreg\_o;      wire [`RegBus] l2\_decoder\_imm\_o;      wire l1\_run;      wire l2\_run;    // l2: if\_id - id\_ex      wire [`InstAddrBus] l2\_regfile\_pc;      wire l2\_regfile\_reg1\_read;      wire l2\_regfile\_reg2\_read;      wire [`RegAddrBus] l2\_regfile\_reg1\_addr;      wire [`RegAddrBus] l2\_regfile\_reg2\_addr;      wire [`AluOpBus] l2\_regfile\_aluop;      wire [`AluSelBus] l2\_regfile\_alusel;      wire [`RegAddrBus] l2\_regfile\_waddr;      wire l2\_regfile\_wreg;      wire [`RegBus] l2\_regfile\_imm;    // l2: regfile - id\_ex      wire [`RegBus] l2\_regfile\_reg1\_rdata;      wire [`RegBus] l2\_regfile\_reg2\_rdata;    // l2: id\_ex - ex      wire [`InstAddrBus] l2\_ex\_pc;      wire l2\_ex\_reg1\_read;      wire l2\_ex\_reg2\_read;      wire [`RegAddrBus] l2\_ex\_reg1\_addr;      wire [`RegAddrBus] l2\_ex\_reg2\_addr;      wire [`AluOpBus] l2\_ex\_aluop;      wire [`AluSelBus] l2\_ex\_alusel;      wire [`RegAddrBus] l2\_ex\_waddr;      wire l2\_ex\_wreg;      wire [`RegBus] l2\_ex\_imm;      wire [`RegBus] l2\_ex\_reg1\_rdata;      wire [`RegBus] l2\_ex\_reg2\_rdata;    // l2: ex - ex\_dcache      wire [`InstAddrBus] l2\_ex\_pc\_o;      wire [`RegAddrBus] l2\_ex\_waddr\_o;      wire l2\_ex\_wreg\_o;      wire [`RegBus] l2\_ex\_wdata\_o;    // l2: ex\_dcache - dcache\_mem      wire [`InstAddrBus] l2\_dcache\_pc;      wire [`RegAddrBus] l2\_dcache\_waddr;      wire l2\_dcache\_wreg;      wire [`RegBus] l2\_dcache\_wdata;    // l2: dcache\_mem - mem\_wb      wire [`InstAddrBus] l2\_mem\_pc;      wire [`RegAddrBus] l2\_mem\_waddr;      wire l2\_mem\_wreg;      wire [`RegBus] l2\_mem\_wdata;    // l2: wb      wire [`InstAddrBus] l2\_wb\_pc;      wire [`RegAddrBus] l2\_wb\_waddr;      wire l2\_wb\_wreg;      wire [`RegBus] l2\_wb\_wdata;        pc u\_pc(      .clk(clk), .rst(rst), .stall(stall),      .flush(flush), .new\_pc(new\_pc),      .branch\_flag\_i(id\_pc\_branch\_flag), .branch\_target\_address\_i(id\_pc\_branch\_target\_address),      .l2\_pc\_plus\_4\_req(l2\_pc\_plus\_4\_req),      .pc(pc\_pc\_o), .ce(pc\_ce\_o),      .excepttype\_o(pc\_excepttype\_o)    );    wire [`InstAddrBus] pc\_pc\_o\_with\_l2;    assign pc\_pc\_o\_with\_l2 = l2\_pc\_plus\_4\_req ? pc\_pc\_o + 32'h4 : pc\_pc\_o;    pre\_icache u\_pre\_icache(      .clk(clk), .rst(rst), .stall(stall), .flush(flush),      .new\_pc(new\_pc), .branch\_flag\_i(id\_pc\_branch\_flag), .branch\_target\_address\_i(id\_pc\_branch\_target\_address),      .pc\_pc(pc\_pc\_o\_with\_l2),      .icache\_pc(icache\_pc\_i),      .pc\_excepttype(pc\_excepttype\_o), .icache\_excepttype(icache\_excepttype)    );    line2\_icache u\_l2\_icache(      .clk(clk), .rst(rst),      .pc\_i(icache\_pc\_i), .inst\_i(inst\_sram\_rdata),      .l1\_pc\_o(l1\_icache\_decoder\_pc), .l1\_inst\_o(l1\_icache\_decoder\_inst), .l1\_is\_ok(l1\_is\_ok),      .l2\_pc\_o(l2\_icache\_decoder\_pc), .l2\_inst\_o(l2\_icache\_decoder\_inst), .l2\_is\_ok\_o(l2\_is\_ok)    );    line2\_pre\_decoder u\_l1\_pre\_decoder(      .rst(rst), .pc\_i(l1\_icache\_decoder\_pc), .inst\_i(l1\_icache\_decoder\_inst),      .pc\_o(), //.inst\_o(),      .reg1\_read\_o(), .reg2\_read\_o(),      .reg1\_addr\_o(), .reg2\_addr\_o(),      .aluop\_o(), .alusel\_o(l1\_decoder\_alusel\_o),      .waddr\_o(l1\_decoder\_waddr\_o), .wreg\_o(l1\_decoder\_wreg\_o),      .imm\_o(),      .l1\_run(l1\_run), .l2\_run()    );    line2\_pre\_decoder u\_l2\_pre\_decoder(      .rst(rst), .pc\_i(l2\_icache\_decoder\_pc), .inst\_i(l2\_icache\_decoder\_inst),      .pc\_o(l2\_decoder\_pc), //.inst\_o(),      .reg1\_read\_o(l2\_decoder\_reg1\_read), .reg2\_read\_o(l2\_decoder\_reg2\_read),      .reg1\_addr\_o(l2\_decoder\_reg1\_addr), .reg2\_addr\_o(l2\_decoder\_reg2\_addr),      .aluop\_o(l2\_decoder\_aluop), .alusel\_o(l2\_decoder\_alusel),      .waddr\_o(l2\_decoder\_waddr), .wreg\_o(l2\_decoder\_wreg),      .imm\_o(l2\_decoder\_imm),      .l1\_run(), .l2\_run(l2\_run)    );    // assign l2\_pc\_plus\_4\_req = `False\_v;    // & (dcache\_aluop != `LW) & (dcache\_aluop != `LB)    // & (ex\_aluop\_o != `LW) & (ex\_aluop\_o != `LB)    // assign l2\_pc\_plus\_4\_req = (id\_alusel\_o != `EXE\_JUMP\_BRANCH) & (id\_alusel\_o != `EXE\_LOAD\_STORE) & (l1\_run & l2\_run) & (l1\_is\_ok & l2\_is\_ok) ?    //                           (l1\_decoder\_wreg\_o ?    //                           ((l1\_decoder\_waddr\_o == l2\_decoder\_reg1\_addr) || (l1\_decoder\_waddr\_o == l2\_decoder\_reg2\_addr) ? `False\_v : `True\_v ) : `True\_v) : `False\_v;    assign l2\_pc\_plus\_4\_req = ~id\_pc\_branch\_flag & ~((id\_alusel\_o == `EXE\_LOAD\_STORE) & (id\_wreg\_o || id\_waddr\_o == l2\_decoder\_reg1\_addr || id\_waddr\_o == l2\_decoder\_reg2\_addr)) & (l1\_run & l2\_run) & (l1\_is\_ok & l2\_is\_ok) ?                              (l1\_decoder\_wreg\_o ?                              ((l1\_decoder\_waddr\_o == l2\_decoder\_reg1\_addr) || (l1\_decoder\_waddr\_o == l2\_decoder\_reg2\_addr) ? `False\_v : `True\_v ) : `True\_v) : `False\_v;    assign l2\_decoder\_pc\_o = l2\_pc\_plus\_4\_req ? l2\_decoder\_pc : `ZeroWord;    assign l2\_decoder\_reg1\_read\_o = l2\_pc\_plus\_4\_req ? l2\_decoder\_reg1\_read : `False\_v;    assign l2\_decoder\_reg2\_read\_o = l2\_pc\_plus\_4\_req ? l2\_decoder\_reg2\_read : `False\_v;    assign l2\_decoder\_reg1\_addr\_o = l2\_pc\_plus\_4\_req ? l2\_decoder\_reg1\_addr : `NOPRegAddr;    assign l2\_decoder\_reg2\_addr\_o = l2\_pc\_plus\_4\_req ? l2\_decoder\_reg2\_addr : `NOPRegAddr;    assign l2\_decoder\_aluop\_o = l2\_pc\_plus\_4\_req ? l2\_decoder\_aluop : `NOP;    assign l2\_decoder\_alusel\_o = l2\_pc\_plus\_4\_req ? l2\_decoder\_alusel : `EXE\_NOP;    assign l2\_decoder\_waddr\_o = l2\_pc\_plus\_4\_req ? l2\_decoder\_waddr : `NOPRegAddr;    assign l2\_decoder\_wreg\_o = l2\_pc\_plus\_4\_req ? l2\_decoder\_wreg : `WriteDisable;    assign l2\_decoder\_imm\_o = l2\_pc\_plus\_4\_req ? l2\_decoder\_imm : `ZeroWord;    line2\_if\_id u\_l2\_if\_id(      .clk(clk), .rst(rst), .stall(stall), .flush(flush),      .if\_pc(l2\_decoder\_pc\_o),      .if\_reg1\_read(l2\_decoder\_reg1\_read\_o), .if\_reg2\_read(l2\_decoder\_reg2\_read\_o),      .if\_reg1\_addr(l2\_decoder\_reg1\_addr\_o), .if\_reg2\_addr(l2\_decoder\_reg2\_addr\_o),      .if\_aluop(l2\_decoder\_aluop\_o), .if\_alusel(l2\_decoder\_alusel\_o),      .if\_waddr(l2\_decoder\_waddr\_o), .if\_wreg(l2\_decoder\_wreg\_o),      .if\_imm(l2\_decoder\_imm\_o),      .id\_pc(l2\_regfile\_pc),      .id\_reg1\_read(l2\_regfile\_reg1\_read), .id\_reg2\_read(l2\_regfile\_reg2\_read),      .id\_reg1\_addr(l2\_regfile\_reg1\_addr), .id\_reg2\_addr(l2\_regfile\_reg2\_addr),      .id\_aluop(l2\_regfile\_aluop), .id\_alusel(l2\_regfile\_alusel),      .id\_waddr(l2\_regfile\_waddr), .id\_wreg(l2\_regfile\_wreg),      .id\_imm(l2\_regfile\_imm)    );    wire [`RegBus] l2\_regfile\_reg1\_rdata\_o;    wire [`RegBus] l2\_regfile\_reg2\_rdata\_o;    wire dcache\_inst\_is\_load = ((dcache\_aluop == `LB) ||                                (dcache\_aluop == `LBU) ||                                (dcache\_aluop == `LH) ||                                (dcache\_aluop == `LHU) ||                                (dcache\_aluop == `LW)) ? `True\_v : `False\_v;    assign l2\_regfile\_reg1\_rdata\_o = (dcache\_inst\_is\_load == `True\_v) & (dcache\_waddr == l2\_regfile\_reg1\_addr) & l2\_regfile\_reg1\_read ? data\_sram\_rdata : l2\_regfile\_reg1\_rdata;    assign l2\_regfile\_reg2\_rdata\_o = (dcache\_inst\_is\_load == `True\_v) & (dcache\_waddr == l2\_regfile\_reg2\_addr) & l2\_regfile\_reg2\_read ? data\_sram\_rdata : l2\_regfile\_reg2\_rdata;    line2\_id\_ex u\_l2\_id\_ex(      .clk(clk), .rst(rst), .stall(stall), .flush(flush),      .id\_pc(l2\_regfile\_pc),      .id\_reg1\_read(l2\_regfile\_reg1\_read), .id\_reg2\_read(l2\_regfile\_reg2\_read),      .id\_reg1\_addr(l2\_regfile\_reg1\_addr), .id\_reg2\_addr(l2\_regfile\_reg2\_addr),      .id\_aluop(l2\_regfile\_aluop), .id\_alusel(l2\_regfile\_alusel),      .id\_waddr(l2\_regfile\_waddr), .id\_wreg(l2\_regfile\_wreg),      .id\_imm(l2\_regfile\_imm),      .id\_reg1\_rdata(l2\_regfile\_reg1\_rdata\_o), .id\_reg2\_rdata(l2\_regfile\_reg2\_rdata\_o),      .ex\_pc(l2\_ex\_pc),      .ex\_reg1\_read(l2\_ex\_reg1\_read), .ex\_reg2\_read(l2\_ex\_reg2\_read),      .ex\_reg1\_addr(l2\_ex\_reg1\_addr), .ex\_reg2\_addr(l2\_ex\_reg2\_addr),      .ex\_aluop(l2\_ex\_aluop), .ex\_alusel(l2\_ex\_alusel),      .ex\_waddr(l2\_ex\_waddr), .ex\_wreg(l2\_ex\_wreg),      .ex\_imm(l2\_ex\_imm),      .ex\_reg1\_rdata(l2\_ex\_reg1\_rdata), .ex\_reg2\_rdata(l2\_ex\_reg2\_rdata)    );    line2\_ex u\_l2\_ex(      .rst(rst), .pc\_i(l2\_ex\_pc), .pc\_o(l2\_ex\_pc\_o),      .reg1\_read\_i(l2\_ex\_reg1\_read), .reg2\_read\_i(l2\_ex\_reg2\_read),      .reg1\_addr\_i(l2\_ex\_reg1\_addr), .reg2\_addr\_i(l2\_ex\_reg2\_addr),      .aluop\_i(l2\_ex\_aluop), .alusel\_i(l2\_ex\_alusel),      .waddr\_i(l2\_ex\_waddr), .wreg\_i(l2\_ex\_wreg),      .imm\_i(l2\_ex\_imm),      .reg1\_rdata\_i(l2\_ex\_reg1\_rdata), .reg2\_rdata\_i(l2\_ex\_reg2\_rdata),      // .l1\_ex\_we(ex\_wreg\_o), .l1\_ex\_waddr(ex\_waddr\_o), .l1\_ex\_wdata(ex\_wdata\_o),      .waddr\_o(l2\_ex\_waddr\_o), .wreg\_o(l2\_ex\_wreg\_o), .wdata\_o(l2\_ex\_wdata\_o)    );    line2\_ex\_dcache u\_l2\_ex\_dcache(      .clk(clk), .rst(rst), .stall(stall), .flush(flush),      .ex\_pc(l2\_ex\_pc\_o), .ex\_wreg(l2\_ex\_wreg\_o), .ex\_waddr(l2\_ex\_waddr\_o), .ex\_wdata(l2\_ex\_wdata\_o),      .dcache\_pc(l2\_dcache\_pc), .dcache\_wreg(l2\_dcache\_wreg), .dcache\_waddr(l2\_dcache\_waddr), .dcache\_wdata(l2\_dcache\_wdata)    );    line2\_dcache\_mem u\_l2\_dcache\_mem(      .clk(clk), .rst(rst), .stall(stall), .flush(flush),      .dcache\_pc(l2\_dcache\_pc), .dcache\_wreg(l2\_dcache\_wreg), .dcache\_waddr(l2\_dcache\_waddr), .dcache\_wdata(l2\_dcache\_wdata),      .mem\_pc(l2\_mem\_pc), .mem\_wreg(l2\_mem\_wreg), .mem\_waddr(l2\_mem\_waddr), .mem\_wdata(l2\_mem\_wdata)    );    line2\_mem\_wb u\_l2\_mem\_wb(      .clk(clk), .rst(rst), .stall(stall), .flush(flush),      .mem\_pc(l2\_mem\_pc), .mem\_wreg(l2\_mem\_wreg), .mem\_waddr(l2\_mem\_waddr), .mem\_wdata(l2\_mem\_wdata),      .wb\_pc(l2\_wb\_pc), .wb\_wreg(l2\_wb\_wreg), .wb\_waddr(l2\_wb\_waddr), .wb\_wdata(l2\_wb\_wdata)    );    assign inst\_sram\_en = rst ? `False\_v : pc\_ce\_o;    assign inst\_sram\_wen = 4'b0000;    assign inst\_sram\_addr = rst ? `ZeroWord :                            flush ? new\_pc :                            id\_pc\_branch\_flag ? id\_pc\_branch\_target\_address :                            l2\_pc\_plus\_4\_req ? pc\_pc\_o + 32'h4 :                            pc\_pc\_o;    assign inst\_sram\_wdata = `ZeroWord;      if\_id u\_if\_id(      .clk(clk), .rst(rst), .stall(stall), .flush(flush),      .if\_pc(icache\_pc\_i), .if\_inst(inst\_sram\_rdata),      .id\_pc(id\_pc\_i), .id\_inst(id\_inst\_i),      .icache\_excepttype(icache\_excepttype), .id\_excepttype(id\_excepttype\_i)    );    decoder u\_decoder(      .rst(rst), .stallreq(stallreq\_from\_id),      .pc\_i(id\_pc\_i), .inst\_i(id\_inst\_i),      .pc\_o(id\_pc\_o), .inst\_o(id\_inst\_o),      .reg1\_read\_o(reg1\_read), .reg2\_read\_o(reg2\_read), .reg1\_addr\_o(reg1\_addr), .reg2\_addr\_o(reg2\_addr),      .reg1\_data\_i(reg1\_data), .reg2\_data\_i(reg2\_data),      .aluop\_o(id\_aluop\_o), .alusel\_o(id\_alusel\_o), .reg1\_o(id\_reg1\_o), .reg2\_o(id\_reg2\_o),      .waddr\_o(id\_waddr\_o), .wreg\_o(id\_wreg\_o),      .is\_in\_delayslot\_i(id\_is\_in\_delayslot\_i), .next\_inst\_in\_delayslot\_o(next\_inst\_in\_delayslot),      .branch\_flag\_o(id\_pc\_branch\_flag), .branch\_target\_address\_o(id\_pc\_branch\_target\_address),      .link\_addr\_o(id\_link\_address\_o), .is\_in\_delayslot\_o(id\_is\_in\_delayslot\_o),      .ex\_waddr\_i(ex\_waddr\_o), .ex\_aluop\_i(ex\_aluop\_o), .ex\_mem\_addr\_i(ex\_mem\_addr\_o),      // .premem\_waddr\_i(premem\_waddr), .premem\_aluop\_i(premem\_aluop),      .dcache\_waddr\_i(dcache\_waddr), .dcache\_aluop\_i(dcache\_aluop), .dcache\_wdata\_i(data\_sram\_rdata),      .excepttype\_i(id\_excepttype\_i), .excepttype\_o(id\_excepttype\_o)    );    regfile u\_regfile(      .clk(clk), .rst(rst),      .we(wb\_wreg\_i), .waddr(wb\_waddr\_i), .wdata(wb\_wdata\_i),      .l2\_we(l2\_wb\_wreg), .l2\_waddr(l2\_wb\_waddr), .l2\_wdata(l2\_wb\_wdata),      .re1(reg1\_read), .raddr1(reg1\_addr), .rdata1(reg1\_data),      .re2(reg2\_read), .raddr2(reg2\_addr), .rdata2(reg2\_data),      .ex\_forwarding\_we(ex\_wreg\_o), .ex\_forwarding\_waddr(ex\_waddr\_o), .ex\_forwarding\_wdata(ex\_wdata\_o),      .dcache\_forwarding\_we(dcache\_wreg), .dcache\_forwarding\_waddr(dcache\_waddr), .dcache\_forwarding\_wdata(dcache\_wdata),      .mem\_forwarding\_we(mem\_wreg\_o), .mem\_forwarding\_waddr(mem\_waddr\_o), .mem\_forwarding\_wdata(mem\_wdata\_o),      .l2\_re1(l2\_regfile\_reg1\_read), .l2\_raddr1(l2\_regfile\_reg1\_addr), .l2\_rdata1(l2\_regfile\_reg1\_rdata),      .l2\_re2(l2\_regfile\_reg2\_read), .l2\_raddr2(l2\_regfile\_reg2\_addr), .l2\_rdata2(l2\_regfile\_reg2\_rdata),      .l2\_ex\_forwarding\_we(l2\_ex\_wreg\_o), .l2\_ex\_forwarding\_waddr(l2\_ex\_waddr\_o), .l2\_ex\_forwarding\_wdata(l2\_ex\_wdata\_o),      .l2\_dcache\_forwarding\_we(l2\_dcache\_wreg), .l2\_dcache\_forwarding\_waddr(l2\_dcache\_waddr), .l2\_dcache\_forwarding\_wdata(l2\_dcache\_wdata),      .l2\_mem\_forwarding\_we(l2\_mem\_wreg), .l2\_mem\_forwarding\_waddr(l2\_mem\_waddr), .l2\_mem\_forwarding\_wdata(l2\_mem\_wdata)    );    id\_ex u\_id\_ex(      .clk(clk), .rst(rst), .stall(stall), .flush(flush),      .id\_pc(id\_pc\_o), .ex\_pc(ex\_pc\_i),      .id\_inst(id\_inst\_o), .ex\_inst(ex\_inst\_i),      .id\_aluop(id\_aluop\_o), .id\_alusel(id\_alusel\_o), .id\_reg1(id\_reg1\_o), .id\_reg2(id\_reg2\_o),      .id\_waddr(id\_waddr\_o), .id\_wreg(id\_wreg\_o),      .ex\_aluop(ex\_aluop\_i), .ex\_alusel(ex\_alusel\_i), .ex\_reg1(ex\_reg1\_i), .ex\_reg2(ex\_reg2\_i),      .ex\_waddr(ex\_waddr\_i), .ex\_wreg(ex\_wreg\_i),      .id\_link\_address(id\_link\_address\_o), .id\_is\_in\_delayslot(id\_is\_in\_delayslot\_o), .next\_inst\_in\_delayslot\_i(next\_inst\_in\_delayslot),      .ex\_link\_addrss(ex\_link\_addrss\_i), .ex\_is\_in\_delayslot(ex\_is\_in\_delayslot\_i), .is\_in\_delayslot\_o(id\_is\_in\_delayslot\_i),      .id\_excepttype(id\_excepttype\_o), .ex\_excepttype(ex\_excepttype\_i)    );    ex u\_ex(      .rst(rst), .stallreq(stallreq\_from\_ex),      .pc\_i(ex\_pc\_i), .pc\_o(ex\_pc\_o), .inst\_i(ex\_inst\_i),      .aluop\_i(ex\_aluop\_i), .alusel\_i(ex\_alusel\_i), .reg1\_i(ex\_reg1\_i), .reg2\_i(ex\_reg2\_i),      .waddr\_i(ex\_waddr\_i), .wreg\_i(ex\_wreg\_i),      .waddr\_o(ex\_waddr\_o), .wreg\_o(ex\_wreg\_o), .wdata\_o(ex\_wdata\_o),      .hi\_i(hi\_i), .lo\_i(lo\_i),      .wb\_hi\_i(wb\_hi\_i), .wb\_lo\_i(wb\_lo\_i), .wb\_whilo\_i(wb\_whilo\_i),      .mem\_hi\_i(mem\_hi\_o), .mem\_lo\_i(mem\_lo\_o), .mem\_whilo\_i(mem\_whilo\_o), // 这里按照雷思磊书上的使用的是mem之后的信号，个人感觉使用mem前的信号也没区别      .dcache\_hi\_i(dcache\_hi), .dcache\_lo\_i(dcache\_lo), .dcache\_whilo\_i(dcache\_whilo),      // .premem\_hi\_i(premem\_hi), .premem\_lo\_i(premem\_lo), .premem\_whilo\_i(premem\_whilo),      .hi\_o(ex\_hi\_o), .lo\_o(ex\_lo\_o), .whilo\_o(ex\_whilo\_o),      // .result\_mul(result\_mul), .result\_is\_ok(result\_is\_ok),      // .mul\_opdata1\_o(mul\_op1), .mul\_opdata2\_o(mul\_op2), .mul\_start\_o(mul\_start), .signed\_mul\_o(mul\_signed),      .div\_result\_i(div\_result), .div\_ready\_i(div\_ready),      .div\_opdata1\_o(div\_op1), .div\_opdata2\_o(div\_op2), .div\_start\_o(div\_start), .signed\_div\_o(div\_signed),      .link\_address\_i(ex\_link\_addrss\_i), .is\_in\_delayslot\_i(ex\_is\_in\_delayslot\_i),      .aluop\_o(ex\_aluop\_o), .mem\_addr\_o(ex\_mem\_addr\_o), .reg2\_o(ex\_reg2\_o),      // .premem\_cp0\_reg\_we(premem\_cp0\_reg\_we), .premem\_cp0\_reg\_write\_addr(premem\_cp0\_reg\_write\_addr), .premem\_cp0\_reg\_data(premem\_cp0\_reg\_data),      .dcache\_cp0\_reg\_we(dcache\_cp0\_reg\_we), .dcache\_cp0\_reg\_write\_addr(dcache\_cp0\_reg\_write\_addr), .dcache\_cp0\_reg\_data(dcache\_cp0\_reg\_data),      .mem\_cp0\_reg\_we(mem\_cp0\_reg\_we), .mem\_cp0\_reg\_write\_addr(mem\_cp0\_reg\_write\_addr), .mem\_cp0\_reg\_data(mem\_cp0\_reg\_data),      // .wb\_cp0\_reg\_we(cp0\_reg\_we\_i), .wb\_cp0\_reg\_write\_addr(cp0\_reg\_write\_addr\_i), .wb\_cp0\_reg\_data(cp0\_reg\_data\_i),      .cp0\_reg\_data\_i(cp0\_reg\_data\_o), .cp0\_reg\_read\_addr\_o(cp0\_reg\_read\_addr\_i),      .cp0\_reg\_we\_o(ex\_cp0\_reg\_we\_o), . cp0\_reg\_write\_addr\_o(ex\_cp0\_reg\_write\_addr\_o), .cp0\_reg\_data\_o(ex\_cp0\_reg\_data\_o),      .excepttype\_i(ex\_excepttype\_i), .excepttype\_o(ex\_excepttype\_o),      .is\_in\_delayslot\_o(ex\_is\_in\_delayslot\_o), .badvaddr\_o(ex\_badvaddr\_o)    );    reg [31:0] excepttype\_temp;    assign premem\_waddr = ex\_waddr\_o;    assign premem\_wreg = ex\_wreg\_o;    assign premem\_wdata = ex\_wdata\_o;    assign premem\_hi = ex\_hi\_o;    assign premem\_lo = ex\_lo\_o;    assign premem\_whilo = ex\_whilo\_o;    assign premem\_aluop = ex\_aluop\_o;    assign premem\_mem\_addr\_i = ex\_mem\_addr\_o;    assign premem\_reg2\_i = ex\_reg2\_o;    assign premem\_cp0\_reg\_we = ex\_cp0\_reg\_we\_o;    assign premem\_cp0\_reg\_write\_addr = ex\_cp0\_reg\_write\_addr\_o;    assign premem\_cp0\_reg\_data = ex\_cp0\_reg\_data\_o;    assign premem\_pc\_i = ex\_pc\_o;    assign premem\_excepttype = |excepttype\_temp ? excepttype\_temp : ex\_excepttype\_o;    assign premem\_is\_in\_delayslot = ex\_is\_in\_delayslot\_o;    assign premem\_badvaddr = ex\_badvaddr\_o;    always @ (posedge clk) begin      excepttype\_temp <= ex\_excepttype\_o;    end    pre\_mem u\_pre\_mem(      .rst(rst),      // .waddr\_i(premem\_waddr\_i), .wreg\_i(premem\_wreg\_i), .wdata\_i(premem\_wdata\_i),      // .waddr\_o(premem\_waddr\_o), .wreg\_o(premem\_wreg\_o), .wdata\_o(premem\_wdata\_o),      // .hi\_i(premem\_hi\_i), .lo\_i(premem\_lo\_i), .whilo\_i(premem\_whilo\_i),      // .hi\_o(premem\_hi\_o), .lo\_o(premem\_lo\_o), .whilo\_o(premem\_whilo\_o),      .aluop\_i(premem\_aluop), .mem\_addr\_i(premem\_mem\_addr\_i), .reg2\_i(premem\_reg2\_i),      // .aluop\_o(premem\_aluop\_o),      // .mem\_data\_i(dcache\_mem\_data),      .mem\_addr\_o(premem\_addr\_un\_mmu), .mem\_we\_o(premem\_we\_o), .mem\_sel\_o(premem\_sel\_o), .mem\_data\_o(premem\_data\_o), .mem\_ce\_o(premem\_ce\_o),      .pc\_i(premem\_pc\_i), .pc\_o(premem\_pc\_o),      .excepttype\_i(premem\_excepttype)    );    assign premem\_addr\_mmu = (premem\_addr\_un\_mmu < 32'h80000000) ? premem\_addr\_un\_mmu :                              (premem\_addr\_un\_mmu < 32'hA0000000) ? (premem\_addr\_un\_mmu - 32'h80000000) :                              (premem\_addr\_un\_mmu < 32'hC0000000) ? (premem\_addr\_un\_mmu - 32'hA0000000) :                              (premem\_addr\_un\_mmu < 32'hE0000000) ? (premem\_addr\_un\_mmu) :                              (premem\_addr\_un\_mmu <= 32'hFFFFFFFF) ? (premem\_addr\_un\_mmu) :                              32'h00000000;    pre\_dcache u\_pre\_dcache(      .clk(clk), .rst(rst), .stall(stall), .flush(flush),      .premem\_waddr(premem\_waddr), .premem\_wreg(premem\_wreg), .premem\_wdata(premem\_wdata),      .dcache\_waddr(dcache\_waddr), .dcache\_wreg(dcache\_wreg), .dcache\_wdata(dcache\_wdata),      .premem\_hi(premem\_hi), .premem\_lo(premem\_lo), .premem\_whilo(premem\_whilo),      .dcache\_hi(dcache\_hi), .dcache\_lo(dcache\_lo), .dcache\_whilo(dcache\_whilo),      .premem\_aluop(premem\_aluop), .dcache\_aluop(dcache\_aluop),      .premem\_addr(premem\_addr\_mmu), .premem\_we(premem\_we\_o), .premem\_sel(premem\_sel\_o), .premem\_data(premem\_data\_o), .premem\_ce(premem\_ce\_o), .premem\_cache(mmu\_cache\_o),      .dcache\_addr(dcache\_addr), .dcache\_we(dcache\_we\_i), .dcache\_sel(dcache\_sel\_i), .dcache\_data(dcache\_data\_i), .dcache\_ce(dcache\_ce\_i), .dcache\_cache(dcache\_cache\_i),      .premem\_cp0\_reg\_we(premem\_cp0\_reg\_we), .premem\_cp0\_reg\_write\_addr(premem\_cp0\_reg\_write\_addr), .premem\_cp0\_reg\_data(premem\_cp0\_reg\_data),      .dcache\_cp0\_reg\_we(dcache\_cp0\_reg\_we), .dcache\_cp0\_reg\_write\_addr(dcache\_cp0\_reg\_write\_addr), .dcache\_cp0\_reg\_data(dcache\_cp0\_reg\_data),      .premem\_pc(premem\_pc\_o), .dcache\_pc(dcache\_pc),      .premem\_excepttype(premem\_excepttype), .premem\_is\_in\_delayslot(premem\_is\_in\_delayslot), .premem\_badvaddr(premem\_badvaddr),      .dcache\_excepttype(dcache\_excepttype), .dcache\_is\_in\_delayslot(dcache\_is\_in\_delayslot), .dcache\_badvaddr(dcache\_badvaddr)    );    assign dcache\_addr\_i = dcache\_addr;    assign data\_sram\_en = rst ? `False\_v : premem\_ce\_o;    assign data\_sram\_wen = rst ? 4'b0000 : premem\_sel\_o & {4{premem\_we\_o}};    assign data\_sram\_addr = rst ? `ZeroWord : premem\_addr\_mmu;    assign data\_sram\_wdata = rst ? `ZeroWord : premem\_data\_o;    dcache\_mem u\_dcache\_mem(      .clk(clk), .rst(rst), .stall(stall), .flush(flush),      .dcache\_waddr(dcache\_waddr), .dcache\_wreg(dcache\_wreg), .dcache\_wdata(dcache\_wdata),      .mem\_waddr(mem\_waddr\_i), .mem\_wreg(mem\_wreg\_i), .mem\_wdata(mem\_wdata\_i),      .dcache\_hi(dcache\_hi), .dcache\_lo(dcache\_lo), .dcache\_whilo(dcache\_whilo),      .mem\_hi(mem\_hi\_i), .mem\_lo(mem\_lo\_i), .mem\_whilo(mem\_whilo\_i),      .dcache\_aluop(dcache\_aluop), .mem\_aluop(mem\_aluop\_i),      .dcache\_addr(dcache\_addr), .mem\_addr(mem\_addr\_i),      .dcache\_data(data\_sram\_rdata), .mem\_data(mem\_data\_i),      .dcache\_cp0\_reg\_we(dcache\_cp0\_reg\_we), .dcache\_cp0\_reg\_write\_addr(dcache\_cp0\_reg\_write\_addr), .dcache\_cp0\_reg\_data(dcache\_cp0\_reg\_data),      .mem\_cp0\_reg\_we(mem\_cp0\_reg\_we), .mem\_cp0\_reg\_write\_addr(mem\_cp0\_reg\_write\_addr), .mem\_cp0\_reg\_data(mem\_cp0\_reg\_data),      .dcache\_pc(dcache\_pc), .mem\_pc(mem\_pc\_i),      .dcache\_excepttype(dcache\_excepttype), .dcache\_is\_in\_delayslot(dcache\_is\_in\_delayslot), .dcache\_badvaddr(dcache\_badvaddr),      .mem\_excepttype(mem\_excepttype\_i), .mem\_is\_in\_delayslot(mem\_is\_in\_delayslot\_i), .mem\_badvaddr(mem\_badvaddr\_i)    );    mem u\_mem(      .rst(rst),      .waddr\_i(mem\_waddr\_i), .wreg\_i(mem\_wreg\_i), .wdata\_i(mem\_wdata\_i),      .waddr\_o(mem\_waddr\_o), .wreg\_o(mem\_wreg\_o), .wdata\_o(mem\_wdata\_o),      .hi\_i(mem\_hi\_i), .lo\_i(mem\_lo\_i), .whilo\_i(mem\_whilo\_i),      .hi\_o(mem\_hi\_o), .lo\_o(mem\_lo\_o), .whilo\_o(mem\_whilo\_o),      .aluop\_i(mem\_aluop\_i), .mem\_addr\_i(mem\_addr\_i), .mem\_data\_i(mem\_data\_i),      .pc\_i(mem\_pc\_i), .pc\_o(mem\_pc\_o),      .excepttype\_i(mem\_excepttype\_i), .is\_in\_delayslot\_i(mem\_is\_in\_delayslot\_i), .badvaddr\_i(mem\_badvaddr\_i),      .cp0\_status\_i(mem\_cp0\_status\_i), .cp0\_cause\_i(mem\_cp0\_cause\_i), .cp0\_epc\_i(mem\_cp0\_epc\_i),      // .wb\_cp0\_reg\_we(cp0\_reg\_we\_i), .wb\_cp0\_reg\_write\_addr(cp0\_reg\_write\_addr\_i), .wb\_cp0\_reg\_data(cp0\_reg\_data\_i),      .excepttype\_o(mem\_excepttype\_o), .cp0\_epc\_o(mem\_cp0\_epc\_o), .is\_in\_delayslot\_o(mem\_is\_in\_delayslot\_o), .badvaddr\_o(mem\_badvaddr\_o)    );    mem\_wb u\_mem\_wb(      .clk(clk), .rst(rst), .stall(stall), .flush(flush),      .mem\_waddr(mem\_waddr\_o), .mem\_wreg(mem\_wreg\_o), .mem\_wdata(mem\_wdata\_o),      .wb\_waddr(wb\_waddr\_i), .wb\_wreg(wb\_wreg\_i), .wb\_wdata(wb\_wdata\_i),      .mem\_hi(mem\_hi\_o), .mem\_lo(mem\_lo\_o), .mem\_whilo(mem\_whilo\_o),      .wb\_hi(wb\_hi\_i), .wb\_lo(wb\_lo\_i), .wb\_whilo(wb\_whilo\_i),      // .mem\_cp0\_reg\_we(mem\_cp0\_reg\_we), .mem\_cp0\_reg\_write\_addr(mem\_cp0\_reg\_write\_addr), .mem\_cp0\_reg\_data(mem\_cp0\_reg\_data),      // .wb\_cp0\_reg\_we(cp0\_reg\_we\_i), .wb\_cp0\_reg\_write\_addr(cp0\_reg\_write\_addr\_i), .wb\_cp0\_reg\_data(cp0\_reg\_data\_i),      .pc\_i(mem\_pc\_o), .pc\_o(debug\_wb\_pc)    );    hilo\_reg u\_hilo\_reg(      .clk(clk), .rst(rst),      .we(wb\_whilo\_i), .hi\_i(wb\_hi\_i), .lo\_i(wb\_lo\_i),      .hi\_o(hi\_i), .lo\_o(lo\_i)    );    cp0\_reg u\_cp0\_reg(      .clk(clk), .rst(rst),      .we\_i(mem\_cp0\_reg\_we), .waddr\_i(mem\_cp0\_reg\_write\_addr), .raddr\_i(cp0\_reg\_read\_addr\_i), .data\_i(mem\_cp0\_reg\_data),      .int\_i(int),      .data\_o(cp0\_reg\_data\_o), .badvaddr\_o(), .count\_o(), .compare\_o(),      .status\_o(mem\_cp0\_status\_i), .cause\_o(mem\_cp0\_cause\_i), .epc\_o(mem\_cp0\_epc\_i), .config\_o(),      .timer\_int\_o(),      .excepttype\_i(mem\_excepttype\_o), .pc\_i(mem\_pc\_o), .is\_in\_delayslot\_i(mem\_is\_in\_delayslot\_o), .badvaddr\_i(mem\_badvaddr\_o)    );    ctrl u\_ctrl(      .rst(rst),      .stallreq\_from\_id(stallreq\_from\_id),      .stallreq\_from\_ex(stallreq\_from\_ex),      .stallreq\_from\_icache(1'b0),      .stallreq\_from\_dcache(1'b0),      .excepttype\_i(mem\_excepttype\_o),      .cp0\_epc\_i(mem\_cp0\_epc\_o),      .new\_pc(new\_pc),      .flush(flush),      .stall(stall)    );    div u\_div(          .clk(clk),          .rst(rst),          .signed\_div\_i(div\_signed),          .opdata1\_i(div\_op1),          .opdata2\_i(div\_op2),          .start\_i(div\_start),          .annul\_i(1'b0),          .result\_o(div\_result),          .ready\_o(div\_ready)      );    // axi\_bus u\_axi\_bus(    //   .clk(clk), .rst(rst),    //   .pc\_i(icache\_pc\_o), .isMiss\_from\_icache(isMiss\_from\_icache),    //   .we\_icache\_o(icache\_we\_w), .pc\_icache\_o(icache\_pc\_w), .inst\_icache\_o(icache\_inst\_w), .last\_for\_icache(last\_for\_icache),    //   .mem\_addr\_i(dcache\_mem\_addr\_i), .mem\_we\_i(dcache\_mem\_we\_i), .mem\_sel\_i(dcache\_mem\_sel\_i), .mem\_data\_i(dcache\_mem\_data\_i), .mem\_ce\_i(dcache\_mem\_ce\_i), .isMiss\_from\_dcache(isMiss\_from\_dcache), .cache\_i(dcache\_cache\_o),    //   .mem\_addr\_o(dcache\_mem\_addr\_w), .mem\_we\_o(dcache\_mem\_we\_w), .mem\_data\_o(dcache\_mem\_data\_w), .cache\_o(dcache\_cache\_w), .last\_for\_dcache(dcache\_last\_w),    // //写地址通道信号    //   .awid(awid),//写地址ID，用来标志一组写信号    //   .awaddr(awaddr),//写地址，给出一次写突发传输的写地址    //   .awlen(awlen),//突发长度，给出突发传输的次数    //   .awsize(awsize),//突发大小，给出每次突发传输的字节数    //   .awburst(awburst),//突发类型    //   .awlock(awlock),//总线锁信号，可提供操作的原子性    //   .awcache(awcache),//内存类型，表明一次传输是怎样通过系统的    //   .awprot(awprot),//保护类型，表明一次传输的特权级及安全等级    //   .awvalid(awvalid),//有效信号，表明此通道的地址控制信号有效    //   .awready(awready),//表明"从"可以接收地址和对应的控制信号    // //写数据通道信号    //   .wid(wid),//一次写传输的ID tag    //   .wdata(wdata),//写数据    //   .wstrb(wstrb),//写数据有效的字节线，用来表明哪8bits数据是有效的    //   .wlast(wlast),//表明此次传输是最后一个突发传输    //   .wvalid(wvalid),//写有效，表明此次写有效    //   .wready(wready),//表明从机可以接收写数据    // //写响应通道信号    //   .bid(bid),//写响应ID tag    //   .bresp(bresp),//写响应，表明写传输的状态 00为正常，当然可以不理会    //   .bvalid(bvalid),//写响应有效    //   .bready(bready),//表明主机能够接收写响应    // //总线侧接口    // //读地址通道信号    //   .arid(arid),//读地址ID，用来标志一组写信号    //   .araddr(araddr),//读地址，给出一次写突发传输的读地址    //   .arlen(arlen),//突发长度，给出突发传输的次数    //   .arsize(arsize),//突发大小，给出每次突发传输的字节数    //   .arburst(arburst),//突发类型    //   .arlock(arlock),//总线锁信号，可提供操作的原子性    //   .arcache(arcache),//内存类型，表明一次传输是怎样通过系统的    //   .arprot(arprot),//保护类型，表明一次传输的特权级及安全等级    //   .arvalid(arvalid),//有效信号，表明此通道的地址控制信号有效    //   .arready(arready),//表明"从"可以接收地址和对应的控制信号    // //读数据通道信号    //   .rid(rid),//读ID tag    //   .rdata(rdata),//读数据    //   .rresp(rresp),//读响应，表明读传输的状态    //   .rlast(rlast),//表明读突发的最后一次传输    //   .rvalid(rvalid),//表明此通道信号有效    //   .rready(rready)//表明主机能够接收读数据和响应信息    // );  endmodule |  thinpad\_top  |  | | --- | | `default\_nettype none  module thinpad\_top(      input wire clk\_50M,           //50MHz 时钟输入      input wire clk\_11M0592,       //11.0592MHz 时钟输入（备用，可不用）      input wire clock\_btn,         //BTN5手动时钟按钮开关，带消抖电路，按下时为1      input wire reset\_btn,         //BTN6手动复位按钮开关，带消抖电路，按下时为1      input  wire[3:0]  touch\_btn,  //BTN1~BTN4，按钮开关，按下时为1      input  wire[31:0] dip\_sw,     //32位拨码开关，拨到“ON”时为1      output wire[15:0] leds,       //16位LED，输出时1点亮      output wire[7:0]  dpy0,       //数码管低位信号，包括小数点，输出1点亮      output wire[7:0]  dpy1,       //数码管高位信号，包括小数点，输出1点亮      //BaseRAM信号      inout wire[31:0] base\_ram\_data,  //BaseRAM数据，低8位与CPLD串口控制器共享      output wire[19:0] base\_ram\_addr, //BaseRAM地址      output wire[3:0] base\_ram\_be\_n,  //BaseRAM字节使能，低有效。如果不使用字节使能，请保持为0      output wire base\_ram\_ce\_n,       //BaseRAM片选，低有效      output wire base\_ram\_oe\_n,       //BaseRAM读使能，低有效      output wire base\_ram\_we\_n,       //BaseRAM写使能，低有效      //ExtRAM信号      inout wire[31:0] ext\_ram\_data,  //ExtRAM数据      output wire[19:0] ext\_ram\_addr, //ExtRAM地址      output wire[3:0] ext\_ram\_be\_n,  //ExtRAM字节使能，低有效。如果不使用字节使能，请保持为0      output wire ext\_ram\_ce\_n,       //ExtRAM片选，低有效      output wire ext\_ram\_oe\_n,       //ExtRAM读使能，低有效      output wire ext\_ram\_we\_n,       //ExtRAM写使能，低有效      //直连串口信号      output wire txd,  //直连串口发送端      input  wire rxd,  //直连串口接收端      //Flash存储器信号，参考 JS28F640 芯片手册      output wire [22:0]flash\_a,      //Flash地址，a0仅在8bit模式有效，16bit模式无意义      inout  wire [15:0]flash\_d,      //Flash数据      output wire flash\_rp\_n,         //Flash复位信号，低有效      output wire flash\_vpen,         //Flash写保护信号，低电平时不能擦除、烧写      output wire flash\_ce\_n,         //Flash片选信号，低有效      output wire flash\_oe\_n,         //Flash读使能信号，低有效      output wire flash\_we\_n,         //Flash写使能信号，低有效      output wire flash\_byte\_n,       //Flash 8bit模式选择，低有效。在使用flash的16位模式时请设为1      //图像输出信号      output wire[2:0] video\_red,    //红色像素，3位      output wire[2:0] video\_green,  //绿色像素，3位      output wire[1:0] video\_blue,   //蓝色像素，2位      output wire video\_hsync,       //行同步（水平同步）信号      output wire video\_vsync,       //场同步（垂直同步）信号      output wire video\_clk,         //像素时钟输出      output wire video\_de           //行数据有效信号，用于区分消隐区  );  /\* =========== Demo code begin =========== \*/  // PLL分频示例  wire locked, clk\_10M, clk\_20M;  pll\_example clock\_gen   (    // Clock in ports    .clk\_in1(clk\_50M),  // 外部时钟输入    // Clock out ports    .clk\_out1(clk\_10M), // 时钟输出1，频率在IP配置界面中设置 // 已改成50    .clk\_out2(clk\_20M), // 时钟输出2，频率在IP配置界面中设置    // Status and control signals    .reset(reset\_btn), // PLL复位输入    .locked(locked)    // PLL锁定指示输出，"1"表示时钟稳定，                       // 后级电路复位信号应当由它生成（见下）   );  reg reset\_of\_clk10M;  // 异步复位，同步释放，将locked信号转为后级电路的复位reset\_of\_clk10M  always@(posedge clk\_20M or negedge locked) begin      if(~locked) reset\_of\_clk10M <= 1'b1;      else        reset\_of\_clk10M <= 1'b0;  end  //cpu inst sram  wire        cpu\_inst\_en;  wire [3 :0] cpu\_inst\_wen;  wire [31:0] cpu\_inst\_addr;  wire [31:0] cpu\_inst\_wdata;  wire [31:0] cpu\_inst\_rdata;  //cpu data sram  wire        cpu\_data\_en;  wire [3 :0] cpu\_data\_wen;  wire [31:0] cpu\_data\_addr;  wire [31:0] cpu\_data\_wdata;  wire [31:0] cpu\_data\_rdata;  // cpu 里出来rst都是高位有效  // cpu 本身没有问题  mycpu\_top u\_mycpu(      .clk              (clk\_20M),      .resetn           (~reset\_of\_clk10M),      .int              (6'b0),      .inst\_sram\_en     (cpu\_inst\_en   ),//1      .inst\_sram\_wen    (cpu\_inst\_wen  ),//0000      .inst\_sram\_addr   (cpu\_inst\_addr ),      .inst\_sram\_wdata  (cpu\_inst\_wdata),      .inst\_sram\_rdata  (cpu\_inst\_rdata),      .data\_sram\_en     (cpu\_data\_en   ),//1      .data\_sram\_wen    (cpu\_data\_wen  ),//sel      .data\_sram\_addr   (cpu\_data\_addr ),      .data\_sram\_wdata  (cpu\_data\_wdata),      .data\_sram\_rdata  (cpu\_data\_rdata),      .debug\_wb\_pc(),      .debug\_wb\_rf\_wen(),      .debug\_wb\_rf\_wnum(),      .debug\_wb\_rf\_wdata()  );  // cpu 本身没有问题    reg [31:0] cpu\_inst\_rdata\_r;  reg [31:0] cpu\_data\_rdata\_r;  reg [31:0] base\_ram\_data\_r;  reg [19:0] base\_ram\_addr\_r;  reg [3:0] base\_ram\_be\_n\_r;  reg base\_ram\_ce\_n\_r;  reg base\_ram\_oe\_n\_r;  reg base\_ram\_we\_n\_r;  reg [31:0] ext\_ram\_data\_r;  reg [19:0] ext\_ram\_addr\_r;  reg [3:0] ext\_ram\_be\_n\_r;  reg ext\_ram\_ce\_n\_r;  reg ext\_ram\_oe\_n\_r;  reg ext\_ram\_we\_n\_r;  reg sel\_inst; // 1-inst 0-data for base\_ram  reg sel\_uart;  reg sel\_uart\_flag; // 1-flag 0-data  wire [31:0] uart\_rdata;  reg [31:0] uart\_wdata;  wire [7:0] ext\_uart\_rx;  reg  [7:0] ext\_uart\_buffer, ext\_uart\_tx;  wire ext\_uart\_ready, ext\_uart\_clear, ext\_uart\_busy;  reg ext\_uart\_start, ext\_uart\_avai;  reg cpu\_data\_avai;  reg uart\_read\_flag;  reg uart\_write\_flag;  assign base\_ram\_data = ~base\_ram\_we\_n\_r ? base\_ram\_data\_r : 32'bz;  assign ext\_ram\_data = ~ext\_ram\_we\_n\_r ? ext\_ram\_data\_r : 32'bz;  assign base\_ram\_addr = base\_ram\_addr\_r;  assign base\_ram\_be\_n = base\_ram\_be\_n\_r;  assign base\_ram\_ce\_n = base\_ram\_ce\_n\_r;  assign base\_ram\_oe\_n = base\_ram\_oe\_n\_r;  assign base\_ram\_we\_n = base\_ram\_we\_n\_r;  assign ext\_ram\_addr = ext\_ram\_addr\_r;  assign ext\_ram\_be\_n = ext\_ram\_be\_n\_r;  assign ext\_ram\_ce\_n = ext\_ram\_ce\_n\_r;  assign ext\_ram\_oe\_n = ext\_ram\_oe\_n\_r;  assign ext\_ram\_we\_n = ext\_ram\_we\_n\_r;  // in  always @ (\*) begin      if (reset\_of\_clk10M) begin          cpu\_inst\_rdata\_r <= 32'b0;          cpu\_data\_rdata\_r <= 32'b0;      end      else begin          cpu\_inst\_rdata\_r <= ~sel\_inst ? 32'b0                              : ~base\_ram\_oe\_n\_r ? base\_ram\_data                              : 32'b0;          cpu\_data\_rdata\_r <= sel\_uart ? uart\_rdata : sel\_inst ? (~ext\_ram\_oe\_n\_r ? ext\_ram\_data : 32'b0) : (~base\_ram\_oe\_n\_r ? base\_ram\_data : 32'b0);      end  end  assign cpu\_inst\_rdata = cpu\_inst\_rdata\_r;  assign cpu\_data\_rdata = cpu\_data\_rdata\_r;  assign uart\_rdata = sel\_uart\_flag ? {30'b0,ext\_uart\_avai,~ext\_uart\_busy} : {24'b0,ext\_uart\_buffer};  reg [3:0] state;    // out  always @ (posedge clk\_20M) begin      if (reset\_of\_clk10M) begin          base\_ram\_addr\_r <= 19'b0;          base\_ram\_be\_n\_r <= 4'b0;          base\_ram\_ce\_n\_r <= 1'b1;          base\_ram\_oe\_n\_r <= 1'b1;          base\_ram\_we\_n\_r <= 1'b1;          base\_ram\_data\_r <= 32'b0;          ext\_ram\_addr\_r <= 19'b0;          ext\_ram\_be\_n\_r <= 4'b0;          ext\_ram\_ce\_n\_r <= 1'b1;          ext\_ram\_oe\_n\_r <= 1'b1;          ext\_ram\_we\_n\_r <= 1'b1;          ext\_ram\_data\_r <= 32'b0;          sel\_inst <= 1'b0;          sel\_uart <= 1'b0;          sel\_uart\_flag <= 1'b0;          uart\_wdata <= 32'b0;          cpu\_data\_avai <= 1'b0;          state <= 4'b0;      end      else if (cpu\_data\_addr >=32'h0 && cpu\_data\_addr <= 32'h003fffff && cpu\_data\_en) begin          base\_ram\_addr\_r <= cpu\_data\_addr[21:2];          base\_ram\_be\_n\_r <= (|cpu\_data\_wen) ? ~cpu\_data\_wen : 4'b0;          base\_ram\_ce\_n\_r <= ~cpu\_data\_en;          base\_ram\_oe\_n\_r <= ~(cpu\_data\_en & ~(|cpu\_data\_wen));          base\_ram\_we\_n\_r <= ~(cpu\_data\_en & (|cpu\_data\_wen));          base\_ram\_data\_r <= cpu\_data\_wdata;          ext\_ram\_addr\_r <= 19'b0;          ext\_ram\_be\_n\_r <= 4'b0;          ext\_ram\_ce\_n\_r <= 1'b1;          ext\_ram\_oe\_n\_r <= 1'b1;          ext\_ram\_we\_n\_r <= 1'b1;          ext\_ram\_data\_r <= 32'b0;          sel\_inst <= 1'b0;          sel\_uart <= 1'b0;          sel\_uart\_flag <= 1'b0;          uart\_wdata <= 32'b0;          cpu\_data\_avai <= 1'b0;          state <= 4'b1;      end      else if (cpu\_data\_addr >= 32'h00400000 && cpu\_data\_addr <= 32'h007fffff && cpu\_data\_en) begin          base\_ram\_addr\_r <= cpu\_inst\_addr[21:2];          base\_ram\_be\_n\_r <= 4'b0;          base\_ram\_ce\_n\_r <= ~cpu\_inst\_en;          base\_ram\_oe\_n\_r <= ~cpu\_inst\_en ;          base\_ram\_we\_n\_r <= 1'b1;          base\_ram\_data\_r <= cpu\_inst\_wdata;            ext\_ram\_addr\_r <= cpu\_data\_addr[21:2];          ext\_ram\_be\_n\_r <= (|cpu\_data\_wen) ? ~cpu\_data\_wen : 4'b0;          ext\_ram\_ce\_n\_r <= ~cpu\_data\_en;          ext\_ram\_oe\_n\_r <= ~(cpu\_data\_en & ~(|cpu\_data\_wen));          ext\_ram\_we\_n\_r <= ~(cpu\_data\_en & (|cpu\_data\_wen));          ext\_ram\_data\_r <= cpu\_data\_wdata;          sel\_inst <= 1'b1;          sel\_uart <= 1'b0;          sel\_uart\_flag <= 1'b0;          uart\_wdata <= 32'b0;          cpu\_data\_avai <= 1'b0;          state <= 4'd2;      end      else if (cpu\_data\_addr == 32'h1fd003fc) begin          base\_ram\_addr\_r <= cpu\_inst\_addr[21:2];          base\_ram\_be\_n\_r <= 4'b0;          base\_ram\_ce\_n\_r <= ~cpu\_inst\_en;          base\_ram\_oe\_n\_r <= ~cpu\_inst\_en ;          base\_ram\_we\_n\_r <= 1'b1;          base\_ram\_data\_r <= cpu\_inst\_wdata;            ext\_ram\_addr\_r <= 19'b0;          ext\_ram\_be\_n\_r <= 4'b0;          ext\_ram\_ce\_n\_r <= 1'b1;          ext\_ram\_oe\_n\_r <= 1'b1;          ext\_ram\_we\_n\_r <= 1'b1;          ext\_ram\_data\_r <= 32'b0;          sel\_inst <= 1'b1;          sel\_uart <= 1'b1;          sel\_uart\_flag <= 1'b1;          uart\_wdata <= 32'b0;          cpu\_data\_avai <= 1'b0;          state <= 4'd3;      end      else if (cpu\_data\_addr == 32'h1fd003f8 && cpu\_data\_en) begin          base\_ram\_addr\_r <= cpu\_inst\_addr[21:2];          base\_ram\_be\_n\_r <= 4'b0;          base\_ram\_ce\_n\_r <= ~cpu\_inst\_en;          base\_ram\_oe\_n\_r <= ~cpu\_inst\_en ;          base\_ram\_we\_n\_r <= 1'b1;          base\_ram\_data\_r <= cpu\_inst\_wdata;            ext\_ram\_addr\_r <= 19'b0;          ext\_ram\_be\_n\_r <= 4'b0;          ext\_ram\_ce\_n\_r <= 1'b1;          ext\_ram\_oe\_n\_r <= 1'b1;          ext\_ram\_we\_n\_r <= 1'b1;          ext\_ram\_data\_r <= 32'b0;          sel\_inst <= 1'b1;          sel\_uart <= 1'b1;          sel\_uart\_flag <= 1'b0;          uart\_wdata <= cpu\_data\_wdata;          cpu\_data\_avai <= (|cpu\_data\_wen) ? 1'b1 : 1'b0;          state <= 4'd4;      end      else begin          base\_ram\_addr\_r <= cpu\_inst\_addr[21:2];          base\_ram\_be\_n\_r <= 4'b0;          base\_ram\_ce\_n\_r <= ~cpu\_inst\_en;          base\_ram\_oe\_n\_r <= ~cpu\_inst\_en ;          base\_ram\_we\_n\_r <= 1'b1;          base\_ram\_data\_r <= cpu\_inst\_wdata;            ext\_ram\_addr\_r <= 19'b0;          ext\_ram\_be\_n\_r <= 4'b0;          ext\_ram\_ce\_n\_r <= 1'b1;          ext\_ram\_oe\_n\_r <= 1'b1;          ext\_ram\_we\_n\_r <= 1'b1;          ext\_ram\_data\_r <= 32'b0;          sel\_inst <= 1'b1;          sel\_uart <= 1'b0;          sel\_uart\_flag <= 1'b0;          uart\_wdata <= 32'b0;          cpu\_data\_avai <= 1'b0;          state <= 4'd5;      end  end  // uart  async\_receiver #(.ClkFrequency(59000000),.Baud(9600)) //接收模块，9600无检验位      ext\_uart\_r(          .clk(clk\_20M),                       //外部时钟信号          .RxD(rxd),                           //外部串行信号输入          .RxD\_data\_ready(ext\_uart\_ready),  //数据接收到标志          .RxD\_clear(ext\_uart\_clear),       //清除接收标志          .RxD\_data(ext\_uart\_rx)             //接收到的一字节数据      );  assign ext\_uart\_clear = ext\_uart\_ready; //收到数据的同时，清除标志，因为数据已取到ext\_uart\_buffer中  always @(posedge clk\_20M) begin //接收到缓冲区ext\_uart\_buffer      if (reset\_of\_clk10M) begin          ext\_uart\_buffer <= 8'b0;          ext\_uart\_avai <= 1'b0;      end      else if(ext\_uart\_ready)begin          ext\_uart\_buffer <= ext\_uart\_rx;          ext\_uart\_avai <= 1'b1;      end      else if(cpu\_data\_addr == 32'h1fd003f8 && (cpu\_data\_en & ~(|cpu\_data\_wen)) && ext\_uart\_avai)begin          ext\_uart\_avai <= 1'b0;      end  end  always @(posedge clk\_20M) begin //将缓冲区ext\_uart\_buffer发送出去      if(!ext\_uart\_busy && cpu\_data\_avai)begin          ext\_uart\_tx <= uart\_wdata[7:0];          ext\_uart\_start <= 1;      end else begin          ext\_uart\_start <= 0;      end  end  async\_transmitter #(.ClkFrequency(59000000),.Baud(9600)) //发送模块，9600无检验位      ext\_uart\_t(          .clk(clk\_20M),                  //外部时钟信号          .TxD(txd),                      //串行信号输出          .TxD\_busy(ext\_uart\_busy),       //发送器忙状态指示          .TxD\_start(ext\_uart\_start),    //开始发送信号          .TxD\_data(ext\_uart\_tx)        //待发送的数据      );  // | 地址 | 位 | 说明 |  // | --- | --- |--- |  // | 0xBFD003F8| [7:0] | 串口数据，读、写地址分别表示串口接收、发送一个字节|  // | 0xBFD003FC| [0] | 只读，为1时表示串口空闲，可发送数据|  // | 0xBFD003FC| [1] | 只读，为1时表示串口收到数据|  // // assign inst\_sram\_rdata =  base\_ram\_data;  // assign base\_ram\_addr = inst\_sram\_en ? inst\_sram\_addr[21:2] : cpu\_inst\_addr[21:2];  // assign base\_ram\_be\_n = {4{~cpu\_inst\_en}};  // assign base\_ram\_ce\_n = ~cpu\_inst\_en;  // assign base\_ram\_oe\_n = ~cpu\_inst\_en;  // assign base\_ram\_we\_n = ~cpu\_inst\_wen;  // // assign ext\_ram\_data = ext\_ram\_data\_r;  // // assign data\_sram\_rdata = ext\_ram\_data;  // assign ext\_ram\_addr = data\_sram\_addr[21:2];  // assign ext\_ram\_be\_n = 4'b0; //~data\_sram\_wen & {4{ext\_ram\_oe\_n}};  // assign ext\_ram\_ce\_n = ~data\_sram\_en;  // assign ext\_ram\_oe\_n = ~(data\_sram\_en & ~data\_sram\_wen[0]);  // assign ext\_ram\_we\_n = ~(data\_sram\_en & data\_sram\_wen[0]);  // 不使用内存、串口时，禁用其使能信号  // assign base\_ram\_ce\_n = 1'b1;  // assign base\_ram\_oe\_n = 1'b1;  // assign base\_ram\_we\_n = 1'b1;  // assign ext\_ram\_ce\_n = 1'b1;  // assign ext\_ram\_oe\_n = 1'b1;  // assign ext\_ram\_we\_n = 1'b1;  // 数码管连接关系示意图，dpy1同理  // p=dpy0[0] // ---a---  // c=dpy0[1] // |     |  // d=dpy0[2] // f     b  // e=dpy0[3] // |     |  // b=dpy0[4] // ---g---  // a=dpy0[5] // |     |  // f=dpy0[6] // e     c  // g=dpy0[7] // |     |  //           // ---d---  p  // 7段数码管译码器演示，将number用16进制显示在数码管上面  // wire[7:0] number;  // SEG7\_LUT segL(.oSEG1(dpy0), .iDIG(number[3:0])); //dpy0是低位数码管  // SEG7\_LUT segH(.oSEG1(dpy1), .iDIG(number[7:4])); //dpy1是高位数码管  // reg [7:0] wdata\_r;  // reg[15:0] led\_bits;  // assign leds = {8'b0,wdata\_r};  // always @ (posedge clk\_10M) begin  //     if (conf\_en & conf\_wen[0])begin  //         wdata\_r <= conf\_wdata[7:0];  //     end  // end  // assign number = wdata\_r;  // always@(posedge clock\_btn or posedge reset\_btn) begin  //     if(reset\_btn)begin //复位按下，设置LED为初始值  //         led\_bits <= 16'h1;  //     end  //     else begin //每次按下时钟按钮，LED循环左移  //         led\_bits <= {led\_bits[14:0],led\_bits[15]};  //     end  // end  //直连串口接收发送演示，从直连串口收到的数据再发送出去  // wire [7:0] ext\_uart\_rx;  // reg  [7:0] ext\_uart\_buffer, ext\_uart\_tx;  // wire ext\_uart\_ready, ext\_uart\_clear, ext\_uart\_busy;  // reg ext\_uart\_start, ext\_uart\_avai;    // assign number = ext\_uart\_buffer;  // async\_receiver #(.ClkFrequency(50000000),.Baud(9600)) //接收模块，9600无检验位  //     ext\_uart\_r(  //         .clk(clk\_50M),                       //外部时钟信号  //         .RxD(rxd),                           //外部串行信号输入  //         .RxD\_data\_ready(ext\_uart\_ready),  //数据接收到标志  //         .RxD\_clear(ext\_uart\_clear),       //清除接收标志  //         .RxD\_data(ext\_uart\_rx)             //接收到的一字节数据  //     );  // assign ext\_uart\_clear = ext\_uart\_ready; //收到数据的同时，清除标志，因为数据已取到ext\_uart\_buffer中  // always @(posedge clk\_50M) begin //接收到缓冲区ext\_uart\_buffer  //     if(ext\_uart\_ready)begin  //         ext\_uart\_buffer <= ext\_uart\_rx;  //         ext\_uart\_avai <= 1;  //     end else if(!ext\_uart\_busy && ext\_uart\_avai)begin  //         ext\_uart\_avai <= 0;  //     end  // end  // always @(posedge clk\_50M) begin //将缓冲区ext\_uart\_buffer发送出去  //     if(!ext\_uart\_busy && ext\_uart\_avai)begin  //         ext\_uart\_tx <= ext\_uart\_buffer;  //         ext\_uart\_start <= 1;  //     end else begin  //         ext\_uart\_start <= 0;  //     end  // end  // async\_transmitter #(.ClkFrequency(50000000),.Baud(9600)) //发送模块，9600无检验位  //     ext\_uart\_t(  //         .clk(clk\_50M),                  //外部时钟信号  //         .TxD(txd),                      //串行信号输出  //         .TxD\_busy(ext\_uart\_busy),       //发送器忙状态指示  //         .TxD\_start(ext\_uart\_start),    //开始发送信号  //         .TxD\_data(ext\_uart\_tx)        //待发送的数据  //     );  // //图像输出演示，分辨率800x600@75Hz，像素时钟为50MHz  // wire [11:0] hdata;  // assign video\_red = hdata < 266 ? 3'b111 : 0; //红色竖条  // assign video\_green = hdata < 532 && hdata >= 266 ? 3'b111 : 0; //绿色竖条  // assign video\_blue = hdata >= 532 ? 2'b11 : 0; //蓝色竖条  // assign video\_clk = clk\_50M;  // vga #(12, 800, 856, 976, 1040, 600, 637, 643, 666, 1, 1) vga800x600at75 (  //     .clk(clk\_50M),  //     .hdata(hdata), //横坐标  //     .vdata(),      //纵坐标  //     .hsync(video\_hsync),  //     .vsync(video\_vsync),  //     .data\_enable(video\_de)  // );  // /\* =========== Demo code end =========== \*/  endmodule |  Pc  |  | | --- | | `include "defines.v"  module pc(    input wire clk,    input wire rst,    input wire [`StallBus] stall,    input wire flush,    input wire [`RegBus] new\_pc,    input wire branch\_flag\_i,    input wire [`RegBus] branch\_target\_address\_i,    input wire l2\_pc\_plus\_4\_req,    output reg [`InstAddrBus] pc,    output reg ce,    output wire [31:0] excepttype\_o  );    reg excepttype\_is\_ft\_adel;    assign excepttype\_o = {15'b0,excepttype\_is\_ft\_adel,16'b0};      always @ (posedge clk) begin      if (rst == `RstEnable) begin        ce <= `ChipDisable;      end      else begin        ce <= `ChipEnable;      end    end    always @ (posedge clk) begin      if (rst == `RstEnable) begin        pc <= 32'h80000000;      end      else if (ce == `ChipEnable) begin        if (flush == `True\_v) begin          pc <= new\_pc + 32'd4;        end        else if (stall [0] == `NoStop) begin          if (branch\_flag\_i == `Branch) begin            pc <= branch\_target\_address\_i + 32'd4;          end          else if (l2\_pc\_plus\_4\_req) begin            pc <= pc + 32'h8;          end          else begin            pc <= pc + 32'h4;          end        end      end    end      always @ (\*) begin      excepttype\_is\_ft\_adel <= (pc[1:0] != 2'b00) ? `True\_v : `False\_v;    end  endmodule |  pre\_icache  |  | | --- | | `include "defines.v"  module pre\_icache(    input wire clk,    input wire rst,    input wire [`StallBus] stall,    input wire flush,    input wire [`RegBus] new\_pc,    input wire branch\_flag\_i,    input wire [`RegBus] branch\_target\_address\_i,    input wire [`InstAddrBus] pc\_pc,    output reg [`InstAddrBus] icache\_pc,    input wire [31:0] pc\_excepttype,    output reg [31:0] icache\_excepttype  );    always @ (posedge clk) begin      if (rst == `RstEnable) begin        icache\_pc <= `ZeroWord;        icache\_excepttype <= `ZeroWord;      end      else if (flush == `True\_v) begin        icache\_pc <= new\_pc;        if (new\_pc[1:0] != 2'b00) begin          icache\_excepttype <= {15'b0,`True\_v,16'b0};        end        else begin          icache\_excepttype <= `ZeroWord;        end      end      else if (stall[1] == `Stop && stall[2] == `NoStop) begin        icache\_pc <= `ZeroWord;        icache\_excepttype <= `ZeroWord;      end      else if (stall[1] == `NoStop) begin        if (branch\_flag\_i == `Branch) begin          icache\_pc <= branch\_target\_address\_i;          if (branch\_target\_address\_i[1:0] != 2'b00) begin            icache\_excepttype <= {15'b0,`True\_v,16'b0};          end          else begin            icache\_excepttype <= pc\_excepttype;          end        end        else begin          icache\_pc <= pc\_pc;          icache\_excepttype <= pc\_excepttype;        end      end    end  endmodule |  line2\_pre\_decoder  |  | | --- | | `include "defines.v"  module line2\_pre\_decoder(    input wire rst,    // output wire stallreq,    input wire [`InstAddrBus] pc\_i,    input wire [`InstBus] inst\_i,    output wire [`InstAddrBus] pc\_o,    // output wire [`InstBus] inst\_o,    output reg reg1\_read\_o, // reg1 read enable    output reg reg2\_read\_o, // reg2 read enable    output reg [`RegAddrBus] reg1\_addr\_o, // reg1 read addr    output reg [`RegAddrBus] reg2\_addr\_o, // reg2 read addr    // input wire [`RegBus] reg1\_data\_i, // reg1 data from rf    // input wire [`RegBus] reg2\_data\_i, // reg2 data from rf    output reg [`AluOpBus] aluop\_o, // 需要运行的指令代码    output reg [`AluSelBus] alusel\_o, // 需要运行的指令的类型    // output reg [`RegBus] reg1\_o, // 指令需要的源操作数1    // output reg [`RegBus] reg2\_o, // 指令需要的源操作数2    output reg [`RegAddrBus] waddr\_o, //指令需要写入的目的寄存器地址    output reg wreg\_o, // 指令是否需要写入寄存器  // jump and branch    // input wire is\_in\_delayslot\_i,    // output reg next\_inst\_in\_delayslot\_o,    // output reg branch\_flag\_o,    // output reg [`RegBus] branch\_target\_address\_o,    // output reg [`RegBus] link\_addr\_o,    // output reg is\_in\_delayslot\_o,  // load 相关    // input wire [`RegAddrBus] ex\_waddr\_i,    // input wire [`AluOpBus] ex\_aluop\_i,    // input wire [`RegBus] ex\_mem\_addr\_i,    // // input wire [`RegAddrBus] premem\_waddr\_i,    // // input wire [`AluOpBus] premem\_aluop\_i,    // input wire [`RegAddrBus] dcache\_waddr\_i,    // input wire [`AluOpBus] dcache\_aluop\_i,    // input wire [`RegBus] dcache\_wdata\_i,  // excepttype    // input wire [31:0] excepttype\_i,    // output wire [31:0] excepttype\_o    output reg [`RegBus] imm\_o, // imm\_o - 32bit  imm - 16bit 记得补充完整    output reg l1\_run,    output reg l2\_run  );    wire [5:0] opcode;    wire [4:0] rs;    wire [4:0] rt;    wire [4:0] rd;    wire [4:0] sa;    wire [5:0] func;    wire [15:0] imm;    wire [25:0] instr\_index;    wire [19:0] code;    wire [4:0] base;    wire [15:0] offset;    wire [2:0] sel;    assign opcode = inst\_i[31:26];    assign rs = inst\_i[25:21];    assign rt = inst\_i[20:16];    assign rd = inst\_i[15:11];    assign sa = inst\_i[10: 6];    assign func = inst\_i[5:0];    assign imm = inst\_i[15:0];    assign instr\_index = inst\_i[25:0];    assign code = inst\_i[25:6];    assign base = inst\_i[25:21];    assign offset = inst\_i[15:0];    assign sel = inst\_i[2:0];      reg instvalid;  // jump and branch    // wire [`RegBus] pc\_plus\_8;    // wire [`RegBus] pc\_plus\_4;    // wire [`RegBus] imm\_sll2\_signedext;    // assign pc\_plus\_8 = pc\_i + 8;    // assign pc\_plus\_4 = pc\_i + 4;    // assign imm\_sll2\_signedext = {{14{inst\_i[15]}},inst\_i[15:0],2'b00};  // pc & inst    assign pc\_o = pc\_i;    // assign inst\_o = inst\_i;  // excepttype    // reg excepttype\_is\_syscall;    // reg excepttype\_is\_eret;    // reg excepttype\_is\_break;    // assign excepttype\_o = {15'b0,excepttype\_i[16],2'b0,excepttype\_is\_break,excepttype\_is\_eret,2'b00,instvalid,excepttype\_is\_syscall,8'b0};    // decoder    always @ (\*) begin      if (rst == `RstEnable) begin        reg1\_read\_o <= `False\_v;        reg2\_read\_o <= `False\_v;        reg1\_addr\_o <= `NOPRegAddr;        reg2\_addr\_o <= `NOPRegAddr;        aluop\_o <= `NOP;        alusel\_o <= `EXE\_NOP;        waddr\_o <= `NOPRegAddr;        wreg\_o <= `WriteDisable;        imm\_o <= `ZeroWord;        // excepttype\_is\_break <= `False\_v;        // excepttype\_is\_syscall <= `False\_v;        // excepttype\_is\_eret <= `False\_v;        instvalid <= `InstValid; // 可能是为了避免异常触发，所以这里用的是valid        // stallreq <= `NoStop;        // link\_addr\_o <= `ZeroWord;        // branch\_target\_address\_o <= `ZeroWord;        // branch\_flag\_o <= `NotBranch;        // next\_inst\_in\_delayslot\_o <= `NotInDelaySlot;      end      else begin        //统一赋值区域        reg1\_read\_o <= `False\_v;        reg2\_read\_o <= `False\_v;        reg1\_addr\_o <= rs;        reg2\_addr\_o <= rt;        aluop\_o <= `NOP;        alusel\_o <= `EXE\_NOP;        waddr\_o <= rd;        wreg\_o <= `WriteDisable;        imm\_o <= `ZeroWord;        // excepttype\_is\_break <= `False\_v;        // excepttype\_is\_syscall <= `False\_v;        // excepttype\_is\_eret <= `False\_v;        instvalid <= `InstInvalid; // 默认可能找不到这个指令，于是这个指令invalid        // stallreq <= `NoStop;        // link\_addr\_o <= `ZeroWord;        // branch\_target\_address\_o <= `ZeroWord;        // branch\_flag\_o <= `NotBranch;        // next\_inst\_in\_delayslot\_o <= `NotInDelaySlot;        //统一赋值区域        case (opcode)          6'b000000:begin            case (func)              6'b100000:begin                instvalid <= `InstValid;                alusel\_o <= `EXE\_ARITHMETIC;                aluop\_o <= `ADD;                reg1\_read\_o <= `True\_v;                reg2\_read\_o <= `True\_v;                reg1\_addr\_o <= rs;                reg2\_addr\_o <= rt;                wreg\_o <= `WriteEnable;                waddr\_o <= rd;              end              6'b100001:begin                instvalid <= `InstValid;                alusel\_o <= `EXE\_ARITHMETIC;                aluop\_o <= `ADDU;                reg1\_read\_o <= `True\_v;                reg2\_read\_o <= `True\_v;                reg1\_addr\_o <= rs;                reg2\_addr\_o <= rt;                wreg\_o <= `WriteEnable;                waddr\_o <= rd;              end              6'b100010:begin                instvalid <= `InstValid;                alusel\_o <= `EXE\_ARITHMETIC;                aluop\_o <= `SUB;                reg1\_read\_o <= `True\_v;                reg2\_read\_o <= `True\_v;                reg1\_addr\_o <= rs;                reg2\_addr\_o <= rt;                wreg\_o <= `WriteEnable;                waddr\_o <= rd;              end              6'b100011:begin                instvalid <= `InstValid;                alusel\_o <= `EXE\_ARITHMETIC;                aluop\_o <= `SUBU;                  reg1\_read\_o <= `True\_v;                reg2\_read\_o <= `True\_v;                reg1\_addr\_o <= rs;                reg2\_addr\_o <= rt;                wreg\_o <= `WriteEnable;                waddr\_o <= rd;              end              6'b101010:begin                instvalid <= `InstValid;                alusel\_o <= `EXE\_ARITHMETIC;                aluop\_o <= `SLT;                reg1\_read\_o <= `True\_v;                reg2\_read\_o <= `True\_v;                reg1\_addr\_o <= rs;                reg2\_addr\_o <= rt;                wreg\_o <= `WriteEnable;                waddr\_o <= rd;              end              6'b101011:begin                instvalid <= `InstValid;                alusel\_o <= `EXE\_ARITHMETIC;                aluop\_o <= `SLTU;                reg1\_read\_o <= `True\_v;                reg2\_read\_o <= `True\_v;                reg1\_addr\_o <= rs;                reg2\_addr\_o <= rt;                wreg\_o <= `WriteEnable;                waddr\_o <= rd;              end              6'b011010:begin                instvalid <= `InstValid;                alusel\_o <= `EXE\_NOP;                aluop\_o <= `DIV;                reg1\_read\_o <= `True\_v;                reg2\_read\_o <= `True\_v;                reg1\_addr\_o <= rs;                reg2\_addr\_o <= rt;                wreg\_o <= `WriteDisable;              end              6'b011011:begin                instvalid <= `InstValid;                alusel\_o <= `EXE\_NOP;                aluop\_o <= `DIVU;                reg1\_read\_o <= `True\_v;                reg2\_read\_o <= `True\_v;                reg1\_addr\_o <= rs;                reg2\_addr\_o <= rt;                wreg\_o <= `WriteDisable;              end              6'b011000:begin                instvalid <= `InstValid;                alusel\_o <= `EXE\_ARITHMETIC;                aluop\_o <= `MULT;                reg1\_read\_o <= `True\_v;                reg2\_read\_o <= `True\_v;                reg1\_addr\_o <= rs;                reg2\_addr\_o <= rt;                wreg\_o <= `WriteDisable;              end              6'b011001:begin                instvalid <= `InstValid;                alusel\_o <= `EXE\_ARITHMETIC;                aluop\_o <= `MULTU;                reg1\_read\_o <= `True\_v;                reg2\_read\_o <= `True\_v;                reg1\_addr\_o <= rs;                reg2\_addr\_o <= rt;                wreg\_o <= `WriteDisable;              end              6'b100100:begin                instvalid <= `InstValid;                alusel\_o <= `EXE\_LOGIC;                aluop\_o <= `AND;                reg1\_read\_o <= `True\_v;                reg2\_read\_o <= `True\_v;                reg1\_addr\_o <= rs;                reg2\_addr\_o <= rt;                wreg\_o <= `WriteEnable;                waddr\_o <= rd;              end              6'b100111:begin                instvalid <= `InstValid;                alusel\_o <= `EXE\_LOGIC;                aluop\_o <= `NOR;                reg1\_read\_o <= `True\_v;                reg2\_read\_o <= `True\_v;                reg1\_addr\_o <= rs;                reg2\_addr\_o <= rt;                wreg\_o <= `WriteEnable;                waddr\_o <= rd;              end              6'b100101:begin                instvalid <= `InstValid;                alusel\_o <= `EXE\_LOGIC;                aluop\_o <= `OR;                reg1\_read\_o <= `True\_v;                reg2\_read\_o <= `True\_v;                reg1\_addr\_o <= rs;                reg2\_addr\_o <= rt;                wreg\_o <= `WriteEnable;                waddr\_o <= rd;              end              6'b100110:begin                instvalid <= `InstValid;                alusel\_o <= `EXE\_LOGIC;                aluop\_o <= `XOR;                reg1\_read\_o <= `True\_v;                reg2\_read\_o <= `True\_v;                reg1\_addr\_o <= rs;                reg2\_addr\_o <= rt;                wreg\_o <= `WriteEnable;                waddr\_o <= rd;              end              6'b000100:begin                instvalid <= `InstValid;                alusel\_o <= `EXE\_SHIFT;                aluop\_o <= `SLLV;                reg1\_read\_o <= `True\_v;                reg2\_read\_o <= `True\_v;                reg1\_addr\_o <= rs;                reg2\_addr\_o <= rt;                wreg\_o <= `WriteEnable;                waddr\_o <= rd;              end              6'b000000:begin                instvalid <= `InstValid;                alusel\_o <= `EXE\_SHIFT;                aluop\_o <= `SLL;                reg1\_read\_o <= `False\_v;                reg2\_read\_o <= `True\_v;                imm\_o <= {27'b0,sa};                reg2\_addr\_o <= rt;                wreg\_o <= `WriteEnable;                waddr\_o <= rd;              end              6'b000111:begin                instvalid <= `InstValid;                alusel\_o <= `EXE\_SHIFT;                aluop\_o <= `SRAV;                reg1\_read\_o <= `True\_v;                reg2\_read\_o <= `True\_v;                reg1\_addr\_o <= rs;                reg2\_addr\_o <= rt;                wreg\_o <= `WriteEnable;                waddr\_o <= rd;              end              6'b000011:begin                instvalid <= `InstValid;                alusel\_o <= `EXE\_SHIFT;                aluop\_o <= `SRA;                reg1\_read\_o <= `False\_v;                reg2\_read\_o <= `True\_v;                imm\_o <= {27'b0,sa};                reg2\_addr\_o <= rt;                wreg\_o <= `WriteEnable;                waddr\_o <= rd;              end              6'b000110:begin                instvalid <= `InstValid;                alusel\_o <= `EXE\_SHIFT;                aluop\_o <= `SRLV;                reg1\_read\_o <= `True\_v;                reg2\_read\_o <= `True\_v;                reg1\_addr\_o <= rs;                reg2\_addr\_o <= rt;                wreg\_o <= `WriteEnable;                waddr\_o <= rd;              end              6'b000010:begin                instvalid <= `InstValid;                alusel\_o <= `EXE\_SHIFT;                aluop\_o <= `SRL;                reg1\_read\_o <= `False\_v;                reg2\_read\_o <= `True\_v;                imm\_o <= {27'b0,sa};                reg2\_addr\_o <= rt;                wreg\_o <= `WriteEnable;                waddr\_o <= rd;              end              6'b010000:begin                instvalid <= `InstValid;                alusel\_o <= `EXE\_MOVE;                aluop\_o <= `MFHI;                // reg1\_read\_o <= `False\_v;                // reg2\_read\_o <= `False\_v;                wreg\_o <= `WriteEnable;                waddr\_o <= rd;              end              6'b010010:begin                instvalid <= `InstValid;                alusel\_o <= `EXE\_MOVE;                aluop\_o <= `MFLO;                // reg1\_read\_o <= `False\_v;                // reg2\_read\_o <= `False\_v;                wreg\_o <= `WriteEnable;                waddr\_o <= rd;              end              6'b010001:begin                instvalid <= `InstValid;                alusel\_o <= `EXE\_MOVE;                aluop\_o <= `MTHI;                // reg1\_read\_o <= `True\_v;                // reg2\_read\_o <= `False\_v;                // reg1\_addr\_o <= rs;                wreg\_o <= `WriteDisable;              end              6'b010011:begin                instvalid <= `InstValid;                alusel\_o <= `EXE\_MOVE;                aluop\_o <= `MTLO;                // reg1\_read\_o <= `True\_v;                // reg2\_read\_o <= `False\_v;                // reg1\_addr\_o <= rs;                wreg\_o <= `WriteDisable;              end              6'b001000:begin                instvalid <= `InstValid;                alusel\_o <= `EXE\_JUMP\_BRANCH;                aluop\_o <= `JR;                // reg1\_read\_o <= `True\_v;                // reg2\_read\_o <= `False\_v;                // reg1\_addr\_o <= rs;                wreg\_o <= `WriteDisable;                // branch\_target\_address\_o <= reg1\_data\_i;                // branch\_flag\_o <= `Branch;                // next\_inst\_in\_delayslot\_o <= `InDelaySlot;              end              6'b001001:begin                instvalid <= `InstValid;                alusel\_o <= `EXE\_JUMP\_BRANCH;                aluop\_o <= `JALR;                // reg1\_read\_o <= `True\_v;                // reg2\_read\_o <= `False\_v;                // reg1\_addr\_o <= rs;                wreg\_o <= `WriteEnable;                waddr\_o <= rd;                // link\_addr\_o <= pc\_plus\_8;                // branch\_target\_address\_o <= reg1\_data\_i;                // branch\_flag\_o <= `Branch;                // next\_inst\_in\_delayslot\_o <= `InDelaySlot;              end              6'b001100:begin                instvalid <= `InstValid;                alusel\_o <= `EXE\_NOP;                aluop\_o <= `SYSCALL;                // reg1\_read\_o <= `False\_v;                // reg2\_read\_o <= `False\_v;                wreg\_o <= `WriteDisable;                // excepttype\_is\_syscall <= `True\_v;              end              6'b001101:begin                instvalid <= `InstValid;                alusel\_o <= `EXE\_NOP;                aluop\_o <= `BREAK ;                // reg1\_read\_o <= `False\_v;                // reg2\_read\_o <= `False\_v;                wreg\_o <= `WriteDisable;                // excepttype\_is\_break <= `True\_v;              end              default:begin                end            endcase // func          end          6'b001000:begin            instvalid <= `InstValid;            alusel\_o <= `EXE\_ARITHMETIC;            aluop\_o <= `ADDI;            reg1\_read\_o <= `True\_v;            reg2\_read\_o <= `False\_v;            reg1\_addr\_o <= rs;            imm\_o <= {{16{imm[15]}},imm};              wreg\_o <= `WriteEnable;            waddr\_o <= rt;          end          6'b001001:begin            instvalid <= `InstValid;            alusel\_o <= `EXE\_ARITHMETIC;            aluop\_o <= `ADDIU;            reg1\_read\_o <= `True\_v;            reg2\_read\_o <= `False\_v;            reg1\_addr\_o <= rs;            imm\_o <= {{16{imm[15]}},imm};            wreg\_o <= `WriteEnable;            waddr\_o <= rt;          end          6'b011100:begin            instvalid <= `InstValid;            alusel\_o <= `EXE\_MUL;            aluop\_o <= `MUL;            reg1\_read\_o <= `True\_v;            reg2\_read\_o <= `True\_v;            reg1\_addr\_o <= rs;            reg2\_addr\_o <= rt;            wreg\_o <= `WriteEnable;            waddr\_o <= rd;          end          6'b001010:begin            instvalid <= `InstValid;            alusel\_o <= `EXE\_ARITHMETIC;            aluop\_o <= `SLTI;            reg1\_read\_o <= `True\_v;            reg2\_read\_o <= `False\_v;            reg1\_addr\_o <= rs;            imm\_o <= {{16{imm[15]}},imm};            wreg\_o <= `WriteEnable;            waddr\_o <= rt;          end          6'b001011:begin            instvalid <= `InstValid;            alusel\_o <= `EXE\_ARITHMETIC;            aluop\_o <= `SLTIU;            reg1\_read\_o <= `True\_v;            reg2\_read\_o <= `False\_v;            reg1\_addr\_o <= rs;            imm\_o <= {{16{imm[15]}},imm};            wreg\_o <= `WriteEnable;            waddr\_o <= rt;          end          6'b001100:begin            instvalid <= `InstValid;            alusel\_o <= `EXE\_LOGIC;            aluop\_o <= `ANDI;            reg1\_read\_o <= `True\_v;            reg2\_read\_o <= `False\_v;            reg1\_addr\_o <= rs;            imm\_o <= {16'b0,imm};            wreg\_o <= `WriteEnable;            waddr\_o <= rt;          end          6'b001111:begin            instvalid <= `InstValid;            alusel\_o <= `EXE\_LOGIC;            aluop\_o <= `LUI;            reg1\_read\_o <= `False\_v;            reg2\_read\_o <= `False\_v;            imm\_o <= {imm,16'b0};            wreg\_o <= `WriteEnable;            waddr\_o <= rt;          end          6'b001101:begin            instvalid <= `InstValid;            alusel\_o <= `EXE\_LOGIC;            aluop\_o <= `ORI;            reg1\_read\_o <= `True\_v;            reg2\_read\_o <= `False\_v;            reg1\_addr\_o <= rs;            imm\_o <= {16'b0,imm};            wreg\_o <= `WriteEnable;            waddr\_o <= rt;          end          6'b001110:begin            instvalid <= `InstValid;            alusel\_o <= `EXE\_LOGIC;            aluop\_o <= `XORI;            reg1\_read\_o <= `True\_v;            reg2\_read\_o <= `False\_v;            reg1\_addr\_o <= rs;            imm\_o <= {16'b0,imm};            wreg\_o <= `WriteEnable;            waddr\_o <= rt;          end          6'b000100:begin            instvalid <= `InstValid;            alusel\_o <= `EXE\_JUMP\_BRANCH;            aluop\_o <= `BEQ;            // reg1\_read\_o <= `True\_v;            // reg2\_read\_o <= `True\_v;            // reg1\_addr\_o <= rs;            // reg2\_addr\_o <= rt;              // if (reg1\_data\_i == reg2\_data\_i) begin            //   branch\_target\_address\_o <= pc\_plus\_4 + imm\_sll2\_signedext;            //   branch\_flag\_o <= `Branch;            // end            // next\_inst\_in\_delayslot\_o <= `InDelaySlot;          end          6'b000101:begin            instvalid <= `InstValid;            alusel\_o <= `EXE\_JUMP\_BRANCH;            aluop\_o <= `BNE;            // reg1\_read\_o <= `True\_v;            // reg2\_read\_o <= `True\_v;            // reg1\_addr\_o <= rs;            // reg2\_addr\_o <= rt;            // if (reg1\_data\_i != reg2\_data\_i) begin            //   branch\_target\_address\_o <= pc\_plus\_4 + imm\_sll2\_signedext;            //   branch\_flag\_o <= `Branch;            // end            // next\_inst\_in\_delayslot\_o <= `InDelaySlot;          end          6'b000001:begin            case (rt)              5'b00001:begin                instvalid <= `InstValid;                alusel\_o <= `EXE\_JUMP\_BRANCH;                aluop\_o <= `BGEZ;                // reg1\_read\_o <= `True\_v;                // reg2\_read\_o <= `False\_v;                // reg1\_addr\_o <= rs;                // if (reg1\_data\_i[31] == 1'b0 || reg1\_data\_i == `ZeroWord) begin                //   branch\_target\_address\_o <= pc\_plus\_4 + imm\_sll2\_signedext;                //   branch\_flag\_o <= `Branch;                // end                // next\_inst\_in\_delayslot\_o <= `InDelaySlot;              end              5'b00000:begin                instvalid <= `InstValid;                alusel\_o <= `EXE\_JUMP\_BRANCH;                aluop\_o <= `BLTZ;                // reg1\_read\_o <= `True\_v;                // reg2\_read\_o <= `False\_v;                // reg1\_addr\_o <= rs;                // if (reg1\_data\_i[31] == 1'b1) begin                //   branch\_target\_address\_o <= pc\_plus\_4 + imm\_sll2\_signedext;                //   branch\_flag\_o <= `Branch;                // end                // next\_inst\_in\_delayslot\_o <= `InDelaySlot;              end              5'b10001:begin                instvalid <= `InstValid;                alusel\_o <= `EXE\_JUMP\_BRANCH;                aluop\_o <= `BGEZAL;                // reg1\_read\_o <= `True\_v;                // reg2\_read\_o <= `False\_v;                // reg1\_addr\_o <= rs;                wreg\_o <= `WriteEnable;                waddr\_o <= 5'd31;                // link\_addr\_o <= pc\_plus\_8;                // if (reg1\_data\_i[31] == 1'b0) begin                //   branch\_target\_address\_o <= pc\_plus\_4 + imm\_sll2\_signedext;                //   branch\_flag\_o <= `Branch;                // end                // next\_inst\_in\_delayslot\_o <= `InDelaySlot;              end              5'b10000:begin                instvalid <= `InstValid;                alusel\_o <= `EXE\_JUMP\_BRANCH;                aluop\_o <= `BLTZAL;                // reg1\_read\_o <= `True\_v;                // reg2\_read\_o <= `False\_v;                // reg1\_addr\_o <= rs;                wreg\_o <= `WriteEnable;                waddr\_o <= 5'd31;                // link\_addr\_o <= pc\_plus\_8;                // if (reg1\_data\_i[31] == 1'b1) begin                //   branch\_target\_address\_o <= pc\_plus\_4 + imm\_sll2\_signedext;                //   branch\_flag\_o <= `Branch;                // end                // next\_inst\_in\_delayslot\_o <= `InDelaySlot;              end            endcase          end          6'b000111:begin            instvalid <= `InstValid;            alusel\_o <= `EXE\_JUMP\_BRANCH;            aluop\_o <= `BGTZ;            // reg1\_read\_o <= `True\_v;            // reg2\_read\_o <= `False\_v;            // reg1\_addr\_o <= rs;              // if (reg1\_data\_i[31] == 1'b0 && reg1\_data\_i != `ZeroWord) begin            //   branch\_target\_address\_o <= pc\_plus\_4 + imm\_sll2\_signedext;            //   branch\_flag\_o <= `Branch;            // end            // next\_inst\_in\_delayslot\_o <= `InDelaySlot;          end          6'b000110:begin            instvalid <= `InstValid;            alusel\_o <= `EXE\_JUMP\_BRANCH;            aluop\_o <= `BLEZ;            // reg1\_read\_o <= `True\_v;            // reg2\_read\_o <= `False\_v;            // reg1\_addr\_o <= rs;            // if (reg1\_data\_i[31] == 1'b1 || reg1\_data\_i == `ZeroWord) begin            //   branch\_target\_address\_o <= pc\_plus\_4 + imm\_sll2\_signedext;            //   branch\_flag\_o <= `Branch;            // end            // next\_inst\_in\_delayslot\_o <= `InDelaySlot;          end          6'b000010:begin            instvalid <= `InstValid;            alusel\_o <= `EXE\_JUMP\_BRANCH;            aluop\_o <= `J;            // reg1\_read\_o <= `False\_v;            // reg2\_read\_o <= `False\_v;            // branch\_target\_address\_o <= {pc\_plus\_4[31:28],instr\_index,2'b0};            // branch\_flag\_o <= `Branch;            // next\_inst\_in\_delayslot\_o <= `InDelaySlot;          end          6'b000011:begin            instvalid <= `InstValid;            alusel\_o <= `EXE\_JUMP\_BRANCH;            aluop\_o <= `JAL;            // reg1\_read\_o <= `False\_v;            // reg2\_read\_o <= `False\_v;            wreg\_o <= `WriteEnable;            waddr\_o <= 5'd31;            // link\_addr\_o <= pc\_plus\_8;            // branch\_target\_address\_o <= {pc\_plus\_4[31:28],instr\_index,2'b0};            // branch\_flag\_o <= `Branch;            // next\_inst\_in\_delayslot\_o <= `InDelaySlot;          end          6'b100000:begin            instvalid <= `InstValid;            alusel\_o <= `EXE\_LOAD\_STORE;            aluop\_o <= `LB;            // reg1\_read\_o <= `True\_v;            // reg2\_read\_o <= `False\_v;            // reg1\_addr\_o <= base;            wreg\_o <= `WriteEnable;            waddr\_o <= rt;          end          6'b100100:begin            instvalid <= `InstValid;            alusel\_o <= `EXE\_LOAD\_STORE;            aluop\_o <= `LBU;            // reg1\_read\_o <= `True\_v;            // reg2\_read\_o <= `False\_v;            // reg1\_addr\_o <= base;            wreg\_o <= `WriteEnable;            waddr\_o <= rt;          end          6'b100001:begin            instvalid <= `InstValid;            alusel\_o <= `EXE\_LOAD\_STORE;            aluop\_o <= `LH;            // reg1\_read\_o <= `True\_v;            // reg2\_read\_o <= `False\_v;            // reg1\_addr\_o <= base;            wreg\_o <= `WriteEnable;            waddr\_o <= rt;          end          6'b100101:begin            instvalid <= `InstValid;            alusel\_o <= `EXE\_LOAD\_STORE;            aluop\_o <= `LHU;            // reg1\_read\_o <= `True\_v;            // reg2\_read\_o <= `False\_v;            // reg1\_addr\_o <= base;              wreg\_o <= `WriteEnable;            waddr\_o <= rt;          end          6'b100011:begin            instvalid <= `InstValid;            alusel\_o <= `EXE\_LOAD\_STORE;            aluop\_o <= `LW;            // reg1\_read\_o <= `True\_v;            // reg2\_read\_o <= `False\_v;            // reg1\_addr\_o <= base;              wreg\_o <= `WriteEnable;            waddr\_o <= rt;          end          6'b101000:begin            instvalid <= `InstValid;            alusel\_o <= `EXE\_LOAD\_STORE;            aluop\_o <= `SB;            // reg1\_read\_o <= `True\_v;            // reg2\_read\_o <= `True\_v;            // reg1\_addr\_o <= base;            // reg2\_addr\_o <= rt;            wreg\_o <= `WriteDisable;          end          6'b101001:begin            instvalid <= `InstValid;            alusel\_o <= `EXE\_LOAD\_STORE;            aluop\_o <= `SH;            // reg1\_read\_o <= `True\_v;            // reg2\_read\_o <= `True\_v;            // reg1\_addr\_o <= base;            // reg2\_addr\_o <= rt;            wreg\_o <= `WriteDisable;          end          6'b101011:begin            instvalid <= `InstValid;            alusel\_o <= `EXE\_LOAD\_STORE;            aluop\_o <= `SW;            // reg1\_read\_o <= `True\_v;            // reg2\_read\_o <= `True\_v;            // reg1\_addr\_o <= base;            // reg2\_addr\_o <= rt;            wreg\_o <= `WriteDisable;          end          6'b010000:begin            case (rs)              5'b00000:begin                instvalid <= `InstValid;                alusel\_o <= `EXE\_MOVE;                aluop\_o <= `MFC0;                // reg1\_read\_o <= `False\_v;                // reg2\_read\_o <= `False\_v;                wreg\_o <= `WriteEnable;                waddr\_o <= rt;              end              5'b00100:begin                instvalid <= `InstValid;                alusel\_o <= `EXE\_MOVE;                aluop\_o <= `MTC0;                // reg1\_read\_o <= `True\_v;                // reg2\_read\_o <= `False\_v;                // reg1\_addr\_o <= rt;                wreg\_o <= `WriteDisable;              end              5'b10000:begin                instvalid <= `InstValid;                alusel\_o <= `EXE\_NOP;                aluop\_o <= `ERET;                // reg1\_read\_o <= `False\_v;                // reg2\_read\_o <= `False\_v;                wreg\_o <= `WriteDisable;                // excepttype\_is\_eret <= `True\_v;              end            endcase          end          default:begin            end        endcase // opcode      end    end  // pc ctrl    always @ (\*) begin // 从l1来判断l2是否适合运行，不代表l1跑了没      if (rst == `RstEnable) begin        l1\_run <= `False\_v;      end      else begin        case (alusel\_o)          `EXE\_JUMP\_BRANCH:l1\_run <= `False\_v;          default:l1\_run <= `True\_v;        endcase      end    end    always @ (\*) begin      if (rst == `RstEnable) begin        l2\_run <= `False\_v;      end      else begin        case (alusel\_o)          `EXE\_NOP,`EXE\_LOGIC,`EXE\_SHIFT,`EXE\_ARITHMETIC,`EXE\_MUL:l2\_run <= `True\_v;          `EXE\_MOVE,`EXE\_JUMP\_BRANCH,`EXE\_LOAD\_STORE:l2\_run <= `False\_v;          default:l2\_run <= `False\_v;        endcase      end    end  endmodule |  line2\_if\_id  |  | | --- | | `include "defines.v"  module line2\_if\_id(      input wire clk,      input wire rst,      input wire [`StallBus] stall,      input wire flush,      input wire [`InstAddrBus] if\_pc,      input wire if\_reg1\_read,      input wire if\_reg2\_read,      input wire [`RegAddrBus] if\_reg1\_addr,      input wire [`RegAddrBus] if\_reg2\_addr,      input wire [`AluOpBus] if\_aluop,      input wire [`AluSelBus] if\_alusel,      input wire [`RegAddrBus] if\_waddr,      input wire if\_wreg,      input wire [`RegBus] if\_imm,      output reg [`InstAddrBus] id\_pc,      output reg id\_reg1\_read,      output reg id\_reg2\_read,      output reg [`RegAddrBus] id\_reg1\_addr,      output reg [`RegAddrBus] id\_reg2\_addr,      output reg [`AluOpBus] id\_aluop,      output reg [`AluSelBus] id\_alusel,      output reg [`RegAddrBus] id\_waddr,      output reg id\_wreg,      output reg [`RegBus] id\_imm  );      always @ (posedge clk) begin          if (rst == `RstEnable) begin              id\_pc <= `ZeroWord;              id\_reg1\_read <= `False\_v;              id\_reg2\_read <= `False\_v;              id\_reg1\_addr <= `NOPRegAddr;              id\_reg2\_addr <= `NOPRegAddr;              id\_aluop <= `NOP;              id\_alusel <= `EXE\_NOP;              id\_waddr <= `NOPRegAddr;              id\_wreg <= `WriteDisable;              id\_imm <= `ZeroWord;          end          else if (flush == `True\_v) begin              id\_pc <= `ZeroWord;              id\_reg1\_read <= `False\_v;              id\_reg2\_read <= `False\_v;              id\_reg1\_addr <= `NOPRegAddr;              id\_reg2\_addr <= `NOPRegAddr;              id\_aluop <= `NOP;              id\_alusel <= `EXE\_NOP;              id\_waddr <= `NOPRegAddr;              id\_wreg <= `WriteDisable;              id\_imm <= `ZeroWord;          end          else if (stall[2] == `Stop && stall[3] == `NoStop) begin              id\_pc <= `ZeroWord;              id\_reg1\_read <= `False\_v;              id\_reg2\_read <= `False\_v;              id\_reg1\_addr <= `NOPRegAddr;              id\_reg2\_addr <= `NOPRegAddr;              id\_aluop <= `NOP;              id\_alusel <= `EXE\_NOP;              id\_waddr <= `NOPRegAddr;              id\_wreg <= `WriteDisable;              id\_imm <= `ZeroWord;          end          else if (stall[2] == `NoStop) begin              id\_pc <= if\_pc;              id\_reg1\_read <= if\_reg1\_read;              id\_reg2\_read <= if\_reg2\_read;              id\_reg1\_addr <= if\_reg1\_addr;              id\_reg2\_addr <= if\_reg2\_addr;              id\_aluop <= if\_aluop;              id\_alusel <= if\_alusel;              id\_waddr <= if\_waddr;              id\_wreg <= if\_wreg;              id\_imm <= if\_imm;          end      end  endmodule |  line2\_if\_ex  |  | | --- | | `include "defines.v"  module line2\_ex(      input wire rst,      input wire [`InstAddrBus] pc\_i,      output wire [`InstAddrBus] pc\_o,      input wire reg1\_read\_i,      input wire reg2\_read\_i,      input wire [`RegAddrBus] reg1\_addr\_i,      input wire [`RegAddrBus] reg2\_addr\_i,      input wire [`AluOpBus] aluop\_i,      input wire [`AluSelBus] alusel\_i,      input wire [`RegAddrBus] waddr\_i,      input wire wreg\_i,      input wire [`RegBus] imm\_i,      input wire [`RegBus] reg1\_rdata\_i,      input wire [`RegBus] reg2\_rdata\_i,      // input wire l1\_ex\_we,      // input wire [`RegAddrBus] l1\_ex\_waddr,      // input wire [`RegBus] l1\_ex\_wdata,      output reg [`RegAddrBus] waddr\_o,      output reg wreg\_o,      output reg [`RegBus] wdata\_o  );      wire [`RegBus] reg1\_i;      wire [`RegBus] reg2\_i;      // assign reg1\_i = (reg1\_addr\_i == l1\_ex\_waddr) && (l1\_ex\_we == `WriteEnable) ? l1\_ex\_wdata : reg1\_read\_i ? reg1\_rdata\_i : imm\_i;      // assign reg2\_i = (reg2\_addr\_i == l1\_ex\_waddr) && (l1\_ex\_we == `WriteEnable) ? l1\_ex\_wdata : reg2\_read\_i ? reg2\_rdata\_i : imm\_i;      assign reg1\_i = reg1\_read\_i ? reg1\_rdata\_i : imm\_i;      assign reg2\_i = reg2\_read\_i ? reg2\_rdata\_i : imm\_i;      reg [`RegBus] logic\_o;      reg [`RegBus] shift\_o;      reg [`RegBus] move\_o;      reg [`RegBus] HI;      reg [`RegBus] LO;      reg [`RegBus] arithmetic\_o;      reg [`DoubleRegBus] result\_mul; // 保存乘法结果，宽度64bit      assign pc\_o = pc\_i;  // arithmetic\_o      wire reg1\_eq\_reg2; // 第一个操作数是否等于第二个操作数      wire reg1\_lt\_reg2; // 第一个操作数是否小于第二个操作数      wire [`RegBus] reg2\_i\_mux; // 保存输入的第二个操作数的补码      wire [`RegBus] reg1\_i\_not; // 保存输入的第一个操作数取反后的值      wire [`RegBus] result\_sum; // 保存加法结果      assign reg2\_i\_mux = ((aluop\_i == `SUB) || (aluop\_i == `SUBU) || (aluop\_i == `SLT) || (aluop\_i == `SLTI)) ? (~reg2\_i)+1 : reg2\_i;      assign result\_sum = reg1\_i + reg2\_i\_mux;      assign ov\_sum = ((!reg1\_i[31] && !reg2\_i\_mux[31] && result\_sum[31]) || (reg1\_i[31] && reg2\_i\_mux[31]) && (!result\_sum[31]));      assign reg1\_lt\_reg2 = ((aluop\_i == `SLT) || (aluop\_i == `SLTI)) ?                              ((reg1\_i[31] && !reg2\_i[31]) ||                              (!reg1\_i[31] && !reg2\_i[31] && result\_sum[31]) ||                              (reg1\_i[31] && reg2\_i[31] && result\_sum[31])) : (reg1\_i < reg2\_i);      assign reg1\_i\_not = ~reg1\_i;      always @ (\*) begin          if (rst == `RstEnable) begin              arithmetic\_o <= `ZeroWord;          end          else begin              case (aluop\_i)                  `ADD,`ADDU,`ADDI,`ADDIU:begin                      arithmetic\_o <= result\_sum;                  end                  `SUB,`SUBU:begin                      arithmetic\_o <= result\_sum;                  end                  `SLT,`SLTI,`SLTU,`SLTIU:begin                      arithmetic\_o <= reg1\_lt\_reg2;                  end                  default:begin                      arithmetic\_o <= `ZeroWord;                  end              endcase          end      end  // result\_mul      always @ (\*) begin          if (rst == `RstEnable) begin              result\_mul <= {`ZeroWord,`ZeroWord};          end          else begin          case (aluop\_i)              `MUL:begin                  result\_mul <= $signed(reg1\_i) \* $signed(reg2\_i);              end              `MULT:begin                  result\_mul <= $signed(reg1\_i) \* $signed(reg2\_i);              end              `MULTU:begin                  result\_mul <= $unsigned(reg1\_i) \* $unsigned(reg2\_i);              end              default:begin                  result\_mul <= {`ZeroWord,`ZeroWord};              end          endcase          end      end // always  // logic\_o      always @ (\*) begin          if (rst == `RstEnable) begin              logic\_o <= `ZeroWord;          end          else begin              case (aluop\_i)                  `OR,`ORI:begin                      logic\_o <= reg1\_i | reg2\_i;                  end // OR                  `AND,`ANDI:begin                      logic\_o <= reg1\_i & reg2\_i;                  end // AND                  `NOR:begin                      logic\_o <= ~(reg1\_i | reg2\_i);                  end                  `XOR,`XORI:begin                      logic\_o <= reg1\_i ^ reg2\_i;                  end                  `LUI:begin                      logic\_o <= reg1\_i;                  end                  default:begin                      logic\_o <= `ZeroWord;                  end // default              endcase          end // if      end //always  // shift\_o      always @ (\*) begin          if (rst == `RstEnable) begin              shift\_o <= `ZeroWord;          end          else begin              case (aluop\_i)                  `SLLV:begin                      shift\_o <= reg2\_i << reg1\_i[4:0];                  end                  `SLL:begin                      shift\_o <= reg2\_i << reg1\_i;                  end                  `SRAV:begin                      shift\_o <= ({32{reg2\_i[31]}} << (6'd32 - {1'b0, reg1\_i[4:0]})) | reg2\_i >> reg1\_i[4:0];                  end                  `SRA:begin                      shift\_o <= ({32{reg2\_i[31]}} << (6'd32 - {1'b0, reg1\_i[4:0]})) | reg2\_i >> reg1\_i[4:0];                  end                  `SRLV:begin                      shift\_o <= reg2\_i >> reg1\_i[4:0];                  end                  `SRL:begin                      shift\_o <= reg2\_i >> reg1\_i;                  end                  default:begin                      shift\_o <= `ZeroWord;                  end              endcase          end      end  // wdata\_o & ovassert      always @ (\*) begin          waddr\_o <= waddr\_i;          wreg\_o <= wreg\_i;          case (alusel\_i)              `EXE\_LOGIC:begin                  wdata\_o <= logic\_o;              end // EXE\_LOGIC              `EXE\_SHIFT:begin                  wdata\_o <= shift\_o;              end // EXE\_SHIFT              `EXE\_ARITHMETIC:begin                  wdata\_o <= arithmetic\_o;              end              `EXE\_MUL:begin                  wdata\_o <= result\_mul[31:0];              end              default:begin                  wdata\_o <= `ZeroWord;              end          endcase      end // always  endmodule |  line2\_dcache\_mem  |  | | --- | | `include "defines.v"  module line2\_dcache\_mem(      input wire clk,      input wire rst,      input wire [`StallBus] stall,      input wire flush,      input wire [`InstAddrBus] dcache\_pc,      input wire dcache\_wreg,      input wire [`RegAddrBus] dcache\_waddr,      input wire [`RegBus] dcache\_wdata,      output reg [`InstAddrBus] mem\_pc,      output reg mem\_wreg,      output reg [`RegAddrBus] mem\_waddr,      output reg [`RegBus] mem\_wdata  );      always @ (posedge clk) begin          if (rst == `RstEnable) begin              mem\_pc <= `ZeroWord;              mem\_wreg <= `WriteDisable;              mem\_waddr <= `NOPRegAddr;              mem\_wdata <= `ZeroWord;          end          else if (flush == `True\_v) begin              mem\_pc <= `ZeroWord;              mem\_wreg <= `WriteDisable;              mem\_waddr <= `NOPRegAddr;              mem\_wdata <= `ZeroWord;          end          else if (stall[6] == `Stop && stall[7] == `NoStop) begin              mem\_pc <= `ZeroWord;              mem\_wreg <= `WriteDisable;              mem\_waddr <= `NOPRegAddr;              mem\_wdata <= `ZeroWord;          end          else if (stall[6] == `NoStop) begin              mem\_pc <= dcache\_pc;              mem\_wreg <= dcache\_wreg;              mem\_waddr <= dcache\_waddr;              mem\_wdata <= dcache\_wdata;          end      end  endmodule |  line2\_mem\_wb  |  | | --- | | `include "defines.v"  module line2\_mem\_wb(      input wire clk,      input wire rst,      input wire [`StallBus] stall,      input wire flush,      input wire [31:0] mem\_pc,      input wire mem\_wreg,      input wire [`RegAddrBus] mem\_waddr,      input wire [`RegBus] mem\_wdata,      output reg [31:0] wb\_pc,      output reg wb\_wreg,      output reg [`RegAddrBus] wb\_waddr,      output reg [`RegBus] wb\_wdata  );      always @ (posedge clk) begin          if (rst == `RstEnable) begin              wb\_pc <= `ZeroWord;              wb\_wreg <= `WriteDisable;              wb\_waddr <= `NOPRegAddr;              wb\_wdata <= `ZeroWord;          end          else if (flush == `True\_v) begin              wb\_pc <= `ZeroWord;              wb\_wreg <= `WriteDisable;              wb\_waddr <= `NOPRegAddr;              wb\_wdata <= `ZeroWord;          end          else if (stall[7] == `Stop && stall[8] == `NoStop) begin              wb\_pc <= `ZeroWord;              wb\_wreg <= `WriteDisable;              wb\_waddr <= `NOPRegAddr;              wb\_wdata <= `ZeroWord;          end          else if (stall[7] == `NoStop) begin              wb\_pc <= mem\_pc;              wb\_wreg <= mem\_wreg;              wb\_waddr <= mem\_waddr;              wb\_wdata <= mem\_wdata;          end      end  endmodule |  四、实验实现截图五、收获心得体会 在普通七级流水的基础上加了一小块缓存来保存已经运行过的指令，当探测器检测到当前正要跑的指令和下一条指令都在缓存中时，则将下一条指令送入辅助流水线实现双发射（具体实现的时候还需要符合一些指令搭配，避开一下相关问题）因为这个双发只在循环时有效果，所以只能算是伪顺序双发。  这次最终的实验是在我们自己定义的 CPU 上实现扩展功能，这不仅以增强对流水线 cpu 的理解和运用能力，还增加了我们对FPGA的兴趣和爱好，甚至于自豪感，还有自主探索能力的提高。希望这次作业能向老师交上一份满意的答卷，以回报方老师和助教一个学期以来的指导与付出。其次，通过本次 CPU 设计大实验，我充分感受到了实验的意义并不是一项老师给学生的任务，而是在实验全过程后无论是对知识还是技术 都有了更深一层次的理解，这种理解不是学习课本知识能达到的。同时实验这种摸索、探索、钻研、纠错的循环迭代中，领悟到科研的魅力。 还有仿真也是实验中不可或缺的步骤和方法，debug 时运用仿真，看每个时钟周期每个变量的状态，可以很快找到问题所在及解决办法，不至于像以前那样一头雾水，浪费时间与精力。  早在大二第二学期《计算机组成原理》开课时，赵文彬老师就曾说过“这门课有一个课设，是让你们自己造一台CPU”，那时觉得是天方夜谭：本科生怎么可能做得出来CPU？后面经过学习与主动了解，明白了“造CPU”的具体要求，畏难情绪少了许多。计组算得上是对于课程考试并不十分擅长的我学得比较透彻的一门课，个中原因可能是自身对硬件比较喜爱吧。本次课程设计的一个遗憾是：我们没有足够的时间来将CPU烧到FPGA上面，完成硬件演示，功底还是差了许多。  本次实验 debug 过程中还学会利用系统弹出的错误、警告、其他能 等所有信息来挖掘对应的错误，从而纠正它的时候可以对症下药，快捷很多。 最后，实验的完成永远都没有止境，在我们搭的 CPU 上每个同学都有自己的想法、创意和亮点，课余有时间我们可以相互沟通交流改进。 总之，在这一学期的实验中学到了很多，特别是这最后一次。知识 巩固与方法总结，兴趣培养与成就感提升，收获颇丰，这都要感谢老师的指导与付出，谢谢！ |