

（深圳）

# 课程报告

开课学期： 2023夏季

课程名称： 计算机设计与实践

项目名称： 基于miniRV/LA的SoC设计

项目类型： 综合设计型

课程学时： 56 地点： t2612

学生班级： 6

学生学号： 210810302

学生姓名： 陈志权

评阅教师：

报告成绩：

实验与创新实践教育中心制

2023年7月

注：本设计报告中各个部分如果页数不够，请同学们自行扩页。原则上一定要把报告写详细，能说明设计的成果、特色和过程。报告应该详细叙述整体设计，以及设计中的每个模块。设计报告将是评定每个人成绩的重要组成部分（**设计内容及报告写作**都作为评分依据）。

|  |
| --- |
| 设计概述（罗列出所有实现的指令，以及单周期/流水线CPU频率） |
| 实现了miniRV的基本24条指令：add,sub,and,or,xor,sll,srl,sra,  addi,andi,ori,xori,slli,srli,srai,lw,jalr,  sw,  beq,bne,blt,bge,  lui,  jal.  单周期CPU频率25MHz，流水线CPU频率100MHz.(其实是懒得测更高的) |
| 设计的主要特色（除基本要求以外的设计） |
| 控制冒险使用静态预测，预测不跳转，之后再由EX阶段的结果由控制冒险检测模块对IF/ID，ID/EX模块以及PC做出裁定。 |
| 资源使用、功耗数据截图（Post Implementation；含单周期、流水线2个截图） |
| 以下是示例，请贴自己的图。  单周期  流水线： |

1 单周期CPU设计与实现

1.1 单周期CPU数据通路设计

|  |
| --- |
| 要求：贴出完整的单周期数据通路图，无需画出模块内的具体逻辑，但要标出模块的接口信号名、模块之间信号线的信号名和位宽，并用文字阐述各模块的功能。 |
| NPC：计算PC的下一次取值。  PC：程序计数器，每个clk上升沿更新，给出当前取指令的地址。  IROM：根据pc的值，读出对应的指令。  Controller：根据指令给出各种控制信号，如符号扩展信号sext\_op,ALU计算信号alu\_op等等。  SEXT：根据sext\_op和给出的指令，扩展出正确的立即数。  RF：寄存器堆模块，读、写寄存器。  ALU：算术逻辑单元，根据控制信号进行不同数据和不同类型的计算，并给出计算结果c和B型指令的比较结果f。  DRAM：数据RAM，读写内存。 |

1.2 单周期CPU模块详细设计

|  |
| --- |
| 要求：以表格的形式列出各个部件的接口信号、位宽、功能描述等，并结合图、表、核心代码等形象化工具和手段，详细描述各个部件的关键实现。 |
| 个人认为表格不如模块声明直观，因此下述给出模块接口的信号。  NPC：  module npc(    input wire[1:0] op, //npc\_op控制npc的选择    input wire br,      //B型指令跳转标志，1是跳转    input wire[31:0] offset,  //B型、Jal指令的offset    input wire[31:0] rs\_imm,  //jalr指令的跳转目的地址    input wire[31:0] pc,  //当前pc    output wire[31:0] pc4,  //当前pc+4    output reg[31:0] npc  //下一个pc取值  );  关键实现：一个选择判断。  always @(\*) begin    if(op == `C\_NPC\_PC4) npc = pc+4;    else if(op == `C\_NPC\_JALR) npc = rs\_imm;    else if(op == `C\_NPC\_B && br == 1) npc = pc+offset;    else if(op == `C\_NPC\_B && br == 0) npc = pc+4;    else if(op == `C\_NPC\_JAL) npc = pc+offset;  end  PC：  module pc(    input wire rst, //复位信号    input wire clk, //cpu时钟    input wire[31:0] din, //下一条pc    output reg[31:0] pc //当前pc  );  关键实现：  always @(posedge clk or posedge rst) begin    if(rst) pc <= 0;    else pc <= din;  end  IROM：      IROM Mem\_IROM (          .a          (inst\_addr),  //wire[13:0] inst\_addr          .spo        (inst)        //wire[31:0] inst      );  由于IROM使用IP核，因此无具体实现，但是两个参数位宽如注释所示。  关键实现：无。  Controller：  module control #(    //各种指令类型的opcode      localparam OP\_R    = 7'b0110011,      localparam OP\_I    = 7'b0010011,      localparam OP\_LOAD = 7'b0000011,      localparam OP\_S    = 7'b0100011,      localparam OP\_B    = 7'b1100011,      localparam OP\_LUI  = 7'b0110111,      localparam OP\_JAL  = 7'b1101111,      localparam OP\_JALR = 7'b1100111  )  (    input wire[31:0] inst,  //当前指令    output reg[1:0] npc\_op, //给出npc的op    output reg rf\_we,       //给出寄存器写使能    output reg[1:0] rf\_wsel,//给出寄存器的数据选择信号    output reg[2:0] sext\_op,//给出立即数扩展信号    output reg alub\_sel,    //给出alu第二个运算数据的数据选择信号    output reg dram\_we,     //给出DRAM的写使能    output reg[3:0] alu\_op  //给出ALU的计算类型信号  );  关键实现：以最复杂的alu\_op为例子，我们先按照opcode的类型对指令大概的分类，之后再通过funct3细分不同指令。  //alu\_op  always @ (\*) begin      case (opcode)          OP\_R: begin              case (funct3)                  3'b000 : alu\_op = funct7[5] ? `SUB : `ADD;                  3'b111 : alu\_op = `AND;                  3'b110 : alu\_op = `OR ;                  3'b100 : alu\_op = `XOR;                  3'b001 : alu\_op = `SLL;                  3'b101 : alu\_op = funct7[5] ? `SRA : `SRL;                  default: alu\_op = `AND;              endcase          end          OP\_I: begin              case (funct3)                  3'b000 : alu\_op = `ADD;                  3'b111 : alu\_op = `AND;                  3'b110 : alu\_op = `OR ;                  3'b100 : alu\_op = `XOR;                  3'b001 : alu\_op = `SLL;                  3'b101 : alu\_op = funct7[5] ? `SRA : `SRL;                  default: alu\_op = `AND;              endcase          end          OP\_LOAD, OP\_S, OP\_JALR:              alu\_op = `ADD;          OP\_B:begin              case(funct3)                3'b000 : alu\_op = `BEQ;                3'b001 : alu\_op = `BNE;                3'b100 : alu\_op = `BLT;                3'b101 : alu\_op = `BGE;                default: alu\_op = `BEQ;              endcase          end          default:              alu\_op = `AND;      endcase  end  SEXT：  module sext(    input wire[31:0] din, //当前指令    input wire[2:0] op,   //立即数扩展类型信号    output reg[31:0] ext  //输出的立即数  );  关键实现：采用拼接运算符一步到位。  //SEXT  wire sign=din[31];  always @(\*) begin    if(op == `SEXT\_R) ext=0;    else if(op == `SEXT\_I) ext={{20{sign}},{din[31:20]}};    else if(op == `SEXT\_MOVE) ext={{27{1'b0}},{din[24:20]}};    else if(op == `SEXT\_S) ext={{20{sign}},{din[31:25]},{din[11:7]}};    else if(op == `SEXT\_B) ext={{19{sign}},{din[31]},{din[7]},{din[30:25]},{din[11:8]},{1'b0}};    else if(op == `SEXT\_U) ext={{din[31:12]},{12{1'b0}}};    else if(op == `SEXT\_J) ext={{11{sign}},{din[31]},{din[19:12]},{din[20]},{din[30:21]},{1'b0}};  end  RF：  module RF(    input wire clk,     //时钟    input wire[4:0] rR1,//一号寄存器地址    input wire[4:0] rR2,//二号寄存器地址    input wire[4:0] wR, //写寄存器地址    input wire we,      //写寄存器使能                      //the following four datas are components of wD    input wire[1:0] rf\_wsel,//写寄存器数据选择信号    input wire[31:0] pc4, //from npc，被选择数据之一    input wire[31:0] ext, //from sext，被选择数据之一    input wire[31:0] alu\_c, //from alu，被选择数据之一    input wire[31:0] rd,  //from dram，被选择数据之一    output wire[31:0] rD1,//rR1地址读出的寄存器值    output wire[31:0] rD2,//rR2地址读出的寄存器值    output reg[31:0] wD  //only for debug,仅用于trace测试的时候给出写入RF的值  );  关键实现：读寄存器采用组合逻辑，下面是写寄存器赋值语句。  //RF  reg[31:0] rf[31:0];  //wb,write  always @(posedge clk) begin      if(we && (wR != 5'b0)) rf[wR] <= wD;  end  ALU:  module alu(    input wire[31:0] rs1, //rD1    input wire[31:0] rs2, //rD2    input wire[31:0] imm, //立即数    input wire alub\_sel,  //第二个数据的选择信号    input wire[3:0] alu\_op,   //ALU运算类型信号    output wire[31:0] alu\_c,  //ALU结果    output wire alu\_f     //用于B型指令的比较结果标志  );  关键实现:给出各种运算的代码，也就是case来选择。先是多路选择给出两个运算数据，后面根据op进行相应的计算。    wire[31:0] dataA = rs1;    wire[31:0] dataB = alub\_sel ? imm : rs2;    reg[31:0] result;      always @(\*) begin          case(alu\_op)              `ADD : result = dataA + dataB;              `SUB : result = dataA - dataB;              `AND : result = dataA & dataB;              `OR  : result = dataA | dataB;              `XOR : result = dataA ^ dataB;              `SLL : result = dataA << dataB[4:0];              `SRL : result = dataA >> dataB[4:0];              `SRA : result = ($signed(dataA)) >>> dataB[4:0];              default : result = 0;          endcase      end  DRAM：      DRAM Mem\_DRAM (          .clk        (clk\_bridge2dram),  //时钟          .a          (addr\_bridge2dram[15:2]), //读、写地址,14位地址          .spo        (rdata\_dram2bridge),//读出的数据，32位数据          .we         (wen\_bridge2dram),  //写使能，1位写使能          .d          (wdata\_bridge2dram) //要写入的数据，32位数据      );  关键实现:IP核生成未具体实现。 |
|  |

1.3 单周期CPU仿真及结果分析

|  |
| --- |
| 要求：包含逻辑运算、访存、分支跳转三类指令的仿真截图及波形分析；每类指令的截图和分析中，至少包含1条具体指令；截图需包含信号名和关键信号。 |
| 选取trace的lw.dump其中一段来分析,蓝色的是要分析的指令:  00000124 <test\_12>:  124: 00c00193 addi x3,x0,12  128: 00000213 addi x4,x0,0  12c: 000020b7 lui x1,0x2  130: 00408093 addi x1,x1,4 # 2004 <tdat2>  134: 0040a703 lw x14,4(x1)  138: 00070313 addi x6,x14,0  13c: 0ff013b7 lui x7,0xff01  140: ff038393 addi x7,x7,-16 # ff00ff0 <\_end+0xfefefe0>  144: 14731663 bne x6,x7,290 <fail>  148: 00120213 addi x4,x4,1 # 1 <\_start+0x1>  14c: 00200293 addi x5,x0,2  150: fc521ee3 bne x4,x5,12c <test\_12+0x8>   1. pc=0x00000130时，指令为addi。   可以发现此时rD1和ext都是正确的，相加后得到结果alu\_c=0x00002004,然后在下一个posedge clk被写入了x1，符合预期。   1. pc=0x00000134时，指令lw。 |
| 可以发现此时rD1和ext也是正确的，相加后得到结果alu\_c=0x00002008,直接读出rd=0x0ff00ff0,下一个clk写入x14，符合预期。   1. pc=0x00000150时,指令为bne。   根据反汇编的注释，可以发现bne后，pc跳到了0x0000012c这一条指令地址，而根据我们的波形图，也发现了由于rD1不等于rD2,因此发生跳转，pc+ext后等于0x0000012c，从而跳到了该处，符合预期。 |

2 流水线CPU设计与实现

2.1 流水线CPU数据通路

|  |
| --- |
| 要求：贴出完整的流水线数据通路图，无需画出模块内的具体逻辑，但要标出模块的接口信号名、模块之间信号线的信号名和位宽，并用文字阐述各模块的功能。  此外，数据通路图应当能体现出流水线是如何划分的，并用文字阐述每个流水级具备什么功能、需要完成哪些操作。 |
| NPC：计算PC的下一次取值。  PC：程序计数器，每个clk上升沿更新，给出当前取指令的地址。  IROM：根据pc的值，读出对应的指令。  IF/ID:流水寄存器，根据取指阶段的信号，在下一个时钟周期传给ID阶段。  Controller：根据指令给出各种控制信号，如符号扩展信号sext\_op,ALU计算信号alu\_op等等。  SEXT：根据sext\_op和给出的指令，扩展出正确的立即数。  RF：寄存器堆模块，读、写寄存器。  ID/EX:流水寄存器，根据译码阶段信号，在一个时钟周期传给EX阶段。  ALU：算术逻辑单元，根据控制信号进行不同数据和不同类型的计算，并给出计算结果c和B型指令的比较结果f。  EX/MEM:流水寄存器，根据执行阶段信号，在一个时钟周期传给MEM阶段。  DRAM:数据存储器，读、写数据。  MEM/WB:流水寄存器，根据访存阶段信号，在一个时钟周期传给WB阶段。  数据冒险检测模块(data\_hazard\_detection)：根据ID、EX、MEM、WB的信号判断是否发生数据冒险，是否需要数据前递，具体地接口信号与实现在下文阐述。  控制冒险检测模块(control\_hazard\_detection)：根据EX阶段的npc\_op和alu\_f即可判断jal，jalr，B型指令的跳转，从而确定是否发生控制冒险。  流水级划分：按照经典五级流水进行划分。  IF：取指阶段，取出指令。  ID：译码阶段，包括立即数扩展，读寄存器堆，生成控制信号。  EX：执行阶段，ALU的计算。  MEM：访存阶段，读或写DRAM。  WB：写回阶段，写寄存器堆。 |

2.2 流水线CPU模块详细设计

|  |
| --- |
| 要求：以表格的形式列出所有与单周期不同的部件的接口信号、位宽、功能描述等，并结合图、表、核心代码等形象化工具和手段，详细描述这些部件的关键实现。此外，如果实现了冒险控制，必须结合数据通路图，详细说明数据冒险、控制冒险的解决方法。 |
| PC：pc相比单周期增加了两个输入信号，用于检测冒险。  module pc(    input wire rst,    input wire clk,    input wire[31:0] din,    input wire data\_hazard, //数据冒险标志    input wire control\_hazard,//控制冒险标志    output reg[31:0] pc  );  关键实现：加入了对数据冒险的控制冒险的判断，要注意，控制冒险的优先级应该高于数据冒险，应该优先判断。  always @(posedge clk or posedge rst) begin    if(rst) pc <= 0;    else if(control\_hazard) pc<=din;    else if(data\_hazard) pc<=pc;    else pc <= din;  end  NPC：NPC的接口信号并有改变，但是实现改变了。  关键实现：由于我统一在EX阶段判断跳转，而我采用静态预测的方法，因此在发生跳转写回pc的时候，jal和B型指令的npc=pc+offset-8。  always @(\*) begin    if(op == `C\_NPC\_JALR) npc = rs\_imm;    else if(op == `C\_NPC\_B && br == 1) npc = pc+offset-8;    else if(op == `C\_NPC\_B && br == 0) npc = pc+4;    else if(op == `C\_NPC\_JAL) npc = pc+offset-8;    else npc = pc+4;  end  IF/ID:IF/ID流水寄存器作用很重要，它有解决数据冒险和控制冒险的作用。  module IF\_ID(    input wire clk,    input wire rst,    input wire[31:0] IF\_inst, //IF阶段的inst    input wire[31:0] IF\_pc4,  //IF阶段的pc+4    input wire data\_hazard,   //数据冒险标志    input wire control\_hazard,  //控制冒险标志，control\_hazard has the top priority!!    output reg[31:0] ID\_inst, //传给ID阶段的inst    output reg[31:0] ID\_pc4   //传给ID阶段的pc+4  );  关键实现：inst与pc4实现相同，下面以inst为例。  always @(posedge clk or posedge rst) begin    if(rst) ID\_pc4<=0;    else if(control\_hazard) ID\_pc4<=0;  //若有控制冒险则应该请空该寄存器信号    else if(data\_hazard) ID\_pc4<=ID\_pc4;//若有数据冒险，则维持一个clk不变    else ID\_pc4<=IF\_pc4;  end  Controller:多了一个输出信号，他的两位分别代表着一号寄存器和二号寄存器是否要读寄存器堆，这用于防止数据冒险误判。    output reg[1:0] rf\_re  关键实现：查表就可以得到哪些指令的reg1、reg2要读rf，哪些不要。  //rf\_re,read enable(bad name......)  always @(\*) begin    case(opcode)      OP\_I, OP\_LOAD, OP\_JALR:        rf\_re = 2'b01;      OP\_LUI, OP\_JAL:        rf\_re = 2'b00;      default:        rf\_re = 2'b11;    endcase  end  ID/EX:信号太多了，但是模式单一，看名字就能很容易知道信号的意义，故不写注释了。  module ID\_EX(    input wire clk,    input wire rst,    input wire[1:0] ID\_npc\_op,    input wire ID\_ram\_we,    input wire[3:0] ID\_alu\_op,    input wire ID\_alub\_sel,    input wire ID\_rf\_we,    input wire[1:0] ID\_rf\_wsel,    input wire[4:0] ID\_wR,    input wire[31:0] ID\_pc4,    input wire[31:0] ID\_rD1,    input wire[31:0] ID\_rD2,    input wire[31:0] ID\_ext,    output reg[1:0] EX\_npc\_op,    output reg EX\_ram\_we,    output reg[3:0] EX\_alu\_op,    output reg EX\_alub\_sel,    output reg EX\_rf\_we,    output reg[1:0] EX\_rf\_wsel,    output reg[4:0] EX\_wR,    output reg[31:0] EX\_pc4,    output reg[31:0] EX\_rD1,    output reg[31:0] EX\_rD2,    output reg[31:0] EX\_ext,    input wire control\_hazard,//控制冒险标志 two hazards have the same flush    input wire data\_hazard    //数据冒险标志  );  关键实现:对于所有的输出信号都是一样的，因此下面以EX\_npc\_op为例。  always @(posedge clk or posedge rst) begin    if(rst) EX\_npc\_op<=0;    else if(control\_hazard | data\_hazard) EX\_npc\_op<=0;//发生控制冒险和数据冒险都是清空信号    else EX\_npc\_op <= ID\_npc\_op; //正常情况下直接继承ID阶段信号  end  EX/MEM:信号多，但都简单模式单一，故不给出注释了。  module EX\_MEM(    input wire clk,    input wire rst,      input wire EX\_ram\_we,    input wire EX\_rf\_we,    input wire[1:0] EX\_rf\_wsel,    input wire[4:0] EX\_wR,    input wire[31:0] EX\_pc4,    input wire[31:0] EX\_alu\_c,    input wire[31:0] EX\_rD2,    input wire[31:0] EX\_ext,    output reg MEM\_ram\_we,    output reg MEM\_rf\_we,    output reg[1:0] MEM\_rf\_wsel,    output reg[4:0] MEM\_wR,    output reg[31:0] MEM\_pc4,    output reg[31:0] MEM\_alu\_c,    output reg[31:0] MEM\_rD2,    output reg[31:0] MEM\_ext  );  关键实现：所有信号都一样，下面以MEM\_ran\_we为例。  always @(posedge clk or posedge rst) begin    if(rst) MEM\_ram\_we <= 0;    else MEM\_ram\_we <= EX\_ram\_we; //直接继承EX阶段信号  end  MEM/WB:信号多，但都简单模式单一，故不给出注释了。    input wire clk,    input wire rst,      input wire MEM\_rf\_we,    input wire[1:0] MEM\_rf\_wsel,    input wire[4:0] MEM\_wR,    input wire[31:0] MEM\_pc4,    input wire[31:0] MEM\_alu\_c,    input wire[31:0] MEM\_rd,    input wire[31:0] MEM\_ext,    output reg WB\_rf\_we,    output reg[1:0] WB\_rf\_wsel,    output reg[4:0] WB\_wR,    output reg[31:0] WB\_pc4,    output reg[31:0] WB\_alu\_c,    output reg[31:0] WB\_rd,    output reg[31:0] WB\_ext  );  关键实现:所有信号都一样，下面以WB\_rf\_we为例。  always @(posedge clk or posedge rst) begin    if(rst) WB\_rf\_we <= 0;    else WB\_rf\_we <= MEM\_rf\_we; //直接继承MEM阶段信号  end  data\_hazard\_detection:由于要数据前递，从而信号巨多，但是同样是模式统一。关键输入信号是所有ID信号和后缀为wR(写地址)、rf\_we(写使能)、rf\_wsel(写选择)、pc4(pc+4)、ext(立即数)、alu\_c(ALU结果)。  而三个输出信号都是关键，给出了注释。  module data\_hazard\_detection(    input wire[4:0] ID\_rR1,    input wire[4:0] ID\_rR2,    input wire[1:0] ID\_rf\_re, //read enable    input wire[31:0] ID\_rD1,    input wire[31:0] ID\_rD2,    input wire[4:0] EX\_wR,    input wire EX\_rf\_we,    input wire[1:0] EX\_rf\_wsel,    input wire[31:0] EX\_pc4,    input wire[31:0] EX\_ext,    input wire[31:0] EX\_alu\_c,    input wire[4:0] MEM\_wR,    input wire MEM\_rf\_we,    input wire[1:0] MEM\_rf\_wsel,    input wire[31:0] MEM\_pc4,    input wire[31:0] MEM\_ext,    input wire[31:0] MEM\_alu\_c,    input wire[31:0] MEM\_rd,    input wire[4:0] WB\_wR,    input wire WB\_rf\_we,    input wire[1:0] WB\_rf\_wsel,    input wire[31:0] WB\_pc4,    input wire[31:0] WB\_ext,    input wire[31:0] WB\_alu\_c,    input wire[31:0] WB\_rd,    output reg[31:0] new\_rD1, //传给ID/EX的rD1被改模块所代理    output reg[31:0] new\_rD2, //传给ID/EX的rD2被改模块所代理    output wire data\_hazard   //if only we counter with load-use, it will be 1,and stop pipeline a clk  );  关键实现:  1.检测三种数据冒险的实现，由于模式统一，下面以ID、EX的数据冒险为例:  //A,ID and EX  wire rR1\_a = (ID\_rR1 == EX\_wR) & EX\_rf\_we &  ID\_rf\_re[0] & (ID\_rR1 != 5'b0);//仅当ID寄存器地址与EX写地址相同 & EX有写使能 & 这个ID寄存器地址是确实要读数据的 & ID寄存器地址不是0,那么发生了ID/EX的数据冒险   1. 输出信号data\_hazard的实现：   //这个data\_hazard不是严格意义的数据冒险，他只有在发生load-use型的数据冒险时才会是1，代表的是暂停流水线1clk，而另外几种普通的数据冒险都可以由前递解决。如下代码所示，仅当发生ID/EX数据冒险 & EX的写选择是DRAM的读出结果时才发生。  assign data\_hazard = (rR1\_a && EX\_rf\_wsel == `S\_DRAM\_rd) || (rR2\_a && EX\_rf\_wsel == `S\_DRAM\_rd);   1. 输出信号new\_rD1,new\_rD2的实现，他们实现完全一样，且三种不同的数据冒险之间差别只有可供选择数据的多少，因此下面以reg1的WB阶段的数据冒险为例。     else if(rR1\_c) begin      case(WB\_rf\_wsel)      `S\_PC4: new\_rD1=WB\_pc4;     //要写pc+4      `S\_SEXT\_ext: new\_rD1=WB\_ext;//要写立即数      `S\_ALU\_C: new\_rD1=WB\_alu\_c; //要写ALU结果      `S\_DRAM\_rd: new\_rD1=WB\_rd;  //要写DRAM读出数据      default: new\_rD1=WB\_alu\_c;      endcase    end  else new\_rD1=ID\_rD1;  control\_hazard\_detection:  module control\_hazard\_detection(    input wire[1:0] EX\_npc\_op,  //EX阶段的npc\_op    input wire alu\_f,           //EX阶段的alu\_f    output reg control\_hazard   //控制冒险标志  );  关键实现:  always @(\*) begin    if(EX\_npc\_op == `C\_NPC\_JALR || EX\_npc\_op == `C\_NPC\_JAL) control\_hazard = 1'b1;//jal、jalr都是无条件跳转    else if(EX\_npc\_op == `C\_NPC\_B && alu\_f == 1) control\_hazard = 1'b1;           //B型指令需要满足alu\_f==1    else control\_hazard = 1'b0;  end  数据冒险解决方法:   1. 普通的三种数据冒险都采用前递解决，并且ID/EX的数据冒险优先级最高。具体来说我们让ID/EX流水寄存器的ID\_rD1和ID\_rD2都由数据冒险检测模块代理，而在数据冒险检测模块里已经实现了对出限数据冒险的解决方案。 2. load-use型数据冒险无法通过前递解决，必须暂停流水线一个clk，之后load-use型数据冒险会就自动转化成了ID/MEM的普通数据冒险。 3. 暂停流水线是通过pc保持不变，IF/ID输出不变，清空ID/EX实现的。   控制冒险解决方法:   1. 静态预测不发生跳转，在EX阶段检测npc\_op和alu\_f。 2. 若是jal或者jalr或者B型指令跳转，则更改npc，从而在下一个clk，pc被更新，此时清空IF/ID,ID/EX两个寄存器。 |

2.3 流水线CPU仿真及结果分析

|  |
| --- |
| 要求：包含控制冒险和数据冒险三种情形的仿真截图，以及波形分析。若仅实现了理想流水，则此处贴上理想流水的仿真截图及详细的波形分析。 |
| 经过精心寻找，找到了一段好test段，来自lw.dump的pc=0x00000188<test\_14>和<test\_15>的前面一小段:  00000188 <test\_14>:  188: 00e00193 addi x3,x0,14  18c: 00000213 addi x4,x0,0  190: 000020b7 lui x1,0x2  194: 00008093 addi x1,x1,0 # 2000 <begin\_signature> //A类数据冒险  198: 0040a703 lw x14,4(x1)  19c: 00000013 addi x0,x0,0  1a0: 00000013 addi x0,x0,0  1a4: 00070313 addi x6,x14,0 //C类数据冒险  1a8: ff0103b7 lui x7,0xff010  1ac: f0038393 addi x7,x7,-256 # ff00ff00 <\_end+0xff00def0>  1b0: 0e731063 bne x6,x7,290 <fail>  1b4: 00120213 addi x4,x4,1 # 1 <\_start+0x1>  1b8: 00200293 addi x5,x0,2  1bc: fc521ae3 bne x4,x5,190 <test\_14+0x8> //B类数据冒险和控制冒险  000001c0 <test\_15>:  1c0: 00f00193 addi x3,x0,15  1c4: 00000213 addi x4,x0,0  1c8: 000020b7 lui x1,0x2  A类数据冒险:  从左到右分别有三个时钟周期，分别对应的是lui x1,0x2的IF、ID、EX阶段，从而是addi x1,x1,0的无、IF、ID阶段。我们来看最后一个时钟周期的情况，此时EX\_wR是1，这与rR1(ID阶段)相同，EX\_ext是0x00002000，看到rR1\_a也可以知道此处有A类数据冒险发生。我们看到寄存器堆rf[1]的值是0x00002008，若不处理冒险则ID\_rD1会是0x00002008，但是事实上他是0x00002000，这与lui x1,0x2的预期结果相符，说明我们正确前递了数据。  B类数据冒险:  从左到右的四个时钟周期，分别对应addi x4,x4,1的IF、ID、EX、MEM四个阶段，由此知道bne x4,x5,190在最后一个时钟周期处于他自己的ID阶段。因此我们来仔细看看最后一个时钟周期的情况。此时，MEM\_wR=4,这与rR1=4相等，且MEM\_rf\_we=1，说明发生了B类数据冒险，看到rR1\_b标志位正确地变为1。rf[1]=0x00002000，而MEM\_alu\_c=0x00000001,最终ID\_rD1=MEM\_alu\_c=0x00000001,说明数据冒险被正确处理，数据正确前递，符合预期。  C类数据冒险:  从左到右的五个时钟周期，分别对应lw x14,4(x1)的IF,ID,EX,MEM,WB五个阶段，从而addi x6,x14,0指令在最后一个时钟周期对应他自己的ID阶段。因此我们来仔细看看最后一个时钟周期发生的事。此时WB\_wR为14，这与读寄存器地址rR1=14相同，且WB\_rf\_we为1，有写使能，显然发生了C类数据冒险，而rR1\_c标志正确地变为1。我们看到rf[14]=0xf00ff00f,而WB\_rd=0xff00ff00,发现最后写入ID\_rD1的值是WB\_rd的值，说明数据冒险被正确处理，符合预期。  控制冒险:  从左到右四个周期，分别是bne x4,x5,190的IF、ID、EX、MEM阶段，我的控制冒险检测在EX阶段，所以我们只要着重看第三个时钟周期即可。可以看到，此时EX\_npc\_op=2’b10,  这个值对应的宏定义是C\_NPC\_B,也就是B型指令的npc\_op,而alu\_f=1，说明应该发生跳转，看到npc马上被更新成了pc应该跳去的下一条正确地址。那么接下来再看看后面一个周期发生了什么，pc被更新成了对应的值,我还列出了ID\_pc4和EX\_pc4这两个变量的变化，是为了说明我们的IF/ID、ID/EX两个流水器的信号因为发生控制冒险而正确地被清零了，至此控制冒险全部结束，符合预期。 |

3 设计过程中遇到的问题及解决方法

|  |
| --- |
| 要求：包括设计过程中遇到的有价值的错误，或测试过程中遇到的有价值的问题。所谓有价值，指的是解决该错误或问题后，能够学到新的知识和技巧，或加深对已有知识的理解和运用。 |
| 1.许多vivado相关的问题，多多利用搜索引擎，vivado里很多错误都可以在官网上的论坛上得到解答。  2.写代码的时候接口出现问题，一定、一定、一定把图给画好，我认为对于设计cpu，画好图就成功了70%。 |

4 总结

|  |
| --- |
| 要求：谈谈学完本课程后的个人收获以及对本课程的建议和意见。请在认真总结和思考后填写总结。 |
| 个人收获：我认为收获还是很大的，在上理论课的时候对于流水线抱着一种懵懵懂懂的感觉度过，考试也没有细考，但是在做完流水线后，真正理解了cpu流水线的各种细节，包括解决数据冒险，控制冒险，时序的问题等等，并且也加深了对处理器体系结构的理解，正如Linus Torvalds说的“talk is cheap, show me the code.”，写一遍实验和不写对cpu数据通路的理解差异是巨大的。  建议：多多鼓励学生尽量写完流水线，想不出别的什么建议。。。。  意见：很好的实验，比计组的实验有趣多，希望也能对别的实验改革，希望下学期的操作系统也有趣。 |