武汉大学计算机学院
本科生课程设计报告

RISC_V CPU 设计

专 业 名 称 ：计算机科学与技术

课 程 名 称 ：计算机系统综合设计

指 导 教 师 ：蔡朝晖 副教授

学 生 学 号 ：2022302191219

学 生 姓 名 ：徐嘉浩

二〇二四年七月

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：徐嘉浩　　日期：2024.7.20

摘　要

计算机系统综合设计实验的实验目的是深入了解 CPU 处理指令的原理。完成 RISC-V 架构的 CPU 设计，能够使用流水线的方式实现 37 条常用指令，并且额外完成了计时器中断以及按键中断。

实验设计主要遵循《计算机组成与设计：硬件/软件接口 RISC-V（第五版）》[1]中介绍的 CPU 设计原理和结构，蔡朝晖老师提供的 CPU 整体框架以及 NEXYS A7 开发板的相关接口要求。

实验内容主要包括：用 Verilog 语言完成单周期和流水线 CPU 设计，并增加了计时器中断和按键中断的复杂运用。其中流水线部分通过旁路解决数据冒险问题，并在 EX 阶段后处理 beq 等跳转指令。相关代码程序先在 Vivado 仿真软件上完成仿真测试，然后烧录到 NEXYS A7 开发板上，通过简单“风车”测试程序和复杂“AC”测试程序。

实验结论为单周期和流水线 CPU 代码均通过 Vivado 仿真软件上的仿真测试，PC 和相关寄存器值的变化结果符合预期。单周期和流水线 CPU 代码烧录到 NEXYS A7 开发板后，先后通过“风车”、“矩形变换”、“AC”等测试代码。说明相关 CPU 代码实现正确。流水线 CPU 采用旁路解决数据冒险问题，并在 EX 阶段后处理 beq 等跳转指令，能够在 CLK[0]的情况下通过测试代码。说明流水线 CPU 代码实现高效。

关键词：RISC-V；CPU；流水线；中断

目　录

5 测试及结果分析

6 实验总结

1 引言

1.1 实验目的

本实验是对计算机组成原理课程知识的综合运用，帮助学生深入了解 CPU 处理指令的原理。学生独立完成 RISC-V 架构的单周期和流水线 CPU 设计，能够实现 37 条常用指令，锻炼了学生的独立研究能力和动手实践能力。学生两两组队额外完成计时器中断以及按键中断等复杂运用，培养了学生团队合作的能力。

1.2 国内外研究现状

RISC-V 架构最早由美国加州大学伯克利分校的 Krste Asanovic 教授、Andrew Waterman 和 Yunsup Lee 等开发人员于 2010 年发明，并且得到了计算机体系结构领域的泰斗 David Patterson 的大力支持。

RISC-V 基金会成立以来，许多公司已经采用了 RISC-V 架构设计处理器。其中一些公司包括 Nvidia、Western Digital、SiFive、Andes Technology 等。这些公司在不同领域应用 RISC-V，包括嵌入式系统、物联网设备和数据中心。

近年来，随着 RISC-V 架构的不断改进和创新，RISC-V 处理器完成了众多商业实现。例如阿里巴巴玄铁 910，这是阿里巴巴旗下半导体公司平头哥发布的它的首款 RISC-V 处理器，采用 3 发射 8 执行的复杂乱序执行架构，是业界首个实现每周期 2 条内存访问的 RISC-V 处理器。晶心科技公司也一直提供可配置性高的 32/64 位高性能 CPU 核心，包含 DSP、FPU、Vector、超标量、乱序执行及多核心系列。

2 实验环境介绍

2.1 Verilog HDL

Verilog HDL 是一种硬件描述语言，用于设计和描述数字电路和系统的行为和结构。Verilog HDL 可以用来表示逻辑电路图、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 广泛应用于数字电路设计、仿真、验证和综合等领域。

2.2 Venus

Venus 是一个开源的 RISC-V 指令集架构模拟器和仿真器。它主要用于教育和研究目的，用于探索和理解 RISC-V 架构。Venus 提供了一个平台，可以编写和执行 RISC-V 汇编代码，使用户能够研究 RISC-V 指令的行为和 RISC-V 处理器的内部工作原理。Venus 使用 Java 编写，并在 Java 虚拟机上运行，使其具有跨平台性，在各种操作系统上都可以使用。它由 RISC-V 社区积极维护和更新，来自全球研究人员和爱好者的贡献。

2.3 ModelSim

ModelSim 是一种常用的硬件描述语言仿真器，由 Mentor Graphics 开发和提供。它广泛用于数字电路设计和验证，支持多种硬件描述语言，包括 VHDL 和 Verilog HDL。ModelSim 提供了一个强大的仿真环境，用于验证和调试设计的行为和时序。ModelSim 在数字电路设计和验证领域得到广泛应用，被许多工程师和研究人员用于验证和验证各种复杂的设计，从简单的逻辑电路到大规模的系统级设计。它是一个强大的工具，有助于加快设计开发过程并减少设计错误的风险。

2.4 Vivado

Vivado 是由 Xilinx 开发的集成电路设计工具套件，用于设计、验证和实现

FPGA 和 SoC 设备。它提供了一个全面的开发平台，支持从设计输入到最终生成可部署到硬件的比特流的完整设计流程。Vivado 用芯片开发和高级综合系统的附加功能取代了 Xilinx ISE。Vivado 是一个功能强大且广泛采用的集成电路设计工具，被广泛用于各种应用领域，包括通信、图像和视频处理、嵌入式系统和高性能计算等。它提供了一套完整的工具链，帮助工程师实现高度定制化的硬件设计，并加速产品开发周期。

2.5 Nexys A7

Nexys A7 是 Digilent 推出的一款 FPGA 开发板，旨在支持学习、原型设计和项目开发。它基于 Xilinx Artix-7 系列 FPGA 芯片，并提供了丰富的硬件资源和接口，适用于各种数字电路设计和嵌入式系统开发。

3 概要设计

3.1 总体设计

采用五阶段流水线设计，五个阶段分别为取指令（IF）、译码（ID）、运算（EX）、存储（MEM）、写回（WB）。该流水线 CPU 支持 37 条基本指令：add, sub, and, or, lw, sw, beq, jalr, jal, ori, xor, xori, andi, addi, sll, sra, srl, slt, sltu, srai, slti, sltiu, slli, srli, lui, lb, lh, lbu, lhu, sb, sh, bne, blt, bge, bltu, bgeu, auipc，包括 R 型、I 型、S 型、SB 型、U 型和 J 型六种不同类型的指令。该流水线 CPU 通过旁路的方式解决数据冒险问题，并在 EX 阶段后处理 beq 等跳转指令。

以下为流水线 CPU 五个阶段的大致功能介绍：

1. IF 阶段

IF 阶段是获取指令的阶段。遇到上升沿信号，调用 PC 模块更新当前的 PC 值，然后获取当前 PC 值对应的指令 inst_in。根据指令 inst_in 初步获取 R 型、I 型、I_shamt 型(slli 等)、S 型、SB 型、U 型和 J 型七种不同类型指令的立即数以及 Opcode、Funct3、Funct7。最后将这些后续阶段要用到的信息放入 IF_ID 寄存器，准备传给后续阶段。

2. ID 阶段

ID 阶段是译码的阶段。该阶段包含处理控制信号的模块 Ctrl、处理立即数的模块 EXT 以及能够读取寄存器值的模块 RF。Ctrl 模块通过在 IF_ID 寄存器中获取的 IF_ID_Opcode、IF_ID_Funct3、IF_ID_Funct7 等信息，判断出是哪条指令并生成对应的控制信号。EXT 模块通过 Ctrl 模块生成的 EXTop 信号进一步处理立即数。RF 模块通过 IF_ID_rs1、IF_ID_rs2，在寄存器堆中读取相应寄存器的值。最后将这些后续阶段要用到的信息放入 ID_EX 寄存器，准备传给后续阶段。

3. EX 阶段

EX 阶段是进行运算的阶段。该阶段包含旁路的处理模块、针对 load 指令进行停顿的 Stall 模块、用来运算的 ALU 模块、处理 beq 等跳转指令生成寄存器 flush 信号的模块以及生成下一条指令 PC 的 NPC 模块。旁路的处理模块会根据

ForwardA 和 ForwardB 两个由 Forward 模块生成的信号，来选择更新 RD1 和 RD2。Stall 模块会判断是否是 load+alu 的指令组合以及是否产生数据冒险，然后通过 PC_we、IF_ID_we、ID_EX_flush 三个信号来控制是否停顿一个周期。ALU 模块则和单周期相同，通过 Ctrl 模块生成的 ALUOp 信号,处理不同的运算。处理 beq 等跳转指令的模块，会在 ALU 模块运算结束之后进行。通过 ALU 模块产生的 Zero 信号，去更新 NPCOp。最后是 NPC 模块，使用更新好后的 NPCOp 信号来确定下一跳指令的 PC 值。

4. MEM 阶段

MEM 阶段是对 data memory 的读写阶段。由于 data memory 使用的是 Vivado 提供的 IP 核，所以该阶段只需将 MemWrite、Addr_out、Data_out、dm_ctrl 四个信号和外部连线即可。同时这个阶段还需要处理 MEM 到 EX 的旁路，生成 ForwardA 和 ForwardB 信号。

5. WB 阶段

WB 阶段是对寄存器的值进行写回的阶段。通过 MEM_WB_WDSel 信号来判断写回寄存器的值是 alu 的结果还是 data memory 中的数据还是 PC+4。最后通过 RF 模块在 PC 信号的下降沿把值写入寄存器。

3.2 PC（程序计数器）

3.2.1 功能描述

遇到上升沿信号，调用 PC 模块更新当前的 PC 值。

3.2.2 模块接口

信号名	方向	描述
clk	input	时钟信号
rst	input	复位信号
NPC	input	提前计算出的下一条指令的地址
PC	output	输出当前指令的 PC

3.3 RF（寄存器文件）

3.3.1 功能描述

维护由 32 个寄存器构成寄存器堆，实现寄存器读写操作。为了完成前半周期写、后半周期读的要求，我们在遇到下降沿的信号时进行写寄存器操作。

3.3.2 模块接口

信号名	方向	描述
clk	input	时钟信号
rst	input	复位信号
RFWr	input	寄存器写使能
A1	input	rs1
A2	input	rs2
A3	input	rd
WD	input	写入寄存器的值
RD1	output	输出读取到的 rs1 的值
RD2	output	输出读取到的 rs2 的值

3.4 Ctrl（生成控制信号）

3.4.1 功能描述

Ctrl 模块通过在 IF_ID 寄存器中获取的 IF_ID_Opcode、IF_ID_Funct3、IF_ID_Funct7 等信息，判断出是哪条指令并生成对应的控制信号。需要注意的是，和单周期不同，由于还没有通过 ALU 算出 Zero 信号，所以 NPCOp 信号是不包含 Zero 的不完整信号。

3.4.2 模块接口

信号名	方向	描述
Op	input	Opcode
Funct7	input	Funct7
Funct3	input	Funct3
RegWrite	output	寄存器写信号
MemWrite	output	data memory 写信号
EXTOp	output	立即数控制信号
ALUOp	output	运算控制信号，判断用什么运算
NPCOp	output	NPC 控制信号，判断下个 PC 的值从哪里来

ALUSrc	output	判断用 rs2 运算还是 imm 运算
WDSel	output	判断写回寄存器的值从哪里来
DMCtrl	output	判断是读写字还是半字还是字节

3.5 EXT（生成立即数）

3.5.1 功能描述

EXT 模块通过 Ctrl 模块生成的 EXTOp 信号进一步处理立即数，包括乘 2 和符号扩展等操作。

3.5.2 模块接口

信号名	方向	描述
iimm_shamt	input	SLLI、SRLI、SRAL 指令的立即数
iimm	input	I 型指令立即数
simm	input	S 型指令立即数
bimm	input	SB 型指令立即数
uimm	input	U 型指令立即数
jimm	input	J 型指令立即数
EXTOp	input	立即数控制信号
immout	output	输出当前指令的正确立即数

3.6 ALU（运算）

3.6.1 功能描述

ALU 模块和单周期相同，通过 Ctrl 模块生成的 ALUOp 信号，处理不同的运算。其中 beq 指令的运算放在了 sub 运算里面。

3.6.2 模块接口

信号名	方向	描述
A	input	rs1 的值
B	input	rs2 的值或是立即数
ALUOp	input	运算控制信号，判断用什么运算
PC	input	该指令的 PC 值
C	output	运算结果
Zero	output	运算结果是否为 0，用于判断 beq

3.7 NPC（生成下一条指令 PC）

3.7.1 功能描述

处理 beq 等跳转指令的模块，会在 ALU 模块运算结束之后进行。通过 ALU 模块产生的 Zero 信号，去更新 NPCOp。然后使用更新好后的 NPCOp 信号来确定下一条指令的 PC 值。

3.7.2 模块接口

信号名	方向	描述
PC	input	目前的 PC
NPCOp	input	NPC 控制信号，判断下个 PC 的值从哪里来
IMM	input	立即数
Zero	input	alu 结果是否为 0，用于判断 beq
aluout	input	alu 结果
ID_EX_PC	input	该指令的 PC 值
PC_we	input	PC 写使能
NPC	output	下个 PC 的值
IF_ID_flush	output	是否要清空 IF_ID 寄存器
ID_EX_flush	output	是否要清空 ID_EX 寄存器

3.8 Stall（处理停顿）

3.8.1 功能描述

Stall 模块会判断是否是 load+alu 的指令组合以及是否产生数据冒险。然后 Stall 模块会生成 PC_we、IF_ID_we、ID_EX_flush 三个信号来控制是否停顿一个周期。

3.8.2 模块接口

信号名	方向	描述
ID_EX_rd	input	ID_EX 阶段的 rd 值
IF_ID_rs1	input	IF_ID 阶段的 rs1 值
IF_ID_rs2	input	IF_ID 阶段的 rs2 值
ID_EX_WDSel	input	用来判断是否是 load 指令
IF_ID_we	output	IF_ID 寄存器写使能
PC_we	output	PC 写使能
ID_EX_flush	output	ID_EX 寄存器清空信号

3.9 Forward（处理前递）

3.9.1 功能描述

Forward 模块用来判断是否要进行 EX 旁路和 MEM 旁路，并生成两个控制信号 ForwardA 和 ForwardB。

3.9.2 模块接口

信号名	方向	描述
EX_MEM_rd	input	EX_MEM 阶段的 rd 值
MEM_WB_rd	input	MEM_WB 阶段的 rd 值
ID_EX_rs1	input	ID_EX 阶段的 rs1 值
ID_EX_rs2	input	ID_EX 阶段的 rs2 值
EX_MEM_RegWrite	input	EX_MEM 阶段的指令是否要写寄存器
MEM_WB_RegWrite	input	MEM_WB 阶段的指令是否要写寄存器
ForwardA	output	rs1 寄存器是否要旁路，从哪里旁路
ForwardB	output	rs2 寄存器是否要旁路，从哪里旁路

3.10 GRE_array（流水线寄存器）

3.10.1 功能描述

GRE_array 模块用来处理流水线寄存器，包括寄存器值的传递和 flush。

3.10.2 模块接口

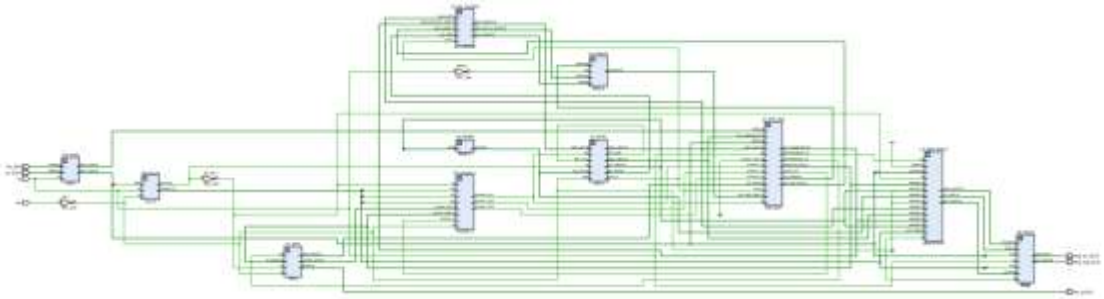
信号名	方向	描述
Clk	input	时钟信号
rst	input	复位信号
we	input	是否传递流水线寄存器的值
flush	input	是否清空流水线寄存器
in	input	上一个流水线寄存器写入该寄存器的内容
out	output	输出该寄存器的内容到下一个寄存器

4　详细设计

4.1 CPU 总体结构

CPU 总体结构框架包括：程序计数器 PC 模块、指令存储器 ROM_D 模块、数据存储器 RAM_B 模块、控制数据存取的 dm_controller 模块、控制数据传输的总线 MIO_BUS 模块、获取按键输入的 Enter 模块、调整时钟信号周期的 clk_div 模块、处理输出数据的多路选择器 Multi_8CH32 模块以及控制输出 Led 灯的 SSeg7 模块。其中指令存储器 ROM_D 模块和数据存储器 RAM_B 模块使用 Vivado 自带的 IP 核。程序计数器 PC 模块和控制数据存取的 dm_controller 模块由学生自行完成。其余模块使用蔡朝晖老师所提供的 edf 文件。

CPU 总体结构框架图如下图所示。



4.2 PC（程序计数器）

IF 阶段是获取指令的阶段。遇到上升沿信号，调用 PC 模块更新当前的 PC 值。PC 模块代码如下：

```
always @(posedge clk, posedge rst)
  if (rst)
    PC <= 32'h0000_0000;
  else
    PC <= NPC;//在上升沿信号更新 PC 为已经计算好的 NPC 的值
```

4.3 Ctrl（生成控制信号）

Ctrl 模块通过在 IF_ID 寄存器中获取的 IF_ID_Opcode、IF_ID_Funct3、

IF_ID_Funct7 等信息，判断出是哪条指令并生成对应的控制信号。需要注意的是，和单周期不同，由于还没有通过 ALU 算出 Zero 信号，所以 NPCOp 信号是不包含 Zero 的不完整信号。

```
module ctrl(Op, Funct7, Funct3,
            RegWrite, MemWrite,
            EXTOp, ALUOp, NPCOp,
            ALUSrc, WDSel,DMType,
            DMCtrl
            );

   input  [6:0] Op;       // opcode
   input  [6:0] Funct7;    // funct7
   input  [2:0] Funct3;    // funct3

   output       RegWrite; // control signal for register write
   output       MemWrite; // control signal for memory write
   output [5:0] EXTOp;    // control signal to signed extension
   output [4:0] ALUOp;    // ALU opertion
   output [2:0] NPCOp;    // next pc operation
   output       ALUSrc;   // ALU source for A
   output [2:0] DMType;
   output [1:0] WDSel;    // (register) write data selection
   output [2:0] DMCtrl;

  // r format
    wire rtype  = ~Op[6]&Op[5]&Op[4]&~Op[3]&~Op[2]&Op[1]&Op[0];
//0110011
    wire i_add  = rtype&
~Funct7[6]&~Funct7[5]&~Funct7[4]&~Funct7[3]&~Funct7[2]&~Funct7[1]&~F
unct7[0]&~Funct3[2]&~Funct3[1]&~Funct3[0]; // add 0000000 000
    wire i_sub  = rtype& ~Funct7[6]&
Funct7[5]&~Funct7[4]&~Funct7[3]&~Funct7[2]&~Funct7[1]&~Funct7[0]&~Fu
nct3[2]&~Funct3[1]&~Funct3[0]; // sub 0100000 000
    wire i_or   = rtype&
~Funct7[6]&~Funct7[5]&~Funct7[4]&~Funct7[3]&~Funct7[2]&~Funct7[1]&~F
unct7[0]& Funct3[2]& Funct3[1]&~Funct3[0]; // or 0000000 110
    wire i_and  = rtype&
~Funct7[6]&~Funct7[5]&~Funct7[4]&~Funct7[3]&~Funct7[2]&~Funct7[1]&~F
unct7[0]& Funct3[2]& Funct3[1]& Funct3[0]; // and 0000000 111
```

```
    wire i_xor  = rtype&
~Funct7[6]&~Funct7[5]&~Funct7[4]&~Funct7[3]&~Funct7[2]&~Funct7[1]&~F
unct7[0]& Funct3[2]&~Funct3[1]&~Funct3[0]; // xor 0000000 100
    wire i_sll  = rtype&
~Funct7[6]&~Funct7[5]&~Funct7[4]&~Funct7[3]&~Funct7[2]&~Funct7[1]&~F
unct7[0]&~Funct3[2]&~Funct3[1]& Funct3[0]; // sll  0000000 001
    wire i_sra  = rtype& ~Funct7[6]&
Funct7[5]&~Funct7[4]&~Funct7[3]&~Funct7[2]&~Funct7[1]&~Funct7[0]&
Funct3[2]&~Funct3[1]& Funct3[0]; // sra  0100000 101
    wire i_srl  = rtype&
~Funct7[6]&~Funct7[5]&~Funct7[4]&~Funct7[3]&~Funct7[2]&~Funct7[1]&~F
unct7[0]& Funct3[2]&~Funct3[1]& Funct3[0]; // srl  0000000 101
    wire i_slt  = rtype&
~Funct7[6]&~Funct7[5]&~Funct7[4]&~Funct7[3]&~Funct7[2]&~Funct7[1]&~F
unct7[0]&~Funct3[2]& Funct3[1]&~Funct3[0]; // slt  0000000 010
    wire i_sltu = rtype&
~Funct7[6]&~Funct7[5]&~Funct7[4]&~Funct7[3]&~Funct7[2]&~Funct7[1]&~F
unct7[0]&~Funct3[2]& Funct3[1]& Funct3[0]; // sltu 0000000 011

    // i format
    wire itype_l  = ~Op[6]&~Op[5]&~Op[4]&~Op[3]&~Op[2]&Op[1]&Op[0];
//0000011
    wire i_lw = itype_l& ~Funct3[2]& Funct3[1]& ~Funct3[0]; // lw 010
    wire i_lb = itype_l& ~Funct3[2]& ~Funct3[1]& ~Funct3[0];//lb 000
    wire i_lh = itype_l& ~Funct3[2]& ~Funct3[1]& Funct3[0];//lh 001
    wire i_lbu = itype_l& Funct3[2]& ~Funct3[1]& ~Funct3[0];//lbu 100
    wire i_lhu = itype_l& Funct3[2]& ~Funct3[1]& Funct3[0];//lhu 101

// i format
    wire itype_r  = ~Op[6]&~Op[5]&Op[4]&~Op[3]&~Op[2]&Op[1]&Op[0];
//0010011
    wire i_addi  =  itype_r& ~Funct3[2]& ~Funct3[1]& ~Funct3[0]; // addi
000
    wire i_ori  =  itype_r& Funct3[2]& Funct3[1]&~Funct3[0]; // ori 110
    wire i_andi  =  itype_r& Funct3[2]& Funct3[1]& Funct3[0]; // andi
111
    wire i_xori  =  itype_r& Funct3[2]& ~Funct3[1]& ~Funct3[0]; // xori
100
    wire i_slti  =  itype_r& ~Funct3[2]& Funct3[1]& ~Funct3[0]; // slti
010
```

```
    wire i_sltiu =  itype_r& ~Funct3[2]& Funct3[1]& Funct3[0]; // sltiu
011
    wire i_slli =  itype_r& ~Funct7[6]& ~Funct7[5]& ~Funct7[4]&
~Funct7[3]& ~Funct7[2]& ~Funct7[1]& ~Funct7[0]& ~Funct3[2]& ~Funct3[1]&
Funct3[0]; // slli 0000000 001
    wire i_srai =  itype_r& ~Funct7[6]& Funct7[5]& ~Funct7[4]&
~Funct7[3]& ~Funct7[2]& ~Funct7[1]& ~Funct7[0]& Funct3[2]& ~Funct3[1]&
Funct3[0]; // srai 0100000 101
    wire i_srli =  itype_r& ~Funct7[6]& ~Funct7[5]& ~Funct7[4]&
~Funct7[3]& ~Funct7[2]& ~Funct7[1]& ~Funct7[0]& Funct3[2]& ~Funct3[1]&
Funct3[0]; // srli 0000000 101

 //jalr
  wire i_jalr =Op[6]&Op[5]&~Op[4]&~Op[3]&Op[2]&Op[1]&Op[0];//jalr
1100111

  // s format
   wire stype  =
~Op[6]&Op[5]&~Op[4]&~Op[3]&~Op[2]&Op[1]&Op[0];//0100011
   wire i_sw   =  stype& ~Funct3[2]& Funct3[1]& ~Funct3[0]; // sw 010
   wire i_sb = stype& ~Funct3[2]& ~Funct3[1]& ~Funct3[0];//sb 000
   wire i_sh = stype& ~Funct3[2]& ~Funct3[1]& Funct3[0];//sh 001

  // sb format
   wire sbtype  =
Op[6]&Op[5]&~Op[4]&~Op[3]&~Op[2]&Op[1]&Op[0];//1100011
   wire i_beq  = sbtype& ~Funct3[2]& ~Funct3[1]&~Funct3[0]; // beq 000
   wire i_bne  = sbtype& ~Funct3[2]& ~Funct3[1]& Funct3[0]; // bne 001
   wire i_blt  = sbtype&  Funct3[2]& ~Funct3[1]&~Funct3[0]; // blt 100
   wire i_bge  = sbtype&  Funct3[2]& ~Funct3[1]& Funct3[0]; // bge 101
   wire i_bltu = sbtype&  Funct3[2]&  Funct3[1]&~Funct3[0]; // bltu 110
   wire i_bgeu = sbtype&  Funct3[2]&  Funct3[1]& Funct3[0]; // bgeu 111

 // j format
   wire i_jal  = Op[6]& Op[5]&~Op[4]& Op[3]& Op[2]& Op[1]& Op[0];  //
jal 1101111

   // u format
   wire utype = ~Op[6] & Op[4] & ~Op[3] & Op[2] & Op[1] & Op[0]; // u
0*10111
```

```
   wire i_lui = ~Op[6] & Op[5] & Op[4] & ~Op[3] & Op[2] & Op[1] & Op[0];
// lui 0110111
   wire i_auipc = ~Op[6] & ~Op[5] & Op[4] & ~Op[3] & Op[2] & Op[1] & Op[0];
// auipc 0010111

  // generate control signals
  assign RegWrite   = rtype | itype_r | i_jalr | i_jal | utype | itype_l;
// register write
  assign MemWrite   = stype;                          // memory write
  assign ALUSrc     = itype_r | stype | i_jal | i_jalr | utype |
itype_l;   // ALU B is from instruction immediate

  // signed extension
  // EXT_CTRL_ITYPE_SHAMT 6'b100000
  // EXT_CTRL_ITYPE       6'b010000
  // EXT_CTRL_STYPE       6'b001000
  // EXT_CTRL_BTYPE       6'b000100
  // EXT_CTRL_UTYPE       6'b000010
  // EXT_CTRL_JTYPE       6'b000001
  assign EXTOp[5]    = i_slli | i_srli | i_srai;
  assign EXTOp[4]    = i_ori | i_andi | i_jalr | i_addi | i_xori | i_sltiu
| itype_l;
  assign EXTOp[3]    = stype;
  assign EXTOp[2]    = sbtype;
  assign EXTOp[1]    = i_lui | i_auipc;
  assign EXTOp[0]    = i_jal;

  // WDSel_FromALU 2'b00
  // WDSel_FromMEM 2'b01
  // WDSel_FromPC  2'b10
  assign WDSel[0] = itype_l;
  assign WDSel[1] = i_jal | i_jalr;

  // NPC_PLUS4   3'b000
  // NPC_BRANCH  3'b001
  // NPC_JUMP    3'b010
  // NPC_JALR 3'b100
  assign NPCOp[0] = sbtype;//完整的应该是 sbtype&Zero；Zero 将在 EX 阶段补
充
  assign NPCOp[1] = i_jal;
```

```
  assign NPCOp[2]= i_jalr;

    assign ALUOp[0] =
itype_l|stype|i_addi|i_ori|i_add|i_or|i_lui|i_jalr|i_slli|i_sll|i_sr
a|i_sltu|i_sltiu|i_srai|i_bne|i_bge|i_bgeu;
    assign ALUOp[1] =
i_andi|i_jalr|itype_l|stype|i_addi|i_add|i_and|i_auipc|i_slli|i_sll|
i_slt|i_sltu|i_slti|i_sltiu|i_blt|i_bge;
    assign ALUOp[2] =
i_andi|i_and|i_ori|i_or|i_bge|i_blt|i_bne|i_sub|i_xori|i_slli|i_xor|
i_sll|i_beq;
    assign ALUOp[3] =
i_andi|i_and|i_ori|i_or|i_xori|i_slli|i_xor|i_sll|i_slt|i_sltu|i_slt
i|i_sltiu|i_bltu|i_bgeu;
    assign ALUOp[4] = i_sra|i_srl|i_srai|i_srli;

   // dm_word 3'b000
   // dm_halfword 3'b001
   // dm_halfword_unsigned 3'b010
   // dm_byte 3'b011
   // dm_byte_unsigned 3'b100
   assign DMCtrl[0] = i_lh | i_lb |i_sh | i_sb;
   assign DMCtrl[1] = i_lhu | i_lb | i_sb;
   assign DMCtrl[2] = i_lbu;

endmodule
```

4.4 EXT（生成立即数）

EXT 模块通过 Ctrl 模块生成的 EXTop 信号进一步处理立即数，包括乘 2 操作和符号扩展操作，以便后续阶段能够正确处理立即数操作。代码如下：

```
    always  @(*)
     case (EXTOp)//处理出 6 种不同类型指令的立即数
        `EXT_CTRL_ITYPE_SHAMT:   immout<={27'b0,iimm_shamt[4:0]};
        `EXT_CTRL_ITYPE:    immout <= {{20{iimm[11]}}, iimm[11:0]};
        `EXT_CTRL_STYPE:    immout <= {{20{simm[11]}}, simm[11:0]};
        `EXT_CTRL_BTYPE:    immout <= {{19{bimm[11]}}, bimm[11:0], 1'b0};
        `EXT_CTRL_UTYPE:    immout <= {uimm[19:0], 12'b0};
```

```
        `EXT_CTRL_JTYPE:    immout <= {{11{jimm[19]}}, jimm[19:0], 1'b0};
        default:            immout <= 32'b0;
    endcase
```

4.5 Stall（处理停顿）

Stall 模块会判断是否是 load+alu 的指令组合以及是否产生数据冒险。然后 Stall 模块会生成 PC_we、IF_ID_we、ID_EX_flush 三个信号来控制是否停顿一个周期。

```
    always@(*)
    begin
        if(((ID_EX_rd == IF_ID_rs1) || (ID_EX_rd == IF_ID_rs2)) &&
(ID_EX_WDSel == 2'b01) && (ID_EX_rd != 0))
        begin//满足 load+alu，停顿一周期
            IF_ID_we = 0;//IF_ID 寄存器不让写
            PC_we = 0;//PC 不让写
            ID_EX_flush = 1;//清空 ID_EX 寄存器
        end
        else begin
            IF_ID_we = 1;
            PC_we = 1;
            ID_EX_flush = 0;
        end
    end
```

4.6 ALU（运算）

ALU 模块则和单周期相同，通过 Ctrl 模块生成的 ALUOp 信号,处理不同的运算。其中 beq 指令的运算放在了 sub 运算里面。代码如下：

```
  always @( * ) begin
    case ( ALUOp )
      `ALUOp_nop:C=A;
      `ALUOp_lui:C=B;
      `ALUOp_auipc:C=PC+B;
      `ALUOp_add:C=A+B;
      `ALUOp_sub:C=A-B;//包括 beq 指令
      `ALUOp_bne:C={31'b0,(A==B)};
      `ALUOp_blt:C={31'b0,(A>=B)};
```

```
        `ALUOp_bge:C={31'b0,(A<B)};
        `ALUOp_bltu:C={31'b0,($unsigned(A)>=$unsigned(B))};
        `ALUOp_bgeu:C={31'b0,($unsigned(A)<$unsigned(B))};
        `ALUOp_slt:C={31'b0,(A<B)};
        `ALUOp_sltu:C={31'b0,($unsigned(A)<$unsigned(B))};
        `ALUOp_xor:C=A^B;
        `ALUOp_or:C=A|B;
        `ALUOp_and:C=A&B;
        `ALUOp_sll:C=A<<B;
        `ALUOp_srl:C=A>>B;
        `ALUOp_sra:C=A>>>B;
    Endcase
```

4.7 NPC（生成下一条指令 PC）

处理 beq 等跳转指令的模块，会在 ALU 模块运算结束之后进行。通过 ALU 模块产生的 Zero 信号，去更新 NPCOp。最后是 NPC 模块，使用更新好后的 NPCOp 信号来确定下一跳指令的 PC 值。此处将这两个模块合二为一，用一个 NPC 模块同步更新 NPCOp 和 NPC 的值。代码如下：

```
always @ (*) begin
    case(NPCOp)
        3'b000://普通指令
        begin
            NPC = PCPLUS4;
            IF_ID_flush = 0;
            ID_EX_flush = 0;
        end
        3'b001://sb 型指令
        begin
            if(Zero)//满足跳转条件
            begin
                NPC = ID_EX_PC + IMM;
                IF_ID_flush = 1;
                ID_EX_flush = 1;
            end
            else begin//不满足跳转条件
                NPC = PCPLUS4;
                IF_ID_flush = 0;
```

```
                ID_EX_flush = 0;
            end
        end
        3'b010://jal 指令
        begin
            NPC = ID_EX_PC + IMM;
            IF_ID_flush = 1;
            ID_EX_flush = 1;
        end
        3'b100://jalr 指令
        begin
            NPC = aluout;
            IF_ID_flush = 1;
            ID_EX_flush = 1;
        end
        default: begin
            NPC = PCPLUS4;
            IF_ID_flush = 0;
            ID_EX_flush = 0;
        end
    endcase
  end
```

4.8 Forward（处理旁路）

Forward 模块需要处理 EX 旁路和 MEM 旁路，生成 ForwardA 和 ForwardB 信号，用来分别记录 rs1 和 rs2 是否需要旁路，从哪里来的旁路。以 ForwardA 为例，代码如下：

```
always@(*)
    begin
        if((EX_MEM_rd === ID_EX_rs1)&&(EX_MEM_rd!=0)&&EX_MEM_RegWrite)
        begin
            ForwardA = 2'b10; //EX 旁路
        end
        else if((MEM_WB_rd ===
ID_EX_rs1)&&(MEM_WB_rd!=0)&&MEM_WB_RegWrite)
        begin
            ForwardA = 2'b01;//MEM 旁路
        end
```

```
        else begin
            ForwardA = 2'b00;//无旁路
        end
    end
```

4.9 RF（寄存器文件）

RF 模块用来处理寄存器的读和写的操作。为了实现先读取寄存器、后写入寄存器，我们选择在信号的下降沿写入寄存器并且在读寄存器时判断读和写的寄存器是否相同。相关代码如下：

```
always @(negedge clk, posedge rst)//下降沿写入
    if (rst) begin//reset
      for (i=1; i<32; i=i+1)
        rf[i] <= 0;
    end
    else
      if (RFWr && A3) begin
        rf[A3] <= WD;
      end
assign RD1 = (A1 != 0) ? ((RFWr && A3 == A1) ? WD : rf[A1]) : 0;
assign RD2 = (A2 != 0) ? ((RFWr && A3 == A2) ? WD : rf[A2]) : 0;
```

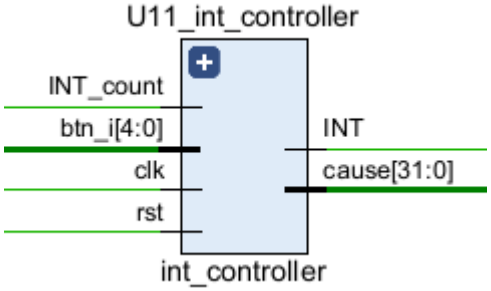
4.10 GRE_array（流水线寄存器）

GRE_array 模块用来处理流水线寄存器，包括寄存器值的传递和 flush。当能写信号 we 为 1 时，如果存在 flush 信号，就直接输出赋成 0 表示清空；如果没有 flush 信号，就把这一层流水线寄存器的值传给下一层。

```
    always@ (posedge Clk or posedge rst)
    begin
        if(rst || flush)begin out<=0; end
        else begin
            if(we)//能否写流水线寄存器
            begin
                if(flush) out <= 0;//清空流水线寄存器
                else out <= in;//传递
            end
        end
    end
```

4.11 复杂应用（中断）

我和陈昭宪合作实现了计时器中断和按键中断。



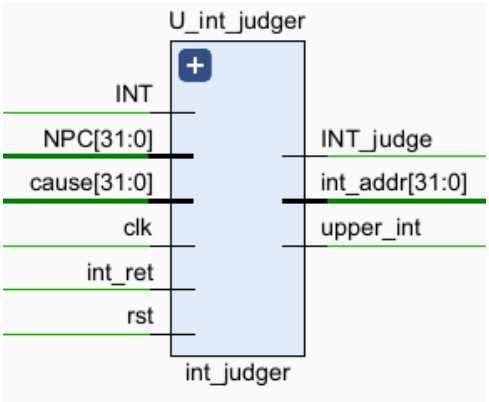
外围模块 int_controller 的功能：读取按键和计时器的输入，并按照按键优先于计时器的规则。这里考虑到长按按键 btn 会一直为 1，所以设计了按键和计时器 0 到 1 时才输出中断信号和中断原因。

信号名	方向	描述
clk	input	时钟信号
rst	input	复位信号
btn_i	input	按键输入
INT_count	input	计时器输入
INT	output	中断信号
cause	output	中断原因信号

```
`define count_cause {1'b1,31'd7}
`define btn_cause {1'b1,31'd11}
/*
output int signal, btn > counter
*/
module int_controller(
    input clk,
    input rst,
    input [4:0] btn_i,
    input INT_count,
    output reg INT,
    output reg [31:0] cause
    );

    reg [1:0] btn_i_input [4:0], count_input;
    reg edge_st,edge_ed;
    integer i,btn_cnt,btn_last,count_cnt,count_last;
```

```
initial
begin
    btn_cnt=0;
    btn_last=0;
    count_cnt=0;
    count_last=0;
    for(i=0;i<=4;i=i+1)
        btn_i_input[i]<=2'b00;
    count_input<=2'b00;
end


always@(posedge clk)
begin
    if(rst)
    begin
        btn_cnt=0;
        btn_last=0;
        count_cnt=0;
        count_last=0;
        for(i=0;i<=4;i=i+1)
            btn_i_input[i]<=2'b00;
        count_input<=2'b00;
    end
    else
    begin
        for(i=0;i<=4;i=i+1)
            btn_i_input[i]<={btn_i_input[i][0],btn_i[i]};
        count_input<={count_input[0],INT_count};
    end

    btn_last=btn_cnt;
    for(i=0;i<=4;i=i+1)
    begin
        edge_st=btn_i_input[i][0] & !btn_i_input[i][1];
        edge_ed=!btn_i_input[i][0] & btn_i_input[i][1];
        if(edge_st) btn_cnt=btn_cnt+1;
        if(edge_ed) btn_cnt=btn_cnt-1;
    end
```

```
        count_last=count_cnt;
        edge_st=count_input[0] & !count_input[1];
        edge_ed=!count_input[0] & count_input[1];
        if(edge_st) count_cnt=count_cnt+1;
        if(edge_ed) count_cnt=count_cnt-1;

        if(btn_last==0 && btn_cnt>0)
        begin
            INT=1;
            cause=`btn_cause;
        end
        else if(count_last==0 && count_cnt>0)
        begin
            INT=1;
            cause=`count_cause;
        end
        else
        begin
            INT=0;
            cause=0;
        end
    end

endmodule
```



CPU 内部的 int_judger 模块具有以下功能：接收一个中断信号、中断原因、int_ret、以及下一条指令地址（NPC），结合当前中断和中断信号，判断应当输出何种中断信号。btn_time 和 count_time 是计数器，用于记录尚未执行的按键和计时器信号数量。当一个中断信号到达时，对应的 btn_time 和 count_time 会增加 1。如果 btn_time 大于 0，则输出按键中断信号；否则，如果 count_time 大于 0，则输出计时器中断信号。此外，如果在执行计时器中断时接收到按键中断信号，则

会先暂停计时器中断，跳转到按键中断的地址，增加 count_time，处理完按键中断后再次处理计时器中断。（尽管这种方法简单，无法完全恢复计时器中断的现场，因此对中断服务函数有特定要求）。

INT	input	中断信号
NPC	input	下一条指令的 PC
cause	input	中断原因信号
clk	input	时钟信号
int_ret	input	中断停止信号
rst	input	复位信号
INT_judge	output	判断后的中断信号
int_addr	output	中断服务函数的地址
upper_int	output	是否执行更高级的中断

相关代码如下

```
`define count_cause {1'b1,31'd7}
`define btn_cause {1'b1,31'd11}
module int_judger(
    input clk,
    input rst,
    input INT,
    input [31:0] cause,
    input int_ret,
    input [31:0] NPC,
//    input is_int,
//    input [31:0] curr_cause,
//    input enable_int,
//    input enable_btn,
//    input enable_count,
    output reg INT_judge,
    output [31:0] cause_judge,
    output reg [31:0] int_addr,
    output reg upper_int
    );

    reg is_int;
    reg [31:0] curr_cause;
    reg enable_int,enable_btn,enable_count,sub_btn,sub_count,add_count;
    reg [31:0] mpc;
    reg [31:0] btn_time,count_time;

    initial
    begin
```

```
        is_int=0;
        curr_cause=0;
        btn_time=0;
        count_time=0;
        sub_btn=0;
        sub_count=0;
        add_count=0;
        mpc=0;
    end

    always @ (posedge clk or posedge rst) begin
        if (rst) begin
            btn_time <= 0;
            count_time <= 0;
            sub_btn <= 0;
            sub_count <= 0;
            add_count <= 0;
            is_int <= 0;
            curr_cause <= 0;
        end else begin
            if (INT) begin
                if (cause == `btn_cause) btn_time <= btn_time + 1;
                else if (cause == `count_cause) count_time <= count_time +
1;
            end else if (sub_btn) begin
                sub_btn <= 0;
                btn_time <= btn_time - 1;
            end else if (sub_count) begin
                sub_count <= 0;
                count_time <= count_time - 1;
            end else if (add_count) begin
                add_count <= 0;
                count_time <= count_time + 1;
            end

            if (int_ret) begin
                int_addr <= mpc;
                is_int <= 0;
                curr_cause <= 0;
            end else
```

```
            if (!is_int && (btn_time > 0 || count_time > 0)) begin
                INT_judge <= 1;
                mpc <= NPC;
                is_int <= 1;
                upper_int <= 0;
                if (btn_time > 0) begin
                    sub_btn <= 1;
                    curr_cause <= `btn_cause;
                    int_addr <= 32'h0000019c; // not done
                end else if (count_time > 0) begin
                    sub_count <= 1;
                    curr_cause <= `count_cause;
                    int_addr <= 32'h0000019c; // not done
                end
            /*else if(is_int && btn_time > 0) begin
                sub_btn <= 1;
                is_int <= 0;
                curr_cause <= 0;
                int_addr <= mpc;
            end
            */
            end else if (is_int && curr_cause == `count_cause && btn_time >
0) begin
                sub_btn <= 1;
                add_count <= 1;
                INT_judge <= 1;
                curr_cause <= `btn_cause;
                int_addr <= 32'h0000019c;
                upper_int <= 1;
            end else begin
                INT_judge <= 0;
                int_addr <= mpc;
                upper_int <= 0;
            end
        end
    end

    assign cause_judge=curr_cause;

endmodule
```

另外还进行了一些轻微的函数修改。特别是在中断发生时需要保存流水线的状态。因此，在对寄存器文件（RF）和指令寄存器（GRE_array）的处理上做了一些调整，新增了 save_out 和 load_out 信号，用于表示中断前后的处理。此外，在计时器中断转向按键中断时，不需要保存现场，以免覆盖流水线的状态。

RF:

```
    else if(save_out)
    begin
      for (i=1; i<32; i=i+1)
      begin
        rf_int[i] <= rf[i];
        rf[i] <= 0;
      end
    end
    else if(load_out)
    begin
       for (i=1; i<32; i=i+1)
        rf[i] <= rf_int[i];
    end
```

GRE_array:

```
if(write_enable)
begin
   if(save_out)
   begin
       if(!upper_int) out_int<=out;
       out<=0;
   end
   else if(flush)
       out<=0;
   else if(load_out)
       out<=out_int;
   else
       out<=in;
end
```

以下是我们测试的中断服务函数：

```
add x1, x0, x0
lui x2, 0x01000
addi x31, x0, 0xe
slli x18, x31, 28
and x7, x7, x0
addi x7, x7, 0xa
slli x7, x7, 4
sw x7, 0(x18)
bne x1, x2, -12

00e00f93 01cf9913 0003f3b3 00a38393 00439393 00792023
```

这段代码会放到 I_mem 后面，其作用是，中断时在数码管上输出一个 A。

5 测试及结果分析

5.1 仿真代码及分析

编写了一段测试代码，考察了 addi+addi 的 EX 旁路、lb+addi 的 MEM 旁路、beq 所代表的跳转指令。RISC_V 命令如下：

```
.section .text
.global _start

_start:
    # 初始化寄存器
    addi x1, x0, 1            # 0x0000: x1 = 1
    addi x1, x1, 2            # 0x0004: x1 = 3 测试 EX 旁路
    addi x2, x0, 3            # 0x0008: x2 = 3
    beq x1, x2, label_beq       # 0x000c: 测试 beq
    addi x2, x2, 1            # 0x0010: 跳过

label_beq:
    sb x2, 0(x2)              # 0x0014: beq 成功跳转, 0(x2) = 3
    lb x3, 0(x2)                  # 0x0018: x3 = 3
    addi x4, x3, 1            # 0x001c: x4 = 4 测试 MEM 旁路
```

将该 RISC_V 指令转换成对应的汇编指令放入指令寄存器中即可仿真测试。对应的汇编指令为：

```
;1.asm
memory_initialization_radix=16;
memory_initialization_vector=
00100093,
00208093,
00300113,
00208463,
00110113,
00210023,
00010183,
00118213;
```

5.2 仿真测试结果

观察 PC 的变化。当 PC=0000000c 时，是一条 beq 指令，该指令会在 EX 阶段判断是否成立。所以在 PC=00000014 时判断 beq 的结果为成立，然后跳转，PC 会跳转到 00000020，开始执行 00000020 地址上的指令。所以可以在上图中看到 PC 的值在 00000014 之后直接就变成了 00000020，且 NPCOp 也等于 1。

数据冒险有 EX 旁路和 MEM 旁路两个处理方式。我在流水线 CPU 的设计中使用了一个 Forward 模块来判断是否产生 EX 旁路和 MEM 旁路。判断 EX 旁路的条件是(EX_MEM_rd == ID_EX_rs1)&&(EX_MEM_rd!=0)&&EX_MEM_RegWrite)。判断 MEM 旁路的条件是(MEM_WB_rd == ID_EX_rs1)&&(MEM_WB_rd!=0)&&MEM_WB_RegWrite)且不满足 EX 旁路的条件。然后我们生成 ForwardA 和 ForwardB 信号，用来分别记录 rs1 和 rs2 是否需要旁路，从哪里来的旁路。有了前递信号，我就可以在 EX 阶段开始的时候，通过前递信号更新最新的 rs1、rs2 寄存器的值。以 ForwardA 为例，代码如下：

```
always@(*)
begin
    case(ForwardA)
        2'b00: RD1_new = ID_EX_RD1;
        2'b01: RD1_new = WD;
        2'b10: RD1_new = EX_MEM_aluout;
        default: RD1_new = ID_EX_RD1;
    endcase
end
```

对于 beq 这类的控制冒险，我的处理方式是：在 EX 阶段完成 alu 运算后，直接通过 alu 算出的 Zero 信号，配合上之前 Ctrl 生成的初步 NPCOp，直接求出下一条指令的地址 NPC。相关代码如下：

```
always @ (*) begin
```

```
  case(NPCOp)
      3'b000://普通指令
      begin
          NPC = PCPLUS4;
          IF_ID_flush = 0;
          ID_EX_flush = 0;
      end
      3'b001://sb 型指令
      begin
          if(Zero)//满足跳转条件
          begin
              NPC = ID_EX_PC + IMM;
              IF_ID_flush = 1;
              ID_EX_flush = 1;
          end
          else begin//不满足跳转条件
              NPC = PCPLUS4;
              IF_ID_flush = 0;
              ID_EX_flush = 0;
          end
      end
      3'b010://jal 指令
      begin
          NPC = ID_EX_PC + IMM;
          IF_ID_flush = 1;
          ID_EX_flush = 1;
      end
      3'b100://jalr 指令
      begin
          NPC = aluout;
          IF_ID_flush = 1;
          ID_EX_flush = 1;
      end
      default: begin
          NPC = PCPLUS4;
          IF_ID_flush = 0;
          ID_EX_flush = 0;
      end
  endcase
end
```

5.3 下载测试代码及分析

使用了蔡朝晖老师提供的一个“风车”、“矩形变换”的测试代码以及一个“AC”的测试代码。当测试通过时，根据相应的按键，会显示“风车”、“矩形变换”以及一个一直在移动的“AC”。

5.4 下载测试结果

流水线 CPU 代码烧录到开放板后，能够正确显示“风车”、“矩形变换”以及一个一直在移动的“AC”。测试结果如下：

6 实验总结

6.1 实验总结

计算机组成原理课程实验分成单周期CPU仿真和下板，流水线CPU仿真和下板以及计时器中断和按键中断的复杂应用五个部分。

实验的难点有很多，从最开始Vivado软件的使用，到CPU模块连线，再到仿真、下板，会遇到各种各样的问题，需要学生有足够的耐心和细心，才能把代码调对、跑出正确的结果。

最后的复杂应用环节，会更加考察学生的上网自学能力和团队合作能力。

我在实验的过程中也遇到了许多问题。在Vivado软件的使用过程中遇到了奇怪的黑盒问题，最后通过重新建工程的方式解决。Dm_controller中需要把data复制到正确的高位，而不是简单生成控制信号。流水线CPU最后WB阶段写回的值应该是MEM_WB_Data_in，不能直接使用MEM阶段获得的Data_in。

综上所述，计算机组成原理课程实验是一个硬核且有意义的实验。

6.2 取得的收获

经历了为期3周的计算机组成原理课程实验，我收获了很多：

1. 对CPU处理指令的方式有了更深的理解。
2. 学会了Vivado仿真软件以及NEXYS A7开发板使用方式。
3. 锻炼了我写代码做项目的耐心和细心。
4. 提高了我的上网查找资料自学的能力以及团队合作能力。
5. 结合嵌入式的知识，对软硬件的连接处理有了更深的体会。

参考文献

[1] David Patterson,John L.Hennessy.计算机组成与设计硬件/软件接口[M].第5版.北京：机械工业出版社，2020年

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。）