武汉大学计算机学院
本科生实验报告

RISC-V 流水线 CPU 设计

专 业 名 称 ：

课 程 名 称 ：

指 导 教 师 ：

学 生 学 号 ：

学 生 姓 名 ：

二〇二一年五月

郑 重 声 明

本人呈交的实验报告，是在指导老师的指导下，独立进行实验工作所取得的成果，所有数据、图片资料真实可靠。尽我所知，除文中已经注明引用的内容外，本实验报告不包含他人享有著作权的内容。对本实验报告做出贡献的其他个人和集体，均已在文中以明确的方式标明。本实验报告的知识产权归属于培养单位。

本人签名：_____ 日期：_____

摘　要

在经过了一学期的计算机系统组成原理理论学习之后，我们进行了 RISC-V 流水线 CPU 设计实验。通过本次实验，融会贯通理论知识于实践之中，加深了对 CPU 系统各模块的工作原理及相互联系的认识；学习了采用 EDA 技术设计 CPU 的方法技术；培养了独立的科研能力和 CPU 的综合开发能力；最后，我们了解了 SOC 系统，并在 FPGA 开发板上完成了简单的实现。

本次实验的内容主要包括：用硬件描述语言（Verilog）设计 RISC-V 流水线 CPU，支持 RV32I 中的大部分指令；用仿真软件 Modelsim 对有数据冒险和控制冒险的汇编程序进行仿真验证指令实现质量；确认指令实现无误后使用 Vivado 使用下载到 N4 FPGA 板上进行综合、实现，并运行学号排序案例和斐波那契数列，查看并记录结果。

经过一次次失败与尝试，我独立地完成了流水线 CPU 的设计，实现的指令代码在仿真软件 Modelsim 上完成了对有数据冒险和控制冒险的汇编程序的仿真。最后，在修改了部分代码后，实现的指令通过 Vivado 下载到 FPGA 板上，顺利地完成了学号排序和斐波那契数列的生成。

通过本次实验，我充分地感受到了理论应用于实践上的重要性，充分地感受到了自己从理论者逐步过渡到实践者的进步与满足。相信在未来的学习中，我能更好地将所学的理论联系到实际中，做到知行合一。

关键词：RISC-V，CPU 流水线，Modelsim，Vivado，FPGA

目 录

1 实验目的和意义

1.1 实验目的

在学习了计算机组成原理理论知识之后，开展本次实验设计流水线 CPU，本次实验的主要目的为：

1. 融会贯通计算机组成与设计课程所教授的知识，通过对知识的综合应用，加深对 CPU 系统各模块的工作原理及相互联系的认识；
2. 学习采用 EDA（Electronic Design Automation）技术设计 RISC-V 流水线 CPU 的技术与方法；
3. 培养科学研究的独立工作能力，获得 CPU 设计、仿真、综合、实现、运行的实践和经验；
4. 了解 SOC 系统，并在 FPGA 开发板上实现简单的 SOC 系统。

1.2 实验意义

本次实验的重要意义如下：

1. 学生学习了一学期的计算机组成原理知识之后，对 CPU 的设计有了一定程度上理论的理解，但还是觉得真实的 CPU 离我们很遥远。通过本次实验，不仅能够将学生一学期所学的理论知识进行整合，也能通过实践，让学生亲身体验实际开发 CPU 模型的成就感。
2. 通过本次实验，学生能对编程语言和各类仿真软件有着更清晰的认识，提高了学生的实际 CPU 模型动手开发能力。在实验过程中，学生们需要回顾数字电路课程中介绍的硬件描述语言（Verilog）；需要掌握仿真软件（Modelsim 和 Vivado）的使用与运行方法；需要懂得 FPGA 开发板的实际操作。这一系列的要求帮助我们提升了综合的工程开发能力。
3. 通过本次能力，学生能养成一定的独立科研能力。从最初的一头雾水，到不断尝试排错之后的进步，再到最后的顺利完成流水线 CPU 的研发。学生的心理承受能力不断增强，对理论知识的掌握也更加科学，

2 实验环境介绍

2.1 Verilog HDL

Verilog HDL 是一种硬件描述语言，用于从算法级、门级到开关级的多种抽象设计层次的数字系统建模。被建模的数字系统对象的复杂性可以介于简单的门和完整的电子数字系统之间。数字系统能够按层次描述，并可在相同描述中显式地进行时序建模。用它可以表示逻辑电路图、逻辑表达式，还可以表示数字逻辑系统所完成的逻辑功能。

Verilog HDL 语言具有下述描述能力：设计的行为特性、设计的数据流特性、设计的结构组成以及包含响应监控和设计验证方面的时延和波形产生机制。所有这些都使用同一种建模语言。此外，Verilog HDL 语言提供了编程语言接口，通过该接口可以在模拟、验证期间从设计外部访问设计，包括模拟的具体控制和运行。

2.2 Venus

Venus 是一款面向教育应用的 RISC-V 指令仿真开源平台。用户可以将写好的汇编代码在此平台上运行，生成对应的机器码，且可以在平台上实时仿真运行生成的机器代码，观察机器内部数据（寄存器、存储器等）的变化情况。

2.3 ModelSim

Mentor 公司的 ModelSim 是业界最优秀的 HDL 语言仿真软件，它能提供友好的仿真环境，是业界唯一的单内核支持 VHDL 和 Verilog 混合仿真的仿真器。它采用直接优化的编译技术、Tcl/Tk 技术、和单一内核仿真技术，编译仿真速度快，编译的代码与平台无关，便于保护 IP 核，个性化的图形界面和用户接口，为用户加快调错提供强有力的手段，是 FPGA/ASIC 设计的首选仿真软件。

2.4 Vivado

Vivado 设计套件，是 FPGA 厂商赛灵思公司 2012 年发布的集成设计环境。包括高度集成的设计环境和新一代从系统到 IC 级的工具，这些均建立在共享的可扩展数据模型和通用调试环境基础上。这也是一个基于 AMBA AXI4 互联规范、IP-XACT IP 封装元数据、工具命令语言(TCL)、Synopsys 系统约束(SDC) 以及其它有助于根据客户需求量身定制设计流程并符合业界标准的开放式环境。赛灵

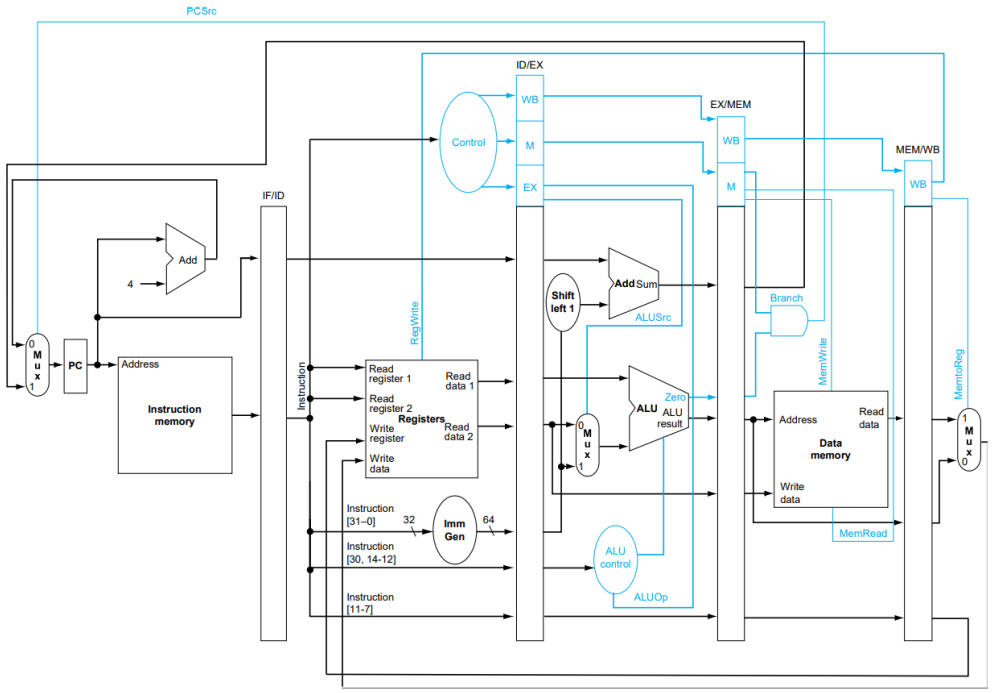
思构建的的 Vivado 工具把各类可编程技术结合在一起，能够扩展多达 1 亿个等效 ASIC 门的设计。

2.5 Nexys 4DDR

Nexys 4 DDR 是一款 Digilent 多孔 RAM-based Nexys 开发板的简易替代品。是一个打开即用型的数字电路开发平台，帮助使用者能够在课堂环境下实现诸多工业领域的应用。Nexys 4 DDR 开发板能实现从理论型组合电路到强大的嵌入式处理器的多种设计。Digilent 将提供一个硬件描述语言（VHDL）参考模块，以封装 DDR2 控制器的复杂性，并且向下兼容 CellularRAM 的异步 SRAM 接口。它兼容 Xilinx 最新的高性能 Vivado®设计套件以及 ISE®工具包，其中包含了 ChipScope™和 EDK。Xilinx 提供免费版本 WebPACK™工具包，帮助用户在无需支付额外费用的情况下就可以轻松实现设计。

3 概要设计

3.1 总体设计



如图，类似于计算机组成原理中的流水线 CPU 架构图，我设计的 CPU 也是**流水线**形式的。同样地，**流水线也分为 5 个阶段**，即包括 IF（取指令阶段），ID（指令译码阶段），EX（运算执行阶段），MEM（存储器操作阶段）和 WB（写回阶段）。下面依次介绍各个阶段的大致功能：

IF：依据 PC（程序计数器）中记录的地址信息，从指令存储器中取出指令，并从 PC+4（顺序执行）和 ID 阶段返回的 PCBranch（跳转执行）进行选择，更新 PC，为下一个指令的取出做好准备。

ID：一，对 IF 阶段取出的指令进行译码，得到后续操作的一系列信息（访问寄存器的编号，是否写回寄存器，写回寄存器的编号，是否要存/取数据，指令中的常数段等等）。二，访问寄存器，获得相应的数据，为后续的计算提供基础。三，与教材中不同的是，我们实现的流水线 CPU 跳转地址的确认不是在 EX 阶段，而是在 ID 阶段。在 ID 阶段，我们将计算出跳转的地址，并送回 IF 区域进行 PC 值的更新。

EX：依据控制器发出的控制信息，选择参与 ALU 运算的数据，设定 ALU

具体的运算方式，完成运算，得到结果。

MEM：依据控制信息，实现数据从数据存储器的存储或者取出。

WB：依据控制信息，将有需要写回寄存器的数据写回寄存器。

在指令实现方面，我们实现了 RISC-V32 中的大部分指令，包括：

- I0={LUI, AUIPC}
- I1={{LB, LH, LW, LBU, LHU, SB, SH, SW }
- I2={ ADD, SUB, SLL, SLT, SLTU, XOR, SRL, SRA, OR, AND }
- I3={ADDI, SLTI, SLTIU, XORI, ANDI, ALLI, SRLI, SRALI, SRAI}
- I4={ JAL, JALR, BEQ, BNE, BLT, BGE, BLTU, BGEU }

在流水线设计方面，我的流水线设计较好地规避了以下 3 个问题：

1. 有条件与无条件分支指令后误读的指令是否能够正确清空；
2. 能否正确处理前递与转发：MEM-->EX, WB-->EX, WB-->MEM, MEM-->ID；
3. 能否正确处理需要停顿的数据依赖: load-use, arith-beq, load-beq, arith-jalr, load-jalr。

3.2 PC（程序计数器）

3.2.1 功能描述

程序计数器，其存储的数据即代执行指令的地址，在 IF 阶段被取出，并用于从指令存储器中加载要运行的指令。

3.3 RF（寄存器文件）

3.3.1 功能描述

寄存器是 CPU 内部用来存放数据的一些小型存储区域，用来暂时存放参与运算的数据和运算结果。

3.3.2 模块接口

信号名	方向	描述
Clk	input	时钟信号
Ra1	Input	访问寄存器 1 的编号
Ra2	input	访问寄存器 2 的编号

Rd1	output	访问寄存器 1 的数据
Rd2	Output	访问寄存器 2 的数据
We3	input	写入信号
Wa3	input	写入的寄存器编号
Wd3	input	写入寄存器的数据

3.4 IMEM（指令存储器）

3.4.1 功能描述

指令存储器用于存储指令。IF 阶段通过 PC 指定的地址，从 IMEM 中取出对应的指令。

3.4.2 模块接口

信号名	方向	描述
a	input	待取指令的地址
instr	output	取出的命令机器代码

3.5 DMEM（数据存储器）

3.5.1 功能描述

数据存储器用于存储数据。

3.5.2 模块接口

信号名	方向	描述
Clk	input	时钟信号
Amp	Input	访存模式
Rd	output	读出的数据
We	input	写入信号
a	input	写入的存储器的地址
Wd	input	写入存储器的数据

3.6 ALU（运算器）

3.6.1 功能描述

EX 阶段对输入的两个操作数进行运算的元件。

3.6.2 模块接口

信号名	方向	描述
a	input	操作数 1
b	Input	操作数 2
shamt	input	Shamt
aluctrl	input	控制信号
aluout	output	输出
overflow	output	溢出标志
zero	Output	相等标志
lt	Output	a<b
ge	output	a>=b

3.7 Controller（控制器）

3.7.1 功能描述

控制信号产生处，控制其它元器件的活动。

3.7.2 模块接口（列出主要部分）

信号名	方向	描述
immctrl	output	立即数生成的控制信号
itype	output	是否为 I 型指令
jal	output	指令为 jal 或者 jalr
jalr	output	指令为 jalr
aluctrl	output	ALU 运算控制信号
alusrc	output	ALU 操作数的选取方式
Memwrite	Output	是否要写存储器
memtoreg	Output	是否要从数据存储器取数据
Regwrite	output	是否要写回寄存器
Pcsrc	output	PC 选择控制信号（是否跳转）

3.8 CMP（比较器）

3.6.1 功能描述

ID 对两个寄存器输出的值进行比较，为是否跳转的确定打基础。

3.6.2 模块接口

信号名	方向	描述
a	input	操作数 1
b	Input	操作数 2
op_unsigned	input	是否为无符号比较

zero	Output	相等标志
lt	Output	a<b

3.9 流水线转发机制

3.9.1 功能描述

正确处理前递与转发：MEM-->EX, WB-->EX, WB-->MEM, MEM-->ID

具体实现过程在详细设计中由代码体现。

3.10 冲突检测机制

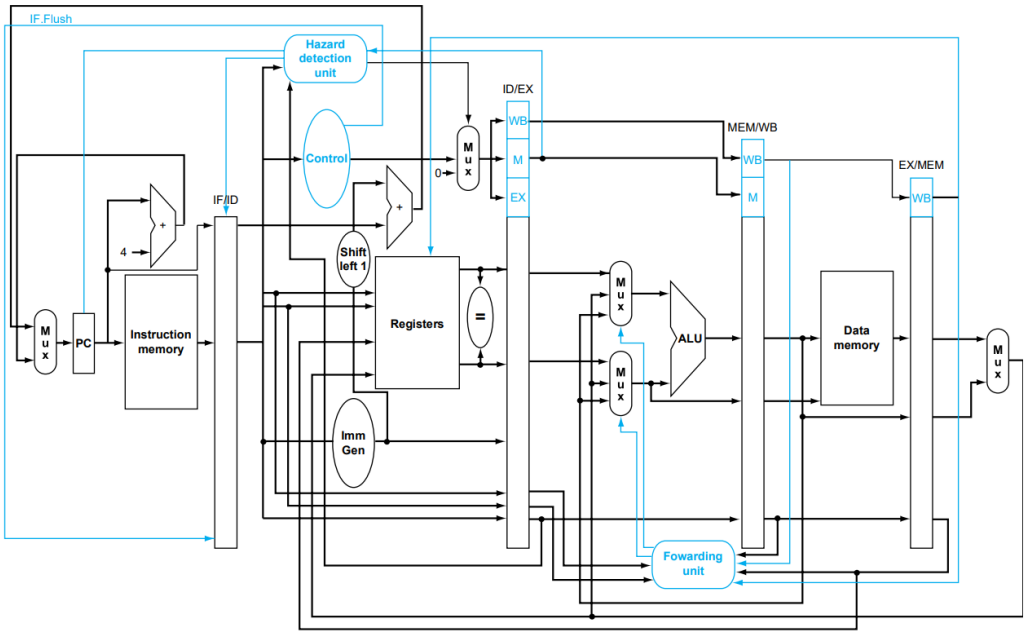
3.10.1 功能描述

正确处理需要停顿的数据依赖：load-use, arith-beq, load-beq, arith-jalr, load-jalr。

具体实现过程在详细设计中由代码体现。

4　详细设计

4.1 CPU 总体结构



4.2 PC（程序计数器）

```
wire [`ADDR_SIZE-1:0]    pcplus4F, nextpcF, pcbranchD, pcadder2aD, pcadder2bD, pcbranch0D, pcadder2bD1;
mux2 #(`ADDR_SIZE)  pcsrcmux(pcplus4F, pcbranchD, pcsrcD, nextpcF); //choose from pc+4 and branch, now pc

// Fetch stage logic
pcenr pcreg(clk, reset,en, nextpcF, pcF); //choose pc from nextpc and initial, pcF is output to IMeM
```

由 controller 发出的控制信号 pcsrcD 进行控制，PC 从顺序执行（pcplus4F）和跳转执行（pcbranchD）进行选择，更新 PC 值，确定下一个执行指令。

4.3 RF（寄存器文件）

```
module regfile(
  //???????
  input                       clk,
  input  [`RFIDX_WIDTH-1:0]   ra1, ra2, //?????
  output [`XLEN-1:0]          rd1, rd2, //????????

  input [4:0] reg_sel,
  output[31:0] reg_data,

  //???????
  input                       we3, //???
  input  [`RFIDX_WIDTH-1:0]   wa3, //????????
  input  [`XLEN-1:0]          wd3 //?????
  );
```

```
  reg [`XLEN-1:0] rf[`RFREG_NUM-1:0];

  always @(negedge clk)

    //????

    if (we3 && wa3!=0)
      begin
          rf[wa3] <= wd3;
          `ifdef DEBUG
            $display("x%d = %h", wa3, wd3);
          `endif
      end

  //?
  assign rd1 = (ra1 != 0) ? rf[ra1] : 0;
  assign rd2 = (ra2 != 0) ? rf[ra2] : 0;
endmodule
```

对于读取数据方面，若有真实的读取字段，则输出对应的寄存器数据；否则为 0。对于写数据方面，若 we 和 wa 均存在且有效，则将 wd 写入对应的寄存器编号。

4.4 IMEM（指令存储器）

```
module imem(input  [`ADDR_SIZE-1:0]   a,
                   output [`INSTR_SIZE-1:0]  rd); //imem(pc, rd)

  reg  [`INSTR_SIZE-1:0] RAM[`IMEM_SIZE-1:0]; ////1024?32??RAM

  initial //Get instr from sim file
    begin
      $readmemh("fib.dat", RAM); //???RAM
    end

  assign rd = RAM[a[11:2]]; // get instruction from RAM
endmodule
```

首先使用 readmemh 读入事先实现的命令集（dat 文件）进行初始化。在指令执行的过程中，由输入的 PC 值来取出相应的指令。

4.5 DMEM（指令存储器）

```
module dmem(input               clk, we, //we:if write
            input  [3:0]        amp,  //
            input  [`XLEN-1:0]  a, wd, //address, data-to-write
            output [`XLEN-1:0]  rd);  //read data

  reg  [31:0] RAM[1023:0]; //1024 RAM

  assign rd = RAM[a[11:2]]; //Get data from RAM

  always @(posedge clk)
    if (we)
      begin
      case (amp)
        4'b1111: RAM[a[11:2]] <= wd;              // sw
        4'b0011: RAM[a[11:2]][15:0] <= wd[15:0];  // sh
        4'b1100: RAM[a[11:2]][31:16] <= wd[15:0]; // sh
        4'b0001: RAM[a[11:2]][7:0] <= wd[7:0];    // sb
         4'b0010: RAM[a[11:2]][15:8] <= wd[7:0];   // sb
        4'b0100: RAM[a[11:2]][23:16] <= wd[7:0];  // sb
        4'b1000: RAM[a[11:2]][31:24] <= wd[7:0];  // sb
        default: RAM[a[11:2]] <= wd;// it shouldn't happen
       endcase
      $display("dataaddr = %h, writedata = %h", {a[31:2],2'b00}, wd);
      //$display("dataaddr = %h, memdata = %h", {a[31:2],2'b00}, RAM[a[11:2]]);
       end
endmodule
```

当从数据存储器取数据时，由输入的地址 a 确定我们将要输出的 rd；在往数据存储器写数据时，通过 a 地址，我们可以知道写数据的位置，通过 amp，我们可以知道是存储多大的字段。

4.6 ALU（运算器）

```
`include "xgriscv_defines.v"
module alu(
    input   [`XLEN-1:0]         a, b,
    input   [4:0]   shamt,
    input   [3:0]   aluctrl,

    output reg [`XLEN-1:0]      aluout,
    output      overflow,
    output                      zero,
    output                      lt,
    output                      ge
    );
```

ALU 端口的定义

```
always@(*)
    case(aluctrl[3:0])
    `ALU_CTRL_MOVEA:    aluout <= a;
    `ALU_CTRL_ADD:      aluout <= sum[`XLEN-1:0];
    `ALU_CTRL_ADDU:     aluout <= sum[`XLEN-1:0];


    `ALU_CTRL_OR:       aluout <= a | b;
    `ALU_CTRL_XOR:      aluout <= a ^ b;
    `ALU_CTRL_AND:      aluout <= a & b;
```

ALU 内通过接受输入的控制信号 aluctrl，进行特定的运算。

```
assign overflow = sum[`XLEN-1] ^ sum[`XLEN];
//assign overflow = (a[31] & b2[31] & ~sum[31]) | (~a[31] & ~b2[31] & sum[31]);
//assign overflow = (~sum[`XLEN-1] & sum[`XLEN]) | (sum[`XLEN-1] & ~sum[`XLEN]);
assign zero = (aluout == `XLEN'b0);
assign lt = aluout[`XLEN-1];
assign ge = ~aluout[`XLEN-1];
```

完成各种比较结果的计算和输出。

4.7 Controller（控制器）

```
module controller(
  input                     clk, reset,
  input [6:0]               opcode,
  input [2:0]               funct3,
  input [6:0]               funct7,
  input [`RFIDX_WIDTH-1:0]  rd, rs1,
  input [11:0]              imm,
  input                     zero, lt, // from cmp in the decode stag

// These control info flow along the pipline
  output [4:0]              immctrl,          // for the ID stage
  output                    itype, jal, jalr, bunsigned, pcsrc, //Af

  output reg  [3:0]         aluctrl,          // for the EX stage
  output [1:0]              alusrca,
  output                    alusrcb,

  output                    memwrite, lunsigned,//if load unsigned
  output [1:0]              lwhb, swhb,

  output                    memtoreg, regwrite  // for the WB stage,
  );
```

Controller 模块的接口定义。

```
// if exist this instruction
  wire rv32_lui    = (opcode == `OP_LUI);
  wire rv32_auipc  = (opcode == `OP_AUIPC);
  wire rv32_jal    = (opcode == `OP_JAL);
  wire rv32_jalr   = (opcode == `OP_JALR);
  wire rv32_branch= (opcode == `OP_BRANCH);
  wire rv32_load   = (opcode == `OP_LOAD);
  wire rv32_store  = (opcode == `OP_STORE);
  wire rv32_addri  = (opcode == `OP_ADDI);
  wire rv32_addrr  = (opcode == `OP_ADD);
```

各种指令是否存在的判定，为后续各个具体判断信号的生成打下基础；

```
  assign itype = rv32_addri | rv32_load | rv32_jalr;

  // S-type
  wire stype = rv32_store;

  // B-type imm????????????branch??????????????x2??????
  wire btype = rv32_branch;

  // U-type imm??20???12??0?LUI?AUIPC?LUI??ALUm MOVA
  wire utype = rv32_lui | rv32_auipc;

  // J-type imm????????????JAL???????????x2????1???????
  wire jtype = rv32_jal;

  assign immctrl = {itype, stype, btype, utype, jtype};
```

立即数生成部分的控制信号。需要生成立即数的指令由 I、S、U、J 型指令。要对各种情况做出逻辑的判断，最后生成立即数生成的控制信号；

```
assign pcsrc = rv32_jal | rv32_jalr | (rv32_beq & zero) | (rv32_bge & (! lt)) | (rv32_bgeu & (! lt)) | (rv32_blt & lt
```

对 pcsrc 的定义（是一个修改的重点），即跳转指令发生的情况，包括 jal，jalr 和 B 型指令确切发生的情况；

```
assign memtoreg = rv32_load;

assign regwrite = rv32_lui | rv32_auipc | rv32_addri | rv32_load | rv32_addrr | rv32_jal | rv32_jalr;
```

对 memtoreg 和 regwrite 做出逻辑判定。

```
always @(*)
  case(opcode)
    `OP_LUI:     aluctrl <= `ALU_CTRL_ADD;
    `OP_AUIPC:    aluctrl <= `ALU_CTRL_ADD;

    `OP_ADDI:    case(funct3)
                   `FUNCT3_ADDI: aluctrl <= `ALU_CTRL_ADD;
                   `FUNCT3_SLTI: aluctrl <= `ALU_CTRL_SLT;
                   `FUNCT3_SLTIU:aluctrl <= `ALU_CTRL_SLTU;
                   `FUNCT3_XORI: aluctrl <= `ALU_CTRL_XOR;
                    `FUNCT3_ORI: aluctrl <= `ALU_CTRL_OR;
                    `FUNCT3_ANDI:  aluctrl <= `ALU_CTRL_AND;
                   `FUNCT3_SLL: aluctrl <= `ALU_CTRL_SLL;
                   `FUNCT3_SR:   case (funct7)
                                   `FUNCT7_SRLI:  aluctrl <= `ALU_CTRL_SRL;
                                   `FUNCT7_SRAI:  aluctrl <= `ALU_CTRL_SRA;
                                   default:   aluctrl <= `ALU_CTRL_ZERO;
                                 endcase
                   default:    aluctrl <= `ALU_CTRL_ZERO;
                 endcase
```

对各个指令对应的 ALU 运算控制（如果有的话）进行设置。通过 opcode、func3 和 func7（如果有需要的话）对各个指令进行分类，从而给每个指令对应的运算进行设置。

4.8 CMP（比较器）

```
module cmp(
  input [`XLEN-1:0] a, b,
  input             op_unsigned,
  output            zero,
  output            lt);

  assign zero = (a == b);
  assign lt = (!op_unsigned & ($signed(a) < $signed(b))) | (op_unsigned & (a < b));
endmodule
```

对输入的待比较的操作数进行运算，输出 a==b 和 a<b 两个布尔信号。

4.9 流水线转发机制

正确处理前递与转发：MEM-->EX, WB-->EX, WB-->MEM, MEM-->ID

MEM→EX 和 WB→EX:

```
//S7_2, MEM-->EX, WB-->EX
wire aM = (regwriteM && rdM != 5'b0 && rdM == rs1E);
wire aW = (regwriteW && waddrW != 5'b0 && waddrW == rs1E && !(regwriteM && rdM != 5'b0 && rdM == rs1E));
wire bM = (regwriteM && rdM != 5'b0 && rdM == rs2E);
wire bW = (regwriteW && waddrW != 5'b0 && waddrW == rs2E && !(regwriteM && rdM != 5'b0 && rdM == rs2E));
mux3 #(2) forwardaEmux(2'b00, 2'b01, 2'b10, {aM, aW}, forwardaE);
mux3 #(2) forwardbEmux(2'b00, 2'b01, 2'b10, {bM, bW}, forwardbE);
```

当 regwrite，rdM 存在且有效的时候，若 rdM（目标寄存器编号）和 rs1E（即

将参加运算的寄存器编号）或 rs2E 相同，比如上一条指令的运算输出是本次指令的运算操作数，则要发生 MEM→EX 的转发；当前者没发生时，当 regwrite，waddrw（写回的寄存器编号）存在且有效的时候，若 waddrW 和 rs1E 或 rs2E 相同，比如上一条 ld 指令读出的数据是本条指令的运算操作数，则要发生 WB→EX 的转发；

WB-->MEM：

```
//s7_2 WB-->MEM
wire forwardM = ((waddrW != 0) && (rs2M == waddrW) && regwriteW);
mux2 #(`XLEN) memOrwbmux(writedataM1, wdataW, forwardM, writedataM);
```

当 regwrite，waddrw（写回的寄存器编号）存在且有效的时候，若 waddrw 和 rs2M 相等，比如上一条 ld 指令读取的数据是本条 store 指令要存储的数据，则要发生 WB→MEM 的转发；

MEM→ID：

```
//MEM->ID
wire regwriteM;
wire [`RFIDX_WIDTH-1:0] rdM;
wire forwardaD = (regwriteM && rdM != 5'b0 && rdM == rs1D);
wire forwardbD = (regwriteM && rdM != 5'b0 && rdM == rs2D);
mux2 #(`XLEN)  rdata1Dmux(rdata1D, aluoutM, forwardaD, rdata1D1);
mux2 #(`XLEN)  rdata2Dmux(rdata2D, aluoutM, forwardbD, rdata2D1);

addr_adder pcadder3(rdata1D1, immoutD, pcadder2bD1); //jalr_1
set_last_zero set_zero(pcadder2bD1, pcadder2bD); //jalr_2

mux2 #(`XLEN) pcsrcmux2(pcadder2aD, pcadder2bD, jalrD, pcbranchD);

cmp cmp(rdata1D1, rdata2D1, bunsignedD, zeroD, ltD);// compare r1data with r2data
```

当 regwrite，rdM 存在且有效的时候，若 rdM（目标寄存器编号）和 rs1D（即将参加比较的寄存器编号）或 rsDE 相同，比如上一条指令的运算结果是本条（跳转）指令待比较的对象，则要发生 MEM→ID 的转发。

4.10 冲突检测机制

正确处理需要停顿的数据依赖：load-use，arith-beq，load-beq，arith-jalr，load-jalr。

```
// hazard detection
wire memtoregE;
wire [`RFIDX_WIDTH-1:0] rdE;

assign hazard = (memtoregD & rdD != 5'b0 & (
    (opF == `OP_JALR) & (rdD == rs1F) |
    (opF == `OP_LOAD) & (rdD == rs1F) |
    (opF == `OP_ADDI) & (rdD == rs1F) |
    (opF == `OP_ADD) & ((rdD == rs1F) | (rdD == rs2F)) |
    (opF == `OP_BRANCH) & ((rdD == rs1F) | (rdD == rs2F)) |
    (opF == `OP_STORE) & ((rdD == rs1F) | (rdD == rs2F)))
)|
(regwriteD & rdD != 5'b0 & (
    (opF == `OP_JALR) & (rdD == rs1F) |
    (opF == `OP_BRANCH) & ((rdD == rs1F) | (rdD == rs2F))
) )|
(memtoregE & rdE != 5'b0 & (
    (opF == `OP_JALR) & (rdE == rs1F) |
    (opF == `OP_BRANCH) & ((rdE == rs1F) | (rdE == rs2F))
) );
```

```
// Fetch stage logic
pcenr pcreg(clk, reset,en, nextpcF, pcF); //choose pc from nextpc and
wire hazard;
assign en = !hazard;// pcF = nextpcF, when hazerd, this need to change
addr_adder  pcadder1 (pcF, `ADDR_SIZE'b100, pcplus4F); //pc = pc + 4
```

我设计的冲突检测单元在 IF 阶段实现冲突指令的检测，若检测为冲突指令，则将 en 设置为 0，制止了 PC 的更新，从而实现了流水线的停顿。从 hazard 的设置可以看出，load-use，arith-beq 和 arith-jalr 会导致一次流水线停顿，而 load-beq，load-jalr 会导致两次流水线的停顿。

4.11 指令实现

4.11.1 S 型指令的实现

```
// next PC logic (operates in fetch and decode)
wire [`ADDR_SIZE-1:0]    pcplus4F, nextpcF, pcbranchD, pcadder2aD, pcadder2bD, pcbranch0D, pcadder2bD1;
mux2 #(`ADDR_SIZE)  pcsrcmux(pcplus4F, pcbranchD, pcsrcD, nextpcF); //choose from pc+4 and branch, now pcsrcD = 0,
```

IF 阶段正常取指令，pcsrcD = 0，NextPC = PC + 4；

ID 阶段传递 PC 和 PC+4（便于进行选择和跳转操作的选择），指令译码，生成 S-type 的立即数。rdata1D 为地址的基址，rdata2D 为待存储的数据，将直接传到 MEM。参与 ALU 运算的被选中的是 S-type 的立即数；

```
mux3 #(`XLEN)  srca1mux(srca1E, wdataW, aluoutM, forwardaE, srca2E);// srca1mux
mux3 #(`XLEN)  srca2mux(srca2E, 0, pcE, alusrcaE, srca3E);          // srca2mux
mux3 #(`XLEN)  srcb1mux(srcb1E, wdataW, aluoutM, forwardbE, srcb2E);// srcb1mux
mux2 #(`XLEN)  srcb2mux(srcb2E, immoutE, alusrcbE, srcb3E);         // srcb2mux

alu alu(srca3E, srcb3E, shamtE, aluctrlE, aluoutE, overflowE, zeroE, ltE, geE);
```

```
assign alusrca = rv32_lui ? 2'b01 : (rv32_auipc ? 2'b10 : 2'b00);

assign alusrcb = rv32_lui | rv32_auipc | rv32_addri | rv32_store | rv32_load;
```

```
`OP_STORE: case (funct3)
  `FUNCT3_SB: aluctrl <= `ALU_CTRL_ADD;
  `FUNCT3_SH: aluctrl <= `ALU_CTRL_ADD;
  `FUNCT3_SW: aluctrl <= `ALU_CTRL_ADD;
  default: aluctrl <= `ALU_CTRL_ZERO;
endcase
default:  aluctrl <= `ALU_CTRL_ZERO;
```

EX 阶段，假设无数据冒险与前递，srcamux 选 rdata1D，alusrcaE=0。srcbmux 选出 immoutE， alusrcbE = 1。aluctrlE 也在 controller 中进行实现，为加法（基址加偏移地址）；

MEM 阶段，将 rdata2D 传入作为待写的数据。进行存储（已经实现）。形式是得到 writedataM,为输送到 mem 的 output。

没有 WB 阶段。

4.11.2 ld 指令的实现

IF 阶段正常取指令，且 NextPC = PC + 4（pcsrcD = 0）；

```
// I-S-B-U-J needs IMM Gen
  // I-type
  assign itype = rv32_addri | rv32_load | rv32_jalr;
```

ID 阶段传递 PC 和 PC+4（便于进行选择和跳转操作的选择），指令译码，生成 I-type 的立即数（controller 中进行设置）。rdata1D 为地址的基址，rd 为将要写入的寄存器地址。参与 ALU 运算的被选中的是 I-type 的立即数；

```
assign alusrca = rv32_lui ? 2'b01 : (rv32_auipc ? 2'b10 : 2'b00);

assign alusrcb = rv32_lui | rv32_auipc | rv32_addri | rv32_store | rv32_load;
```

```
`OP_LOAD: case (funct3)
  `FUNCT3_LB: aluctrl <= `ALU_CTRL_ADD;
  `FUNCT3_LH: aluctrl <= `ALU_CTRL_ADD;
  `FUNCT3_LW: aluctrl <= `ALU_CTRL_ADD;
  `FUNCT3_LBU: aluctrl <= `ALU_CTRL_ADDU;
  `FUNCT3_LWU: aluctrl <= `ALU_CTRL_ADDU;
  `FUNCT3_LHU: aluctrl <= `ALU_CTRL_ADDU;
  default: aluctrl <= `ALU_CTRL_ZERO;
endcase
```

EX 阶段，srcamux 选 rdata1D，alusrcaE = 0。srcbmux 选出 immoutE， alusrcaE = 1。aluctrlE 也要在 controller 中实现，为 ADD 或者 ADDU（基址加偏移地址）；

MEM 阶段，取将要 load 的数据，记为 memdataM。

```
// write-back stage logic
mux2 #(`XLEN) wbmux1(aluoutW, memdataW, memtoregW, wdataW1);
mux2 #(`XLEN) wbmux2(wdataW1, pcplus4W, jalW, wdataW);
```

WB 阶段，rdM 传为 waddrW，将 memdataW 写回 reg（与 aluoutW 进行比较挑选，memtoregW = 1）。

4.11.3 R 型运算指令的实现

IF 阶段正常取指令，且 NextPC = PC + 4（pcsrcD = 0）

ID 阶段传递 PC 和 PC+4（便于进行选择和跳转操作的选择），指令译码。rdata1D 为数据一，，rdata2D 为数据二，rd 为将要写入的寄存器地址。参与 ALU 运算的被选中的是 rdata1 和 rdata2。

EX 阶段，srcamux 选 rdata1D，alusrcaE = 0。srcbmux 选出 rdata2D， alusrcaE = 0。aluctrlE 也要在 controller 实现。

```
`OP_ADD: case (funct3)

  `FUNCT3_ADD: case (funct7)
    `FUNCT7_ADD: aluctrl <= `ALU_CTRL_ADD;
    `FUNCT7_SUB: aluctrl <= `ALU_CTRL_SUB;
    default: aluctrl <= `ALU_CTRL_ZERO;
  endcase

  `FUNCT3_XOR: aluctrl <= `ALU_CTRL_XOR;
  `FUNCT3_OR: aluctrl <= `ALU_CTRL_OR;
  `FUNCT3_AND: aluctrl <= `ALU_CTRL_AND;

  `FUNCT3_SR: case (funct7)
    `FUNCT7_SRL: aluctrl <= `ALU_CTRL_SRL;
    `FUNCT7_SRA: aluctrl <= `ALU_CTRL_SRA;
    default: aluctrl <= `ALU_CTRL_ZERO;
  endcase

  `FUNCT3_SLL: aluctrl <= `ALU_CTRL_SLL;
  `FUNCT3_SLT: aluctrl <= `ALU_CTRL_SLT;
  `FUNCT3_SLTU: aluctrl <= `ALU_CTRL_SLTU;

  default: aluctrl <= `ALU_CTRL_ZERO;
```

MEM 阶段，无事可做。

```
// write-back stage logic
mux2 #(`XLEN) wbmux1(aluoutW, memdataW, memtoregW, wdataW1);
mux2 #(`XLEN) wbmux2(wdataW1, pcplus4W, jalW, wdataW);
```

WB 阶段，rdM 传为 waddrW，将 aluoutW 作为 wdataW 写回 reg（memtoregW = 0）。

4.11.4 I 型运算指令的实现

IF 阶段正常取指令，且 NextPC = PC + 4（pcsrcD = 0）；

```
// I-S-B-U-J needs IMM Gen
  // I-type
  assign itype = rv32_addri | rv32_load | rv32_jalr;
```

ID 阶段传递 PC 和 PC+4（便于进行选择和跳转操作的选择），指令译码。生成 I-type 的立即数（controller 中进行设置）。rdata1D 为数据一，rd 为将要写入的寄存器地址。参与 ALU 运算的被选中的是 I-type 的立即数；

```
`OP_ADDI:   case(funct3)
              `FUNCT3_ADDI: aluctrl <= `ALU_CTRL_ADD;
              `FUNCT3_SLTI: aluctrl <= `ALU_CTRL_SLT;
              `FUNCT3_SLTIU:aluctrl <= `ALU_CTRL_SLTU;
              `FUNCT3_XORI: aluctrl <= `ALU_CTRL_XOR;
               `FUNCT3_ORI: aluctrl <= `ALU_CTRL_OR;
               `FUNCT3_ANDI:  aluctrl <= `ALU_CTRL_AND;
              `FUNCT3_SLL: aluctrl <= `ALU_CTRL_SLL;
              `FUNCT3_SR:   case (funct7)
                                  `FUNCT7_SRLI:  aluctrl <= `ALU_CTRL_SRL;
                                  `FUNCT7_SRAI:  aluctrl <= `ALU_CTRL_SRA;
                                  default:   aluctrl <= `ALU_CTRL_ZERO;
                            endcase
              default:    aluctrl <= `ALU_CTRL_ZERO;
            endcase
```

EX 阶段，srcamux 选 rdata1D, alusrcaE = 0。srcbmux 选出 immout， alusrcaE = 1。aluctrlE 也要实现；

MEM 阶段，无事可做；

WB 阶段，rdM 传为 waddrW，将 aluoutW 作为 wdataW 写回 reg（memtoregW = 0）。

4.11.5 jar,jalr

jal：

```
//assign pcsrc = 0; //pcsrc = 0 means pc + 4, need to change
//S5
assign pcsrc = rv32_jal | rv32_jalr | (rv32_beq & zero) | (rv32_bge & (! lt)) | (rv32_bgeu & (! lt)) |
          (rv32_blt & lt) | (rv32_bltu & lt) | (rv32_bne & (! zero));
```

跳转指令引入后，在 IF 阶段将要进行下一个 pc 取值的判定（不考虑流水线的话），即 pc+4 还是 pcbranch。这是通过 pcsrcD 的设置来实现的。之前的 pcsrcD 一直默认为 0，此时需要在 controller 中进行修改设置。即 jal 一定会导致 pcsrcD = 1。

```
// J-type imm?????????????JAL??????????x2????1???????
wire jtype = rv32_jal;
```

```
sl1 immGen(immoutD, immoutD_sl1);
addr_adder pcadder2(pcD, immoutD_sl1, pcadder2aD); //jal or b-type
```

ID 阶段传递 PC 和 PC+4 便于进行选择和跳转操作的选择），指令译码。生成 J-type 的立即数（controller 中进行设置）。将立即数左移一位后，与 PC 相加，即

为跳转后的结果，记为 pcadder2aD。

```
// write-back stage logic
mux2 #(`XLEN) wbmux1(aluoutW, memdataW, memtoregW, wdataW1);
mux2 #(`XLEN) wbmux2(wdataW1, pcplus4W, jalW, wdataW);
```

WB 阶段返回地址 PC+4 被写回 rd

jalr：

```
//assign pcsrc = 0; //pcsrc = 0 means pc + 4, need to change
//S5
assign pcsrc = rv32_jal | rv32_jalr | (rv32_beq & zero) | (rv32_bge & (! lt)) | (rv32_bgeu & (! lt)) |
               (rv32_blt & lt) | (rv32_bltu & lt) | (rv32_bne & (! zero));
```

跳转指令引入后，在 IF 阶段将要进行下一个 pc 取值的判定（不考虑流水线的话），即 pc+4 还是 pcbranch。这是通过 pcsrcD 的设置来实现的。之前的 pcsrcD 一直默认为 0，此时需要在 controller 中进行修改设置。即 jalr 一定会导致 pcsrcD = 1。

```
addr_adder pcadder3(rdata1D1, immoutD, pcadder2bD1); //jalr_1
set_last_zero set_zero(pcadder2bD1, pcadder2bD); //jalr_2

mux2 #(`XLEN) pcsrcmux2(pcadder2aD, pcadder2bD, jalrD, pcbranchD);
```

ID 阶段传递 PC 和 PC+4（便于进行选择和跳转操作的选择），指令译码。生成 I-type 的立即数（controller 中进行设置）。将立即数与 rdata1D 相加（pcadder），并置最后一位为 0（parts 中新建 set_zero），即为跳转后的结果，记为 pcadder2bD。

```
// write-back stage logic
mux2 #(`XLEN) wbmux1(aluoutW, memdataW, memtoregW, wdataW1);
mux2 #(`XLEN) wbmux2(wdataW1, pcplus4W, jalW, wdataW);
```

WB 阶段返回地址 PC+4 被写回 rd。

综上，ID 阶段按照 jalr 来区分 pcbranch 是 pcadder2aD 还是 pcadder2bD。控制信号 jal 代表 jal/jalr 的指令，jalr 特指 jalr。ID 阶段，jalrD 为 1 时，选择 pcadder2bD；否则，为 pcadder2aD。WB 阶段，rdM 传为 waddrW，将 pcplus4W 作为 wdataW 写回 reg（jalW = 1）。

4.11.6 B 型运算指令的实现

```
//assign pcsrc = 0; //pcsrc = 0 means pc + 4, need to change
//S5
assign pcsrc = rv32_jal | rv32_jalr | (rv32_beq & zero) | (rv32_bge & (! lt)) | (rv32_bgeu & (! lt)) |
               (rv32_blt & lt) | (rv32_bltu & lt) | (rv32_bne & (! zero));
```

判断指令引入后，在 IF 阶段将要进行下一个 pc 取值的判定（不考虑流水线的话），即 pc+4 还是 pcbranch。这是通过 pcsrcD 的设置来实现的。之前的 pcsrcD 一直默认为 0，此时需要在 controller 中进行修改设置。即发生 branch 时一定会导致 pcsrcD = 1。

```
// B-type imm????????????branch????????????x2??????
wire btype = rv32_branch;
```

```
sl1 immGen(immoutD, immoutD_sl1);
addr_adder pcadder2(pcD, immoutD_sl1, pcadder2aD); //jal or b-type
```

ID 阶段传递 PC 和 PC+4（便于进行选择和跳转操作的选择），指令译码。生成 B-type 的立即数（controller 中进行设置）。将立即数左移一位后，与 PC 相加，即为跳转后的结果，记为 pcadder2aD。

综上，ID 阶段按照 jalr 来区分 pcbranch 是 pcadder2aD 还是 pcadder2bD。控制信号 jalr 特指 jalr。ID 阶段，jalrD 为 1 时，选择 pcadder2bD；否则，为 pcadder2aD。注意 WB 阶段不用写回，jal 控制信号不能把 B-type 给覆盖进去。

4.11.7 跳转后误读指令清空

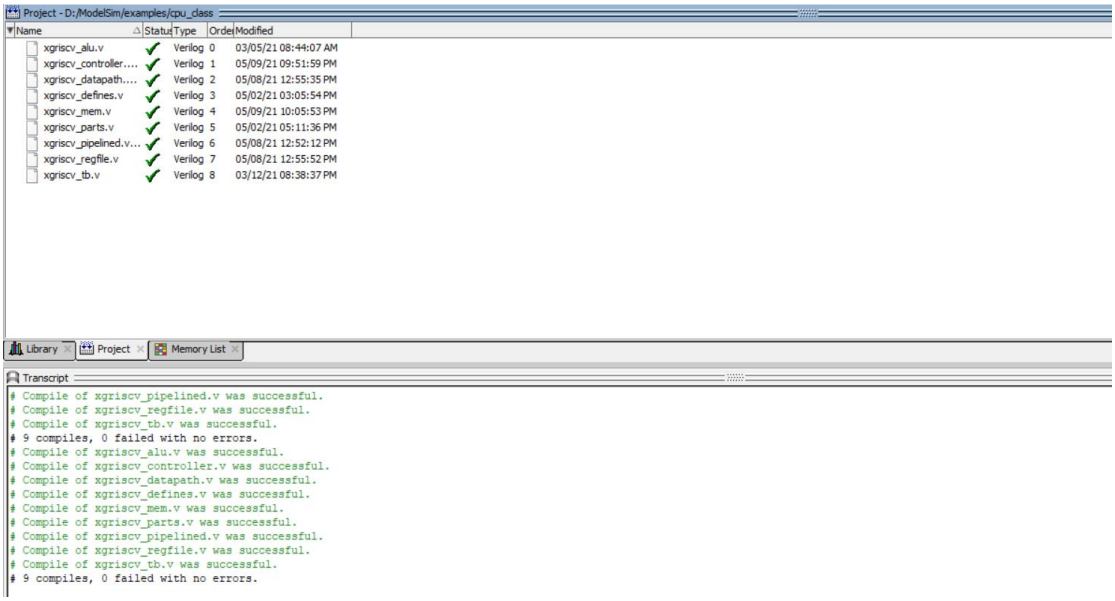
```
wire [`INSTR_SIZE-1:0]  instrD;
wire [`ADDR_SIZE-1:0]   pcD, pcplus4D;
wire flushD = pcsrcD | hazard;//clear
//mux2 #(1) flushDmux(0, 1, jalD, flushD);
//Data from IF/ID to ID/EX
floprc #(`INSTR_SIZE)   pr1D(clk, reset, flushD, instrF, instrD); // instruction
floprc #(`ADDR_SIZE)    pr2D(clk, reset, flushD, pcF, pcD); // pc now
floprc #(`ADDR_SIZE)    pr3D(clk, reset, flushD, pcplus4F, pcplus4D); // pc+4, maybe used to choose
```

跳转指令 pcsrcD 是在 ID 阶段计算得到，而此时下一条指令已经完成了 IF 阶段的取指令。在跳转成功发生时，应当对下一条指令进行清除。所以，当 pcsrcD = 1 时，将 flushD 设为 1，将下一条指令清零即可。

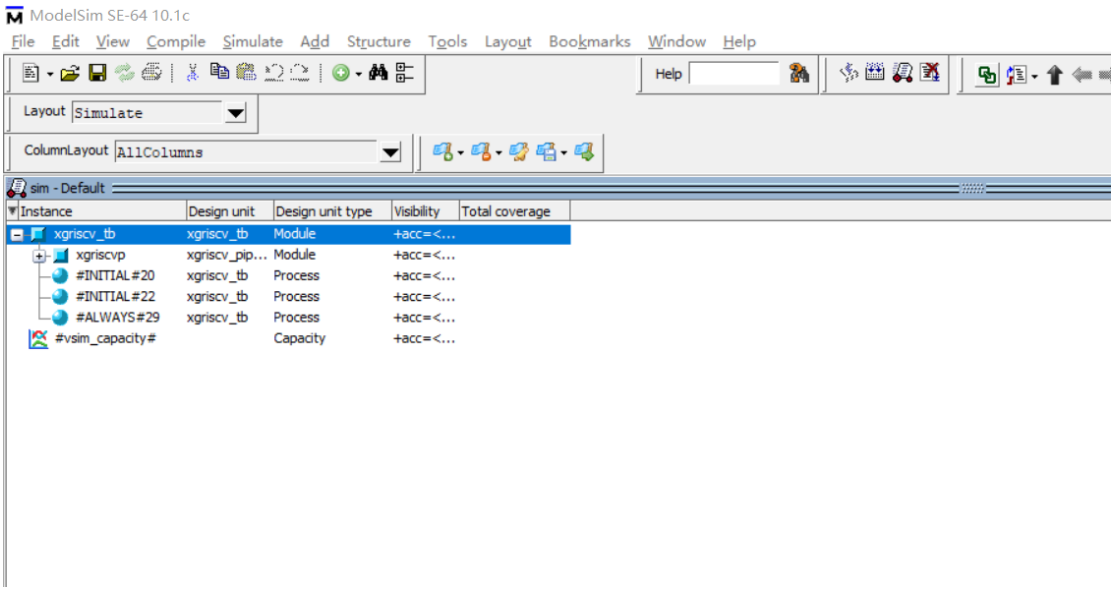
5 测试及结果分析

5.1 仿真代码及分析

仿真代码执行过程由 Modelsim 进行。



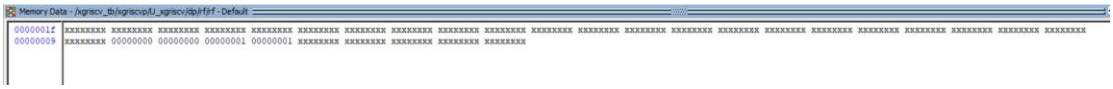
所有程序编译通过。



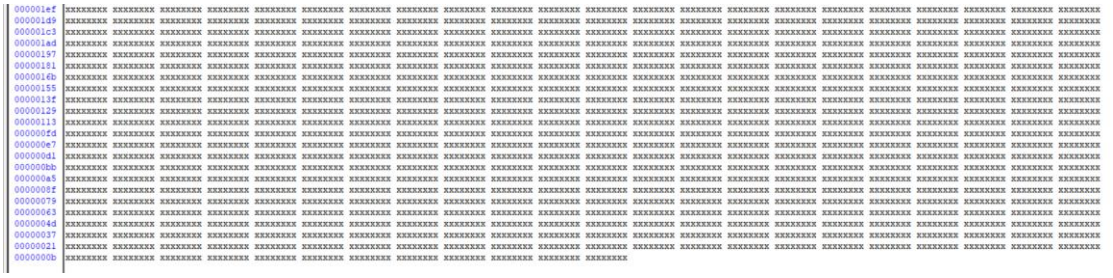
开始模拟。

```
# clock:           1
# pc:            80000000
# instr:         00100293
run
# clock:           2
# pc:            80000004
# instr:         00100313
run
# clock:           3
# pc:            80000008
# instr:         00000393
run
# clock:           4
# pc:            8000000c
# instr:         00000413
run
# x 5 = 00000001
# clock:           5
# pc:            80000010
# instr:         00000013
run
# x 6 = 00000001
# clock:           6
# pc:            80000014
# instr:         00000013
run
# x 7 = 00000000
# clock:           7
# pc:            80000018
# instr:         00628463
VSIM 3> run
# x 8 = 00000000
# clock:           8
# pc:            8000001c
# instr:         00140413
```

运行并查看运行过程中的数据变化。



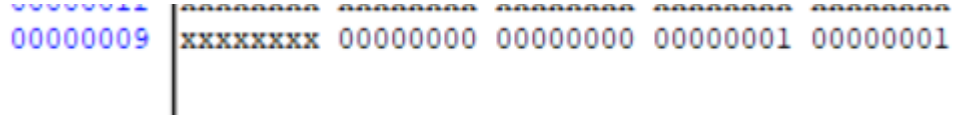
查看运行过程中的寄存器数据变化。

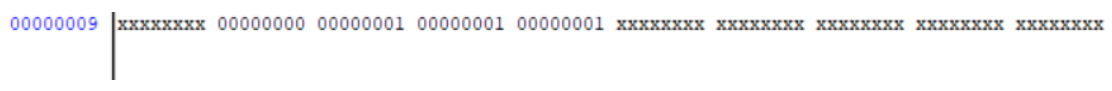


查看运行过程中的存储器变化。

5.2 仿真测试结果（仅展示未检查的部分）

Forwarding_sim1:



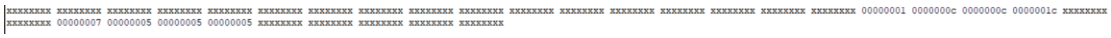


X7 在 0 与 1 间反复变化，x8 始终为 0

Forwarding_sim2:

X5：3，X6：2，X7：3，X8：5，X9：5，X11：30

Forwarding_sim3:



X5：5，X6：5，X7：5，X8：7，X9：5，X11：1c，X12：c，X13：c，X14：1

5.3 下载测试代码及分析

首先要对原有的代码进行修改，以适应 vivado 下板子的配置。

1. 删除 module imem 的定义
2. 删除 module xgriscv_pipeline 的定义
3. 修改 fpga_top.v 和 cpu 实现

```
input [4:0] reg_sel,
output [31:0] reg_data
```

在 piplined，datapath，regfile 中添加这两个输入和输出；

```
reg  [31:0] RAM[127:0]; //1024 RAM

assign rd = RAM[a[6:0]]; //Get data from RAM

always @(posedge clk)
  if (we)
    begin
    case (amp)
      4'b1111: RAM[a[6:0]] <= wd;             // sw
      4'b0011: RAM[a[6:0]][15:0] <= wd[15:0];  // sh
      4'b1100: RAM[a[6:0]][31:16] <= wd[15:0]; // sh
      4'b0001: RAM[a[6:0]][7:0] <= wd[7:0];    // sb
       4'b0010: RAM[a[6:0]][15:8] <= wd[7:0];   // sb
      4'b0100: RAM[a[6:0]][23:16] <= wd[7:0];  // sb
      4'b1000: RAM[a[6:0]][31:24] <= wd[7:0];  // sb
      default: RAM[a[6:0]] <= wd;// it shouldn't happen
     endcase
```

对 MEM 的大小进行修改，从 1024 个变成 128 个，相应的地址也要修改；

```
xgriscv  U_xgriscv(
 .clk(Clk_CPU),
 .reset(rst),
 .pc(PC),
 .instr(instr),
 .memwrite(MemWrite),
 .amp(cpu_data_amp),
 .daddr(cpu_data_addr),
 .writedata(cpu_data_out),
 .readdata(cpu_data_in),
 .reg_sel(sw_i[4:0]),
 .reg_data(reg_data)
 );
```

对 fpga 中的 top 文件进行相应的修改。

将学号修改后，将相应代码放入 venus

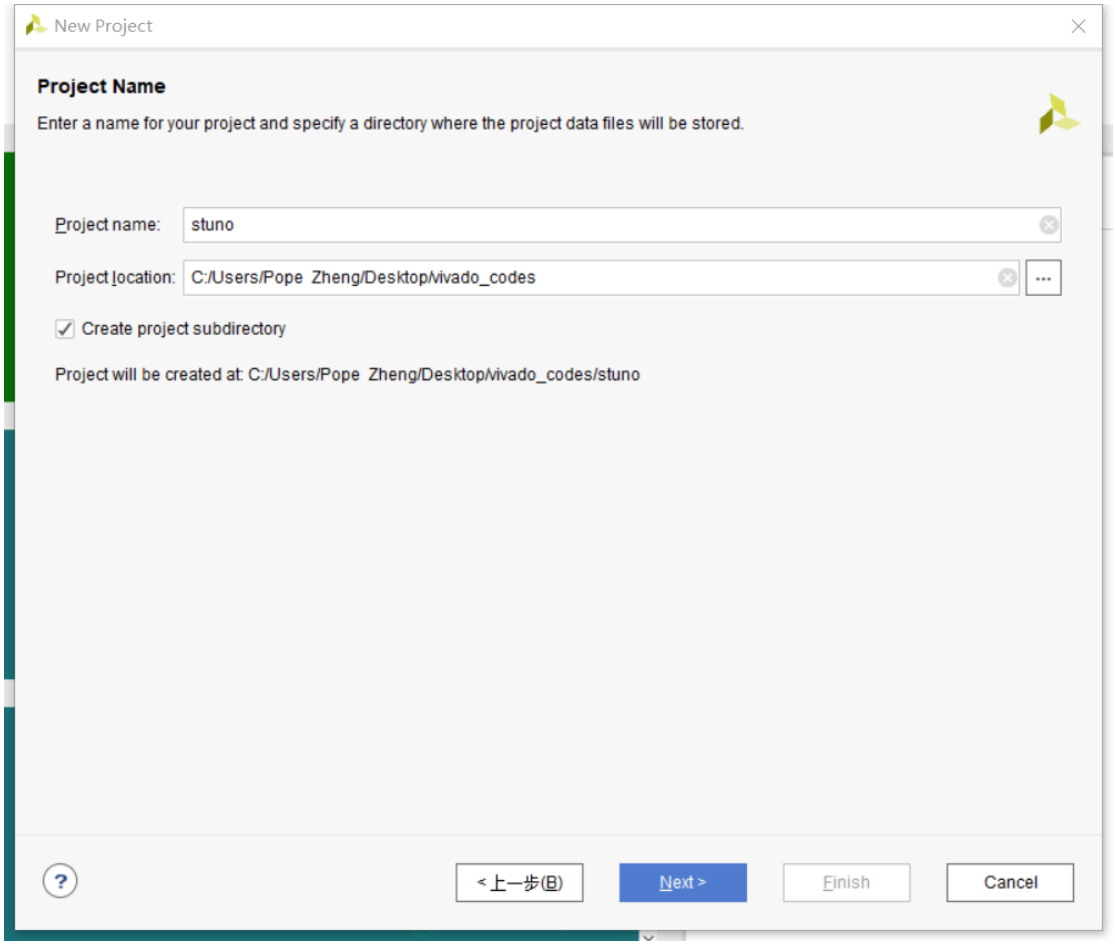
```
#################################################
    addi    x2, x0, 0x02        # This is the BCD encoding for the studnet no: a hex digit for each decimal digit
    slli    x2, x2, 8
    addi    x2, x2, 0x11
    slli    x2, x2, 16

    addi    x3, x0, 0x03
    slli    x3, x3, 8
    addi    x3, x3, 0x80
    add     x2, x2, x3          # set the student no., USE YOUR OWN STUDENT NO.!!!

    sw      x2, 0(x0)           # store the original stuno at data memory
    addi    x11, x0, 8          # the size of stuno, N = 8
    lw      x15, 0(x0)          # x15 = [0x0] = stuno
    add     x2, x0, x0          # the outer loop variable initilization, i = 0,
    addi    x4, x0, 0x0f        # mask0 = 0xf
loop1:
    and     x7, x15, x4         # a = sortedstuno & mask0, get the BCD to be processed
    slli    x9, x2, 2           # (4 * i)
    srl     x7, x7, x9          # a = a >> (4 * i), shift the BCD to the LSB 4 bits
    slli    x5, x4, 4           # mask1 = mask0 << 4
    add     x12, x2, x0         # bestj = i, remmember the position of the largest BCD in this loop
    add     x13, x7, x0         # tmpMax = a, remember the last BCD in this loop
    addi    x3, x2, 1           # j = i + 1, the inner loop variable initilization, j = i + 1
```

利用 venus 生成机器代码

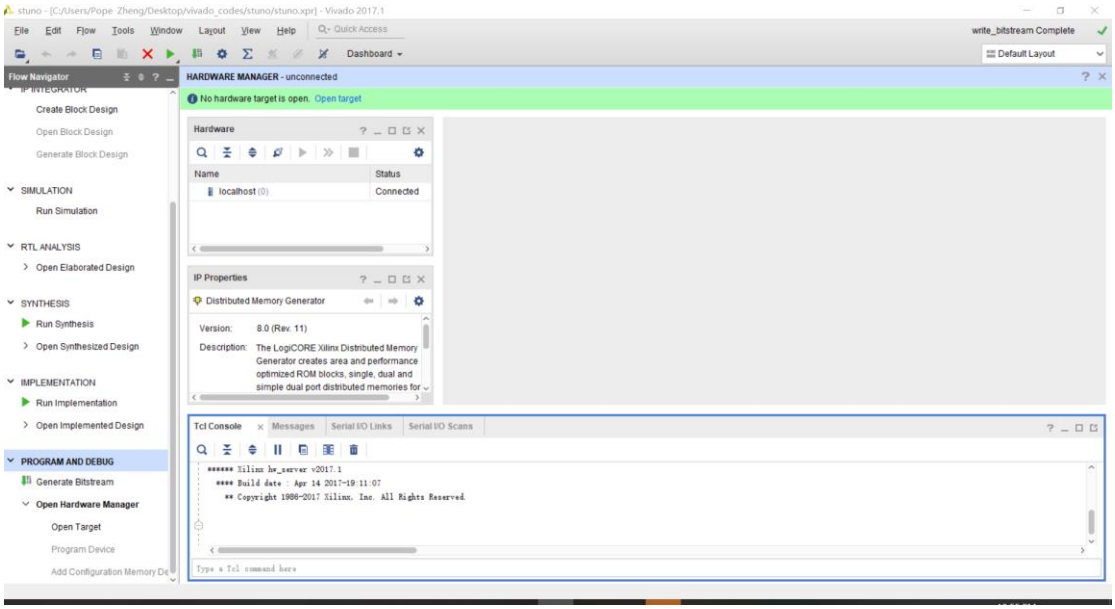
生成 coe 文件

```
memory_initialization_radix=16;
memory_initialization_vector=
00200113,
00811113,
01110113,
01011113,
00300193,
00819193,
08018193,
00310133,
00202023,
00800593,
00002783,
00000133,
```

创建 vivado 项目



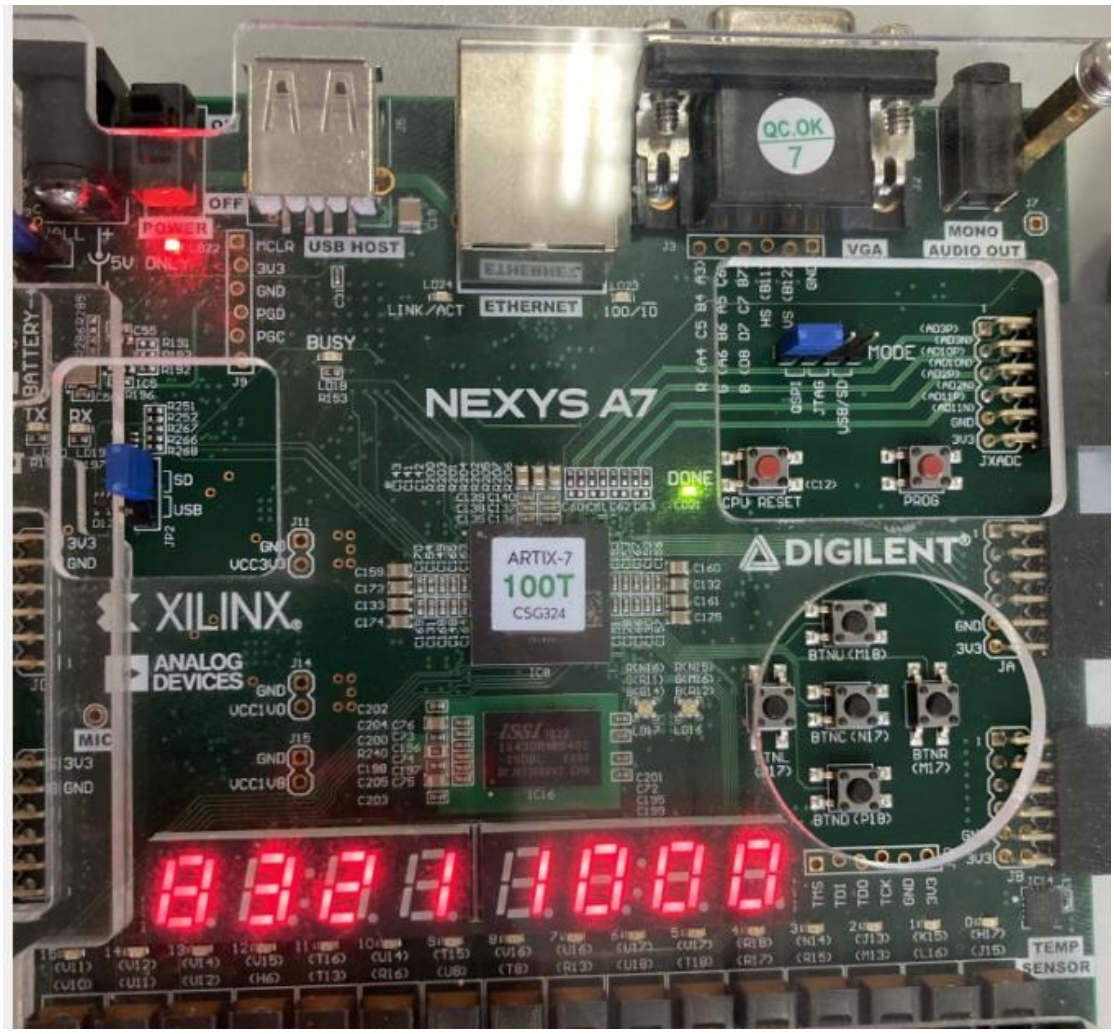
按照 PPT 中的流程一步步实现（不再赘述）



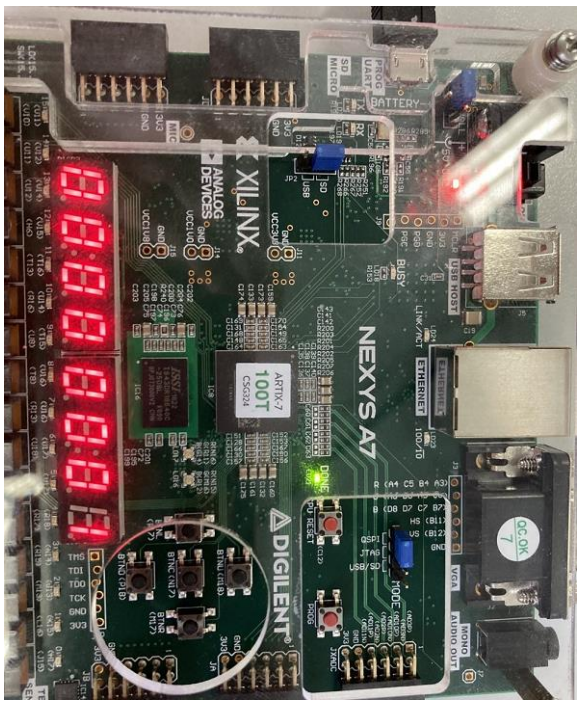
成功编译。

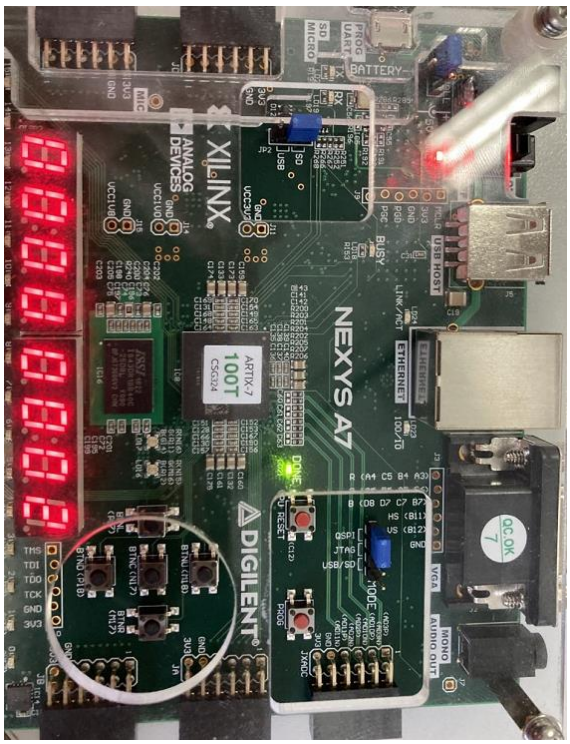
5.4 下载测试结果

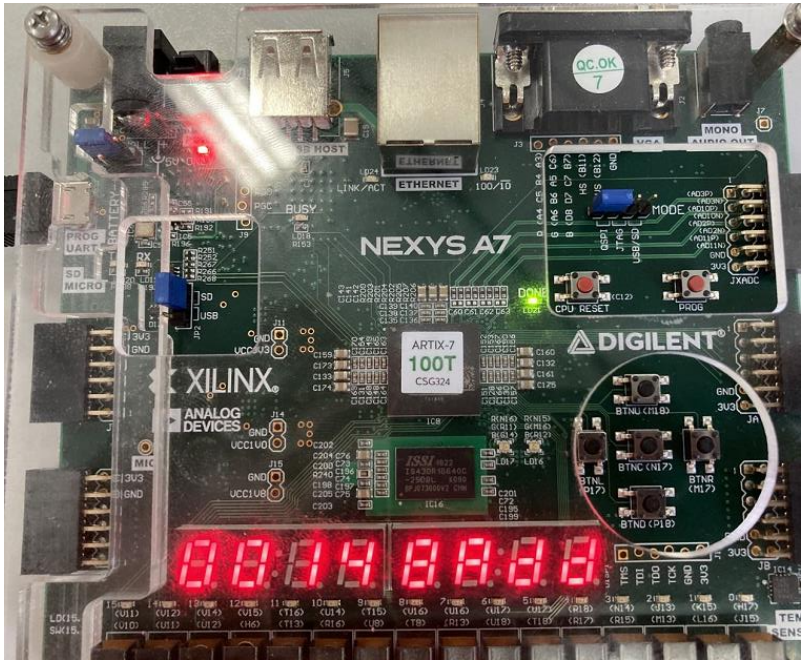
学生学号排序，输入 02110380

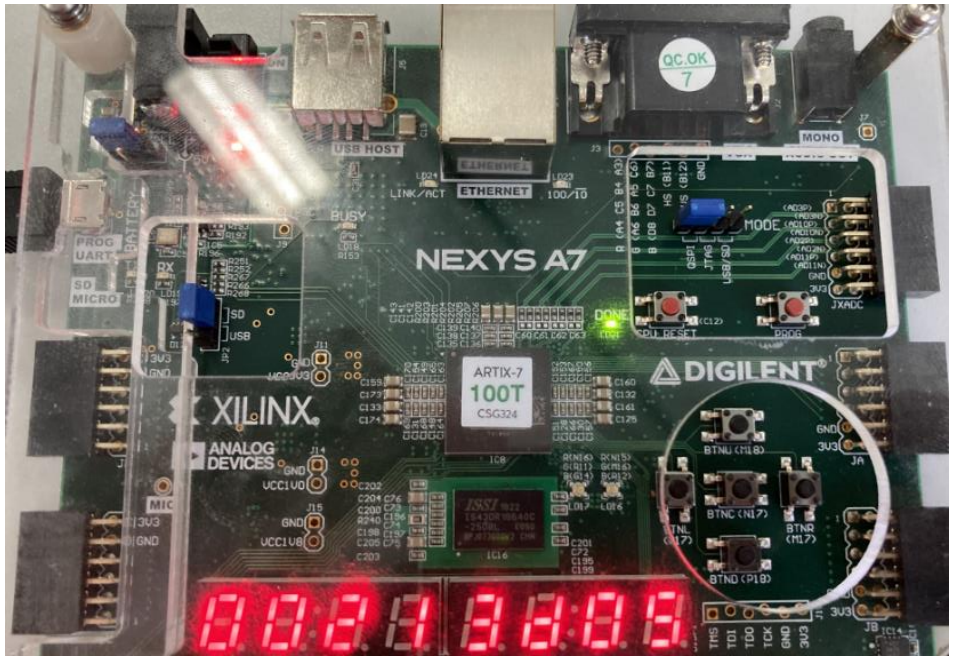


斐波那契数列，仅展示前两个和最后两个






NEXYS A7
DIGILENT
XILINX
ANALOG DEVICES
ARTIX-7
100T
CSG324
ETHERNET
USB HOST
MODE
CPU RESET
PROG
BTNU (M18)
BTNC (N17)
BTNL (P17)
BTNR (M17)
BTND (P18)


NEXYS A7
DIGILENT
XILINX
ANALOG DEVICES
ARTIX-7
100T
CSG324
ETHERNET
USB HOST
MODE
CPU RESET
PROG
BTNU (M18)
BTNC (N17)
BTNL (P17)
BTNR (M17)
BTND (P18)

参考文献

教师评语评分

评语：_____

_____

_____

_____

_____

_____

_____

_____

评分：_____

评阅人：

年　　月　　日

（备注：对该实验报告给予优点和不足的评价，并给出百分之评分。）