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： 计算机组成与设计课程设计

指 导 教 师： 蔡朝晖 副教授

学 生 学 号：

学 生 姓 名：

二○二四年四月

郑 重 声 明

本人呈交的设计报告，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，本设计报告不包含他人享有著作权的内容。对本设计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。本设计报告的知识产权归属于培养单位。

本人签名: _____ 日期: _____

摘　　要

本实验中，我们需要完成的是基于 RISC-V 语言及其架构完成流水线 CPU 以及单周期 CPU 的设计以及实现，使得其能够支持除几条特权指令之外的 37 条简单指令集。并且在流水线的 CPU 设计中需要能够在 RISC-V 的五级流水线[1]下正常处理数据冒险，控制冒险并正常执行指令。还需要将用于测试的汇编代码综合并通过 Vivado 软件导入到 Nexys A7 实验板上，完成基于该 CPU 的简单应用程序设计。

本实验中的流水线 CPU 以及单周期 CPU 主要参考的是《计算机组成与设计：软件/硬件接口 RISC-V（第五版）》[2]中流水线架构中组成 CPU 的各个模块的相关输入输出端口、信号功能以及 RISC-V 中各条指令执行的方式和流程。最后的设计成功与否主要取决于仿真的结果是否正确以及最终在实验开发板上的数码管中显示的结果是否正确。

本次实验的主要内容主要包括：根据给定的 SCPU 各个模块的文件结构的基础上修改并且拓展各个模块的功能，使得其能够正确执行 RISC-V 的精简指令集，包括 { add,sub,and,or,jal,jalr,ori,xor,andi,addi,sll,sra,srl,slt,sltu,srai,slti,sltiu,slli, srli,lui,lw,lb,lh,lbu,lhu,sw,sb,sh,auipc,beq,bne,blt,bge,bltu,bgeu} 指令，并且本次实验中在内存中存储数据的方式为小端存储。在流水线时，引入流水线寄存器并且使用旁路前递单元、冒险检测单元来处理可能出现的控制冒险和数据冒险。仿真时使用老师已经给定的 dat 指令文件和自主通过 venus 网站生成的机器码在 vivado 2018.3 环境中进行仿真。导入实验板时借助 vivado 软件中的 IP 核来对各大模块做综合连接，并生成设备可以读取的比特流，在实验板上通过显示的内容来判断

使用的情况。

本次实验中产生的仿真波形与 Venus 中模拟的结果相同，并且在实验板上的应用中能够成功解决数据冒险，控制冒险，实验板也能正确显示代码预期的执行结果。

关键词: 流水线 CPU；单周期 CPU；FPGA;RISC-V 架构

目　录

1 引言

1.1 实验目的

首先，本课程的开设是基于前一个学期所学的《计算机组成与设计》这门课所学的知识，实验目的是融汇冠荣计算机组组成与设计这门课所教授的关于处理器的知识，通过对这些知识的总和应用，将其用 verilog HDL 语言实现出来，以达到加深对于 CPU 各个组成模块的工作原理和相互联系关系的认识与理解。

此外，本实验还旨在让我们了解包括硬件描述语言 Verilog HDL，vivado 开发套件以及 Xilinx Nexys A7 芯片在内的 EDA（Electronic Design Automation）、FPGA（Field Programmable Gate Array）技术来实现设计 RISC-V 单周期 CPU 和流水线 CPU 的方法。

最后，完成基于 RISC-V 的 CPU 设计，使其能够支持较为简单的 37 条指令（不含三条特殊指令），并且在五级流水线下能够正常处理每一条进入流水线的指令，避免数据冒险、控制冒险、结构冒险的发生。以及通过 vivado 软件将相关的导入到 FPGA 开发板上并完成基于 RISC-V 架构 CPU 的简单应用程序实现。

1.2 国内外研究现状

RISC-V 起源于 2010 年加州大学伯克利分校的一个项目，旨在开发一种新的指令集架构（ISA）。该项目由伯克利大学的安德鲁 · 沃特曼、尹素普 · 李、大卫 · 帕特森和克里斯特 · 阿萨诺维奇共同进行。RISC-V 区别于其他学术设计，它不仅为了简单的表述而优化，设计者的初衷是使 RISC-V 指令集能够用于实际计算机，并且希能够开辟一条有区别与传统架构 x86 或 ARM 复杂繁冗的道路，并且区别于 ARM 和 x86 专利和授权问题，RISC-V 被设计为一个开源指令集。

目前，已经有多种基于 RISC-V 的处理器[3]已经投入使用或者正在研发过程中，主要有处理器家族——Western Digital SweRV Core[4]，有多扩展功能的 RISC-V 处理器——AndesCore，以及包括 Nvidia Grace CPU、SiFive Core IP 和标量处

理器 Rocket 等等处理器。因此，RISC-V 架构在处理器方面在将来还有很高的发展前景。

目前市场主导的处理器架构还是 intel、AMD 以及 apple 基于 x86 和 ARM 的 CPU[5]。虽然在目前的国内外市场中，RISC-V 商用处理器的存在感还相对较小，但其开放性、灵活性、简洁性正在吸引越来越多的公司和开发者。同样，目前国产的 CPU 相比国外先进水平任存在一定的差距，从主频方面来看的话，国产 CPU 龙芯 3A6000 最高主频可达到 2.5GHz，而 Intel 公司的酷睿 13 代处理器已经达到了 6GHz，AMD 公司的 Ryzen 7 1800X 处理器的主频也已经达到了 4GHz。

无论是基于 RISC-V，x86 还是 ARM 架构的处理器最早都在外国研发问世，国内在处理器关键技术上因此产生了一定的落后。但当前时代继承电路飞速发展，基于开源的 RISC-V 处理器的研究[6]与创新才是我国能够缩小与西方国家芯片差距甚至是追上西方国家技术的关键。

2 实验环境介绍

2.1 Verilog HDL

Verilog HDL（Hardware Description Language）是由 Gateway 设计自动化公司的工程师于 1983 年末创立的。它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。

Verilog HDL 语言可以用于描述设计电路的行为特性、数据流特性、组成结构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，也可以用于实现数字电路的仿真、时序分析、逻辑综合。

2.2 Venus

Venus 是由加州大学伯克利分校开发的主要用于 CS 61C 课程（Great ideas in Computer Architecture）的软件工具。该工具是一个用于 RISC-V 汇编语言的编辑器和模拟器，为编写、运行和调试程序提供了一个强大的平台。它可以将 RISV-V 指令转换为机器码、单步调试 RISC-V 指令并实时查看寄存器和内存的值、运行 RISC-V 指令并得到结果的功能。

使用 Venus 的相关网站：`https://venus.kvakil.me/`和`https://venus.cs61c.org/`。

2.3 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，要用于 FPGA 和 SoC（系统级芯片）的设计和开发。其主要的功能有从设计输入到合成、布局与布线以及验证和仿真的完整设计流程等等。它能够支持 Xilinx 的 7 系列以及所有更新的设备，包括本实验中所使用的 Nexys A7 设备。

此外，Vivado 提供了全面的仿真和调试工具套件。Vivado 仿真器允许在不同的抽象级别进行详细的设计验证，包括行为、功能和时序仿真。对于调试，Vivado 包括逻辑分析器和虚拟 I/O 等功能，能够更好的将 FPGA 内的运行结果可视化。

本实验中使用的版本为：Vivado 2018.3

2.4 Nexys A7

NEXYS A7 开发板是基于 Xilinx Artix-7™ FPGA 的数字电路开发平台，适用于从基础组合电路到复杂的嵌入式处理器的各种设计。这块开发板配备了大容量的 FPGA、丰富的外部存储器以及 USB、以太网等多种端口，使其成为一个完整且使用灵活的开发工具，适用于处理从简单的逻辑电路到复杂的数字电路系统的各类项目。

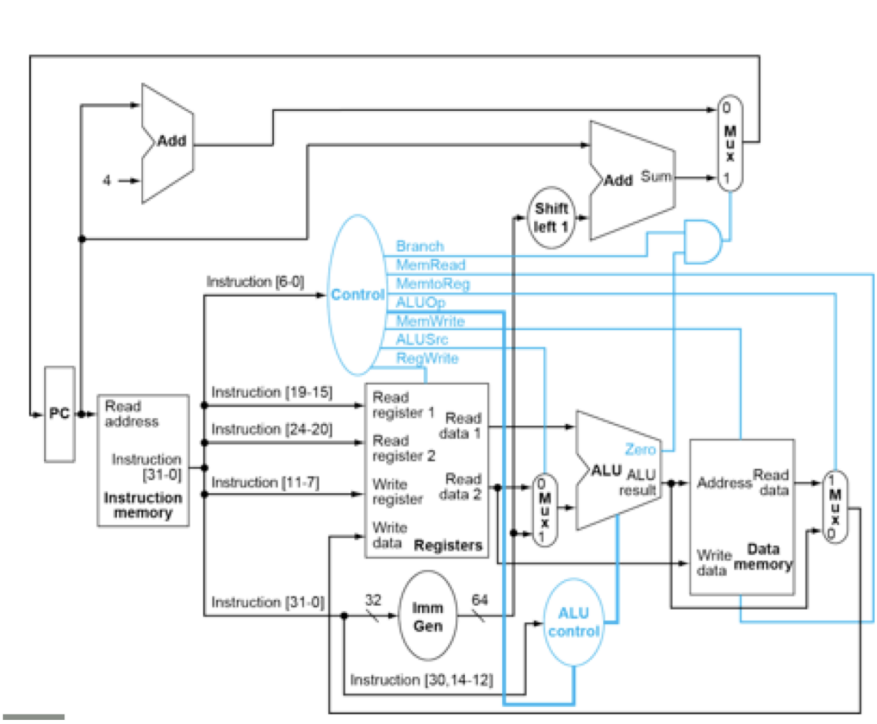
本实验中使用的设备：Artix-7 FPGA（型号：XC7A100T-1CSG324C）

3　单周期 CPU 概要设计

3.1　总体设计

单周期 CPU 由 ctrl（控制器模块）、PC（程序计数器）、NPC（下一指令地址模块）、EXT（符号扩展模块）、RF（寄存器堆模块）、ALU（运算器模块）、DM(数据存储器模块)、IM（指令存储器模块）等模块组成。其中，PC、NPC、EXT、RF、ALU、Ctrl 模块在 SCPU 模块中被例化并且被连接起来；DM 和 IM 作为两个存储器，独立于前文所属的模块在 sccomp 顶层模块中被实例化并和 scpu 模块进行连接，形成一个完整的 cpu 整体。所有模块单独作为一个 verilog HDL 语言的文件来实现。以下是单周期 CPU 的大致原理图：

图 3.1　单周期 CPU 原理图



还需补充的是，与单周期 CPU 所有相关的宏定义信息保存于 ctrl_encode_def.v 文件中；测试文件和与仿真相关的初始化信息，pc 信息存放于 sccomp_tb.v 文件

中，用于进行仿真时的调试和可视化控制。

3.2 控制模块（Ctrl）

3.2.1 功能描述

根据输入的指令，对其进行解码，生成其他模块需要的输入信号，包括使能信号，选择信号以及读写模式信号等等。

3.2.2 模块接口

Op[6:0]	Input	输入指令的 [6:0] 位，指令中的 opcode 字段
Funct7[6:0]	Input	输入指令的 [31:25] 位，指令的 funct7 字段
Funct3[2:0]	Input	输入指令的 [14:12] 位，指令的 funct3 字段
Zero	Input	运算单元 alu 是否输出为 0 的信号
RegWrite	Output	写使能信号，判断是否可以写寄存器
MemWrite	Output	写使能信号，判断是否可以写数据存储器
EXTOp[5:0]	Output	立即数扩展运算信号
ALUOp[4:0]	Output	运算单元 alu 运算控制信号
NPCOp[2:0]	Output	下一指令地址模块控制信号
ALUSrc	Output	运算单元 alu 操作数选择信号
DMType[2:0]	Output	数据存储器读写模式控制信息
GPRSel[1:0]	Output	备用端口，没有特殊功能
WDSel[1:0]	Output	写回寄存器的数据选择信号

表 3.1 Ctrl 模块接口

3.3 PC（程序计数器）

3.3.1 功能描述

根据输入信号（包括下一周期地址数据和控制信号）输出程序在该周期需要执行的指令地址，并向 IM（指令存储器模块）输送这个地址。该输出的 PC 值在每一个周期中通过 NPC 模块的输出 NPC 来进行更新。

3.3.2 模块接口

信号名	方向	描述
clk	Input	时钟信号
rst	Input	重置复位信号
NPC[31:0]	Input	来自 NPC 的模块控，控制下一周期 PC 的输入
PC[31:0]	Output	输出的当前周期指令的 PC 值

表 3.2 PC（程序计数器）模块接口表

3.4 NPC（下一指令地址）

3.4.1 功能描述

根据输入的信号，包括当前指令 PC、下一周指令地址选择信号 NPCOp 以及立即数和来自 ALU 模块的运算结果来计算得到下一周期需要执行的指令 PC，并在下一周期时送往程序计数器模块。

3.4.2 模块接口

信号名	方向	描述
PC[31:0]	Input	当前正在执行的指令地址
NPCOp[2:0]	Input	选择下一条指令地址的控制信号
IMM[31:0]	Input	来自 EXT 模块的立即数
Aluout[31:0]	Input	当前 alu 运算器的运算结果
NPC[31:0]	Output	下一时钟周期的指令地址

表 3.3 NPC（下一周期指令地址）模块接口

3.5 EXT（符号扩展模块）

3.5.1 功能描述

根据输入的指令各个立即数字段的值和立即数生成控制信号 EXTOp 来生成当前指令所需要的立即数并输出到运算器模块或 NPC 模块。

3.5.2 模块接口

信号名	方向	描述
Iimm_shamt[4:0]	Input	指令中第 [24:20] 位，用于三条移位指令
Iimm[11:0]	Input	指令中第 [31:20] 位，用于 I 型指令
Simm[11:0]	Input	指令中的 [31:25,11:7] 位，用于 S 型指令
Bimm[11:0]	Input	当指令中的 [31,7,30:25,11:8] 位，用于 SB 型指令
Uimm[19:0]	Input	指令中的 [31:12] 位，用于 U 型指令（即 lui）
Jimm[19:0]	Input	指令中的 [31,19:12,20,30:21] 位，用于 J 型指令（即 jal 指令）
EXTOp[5:0]	Input	来自 Ctrl 模块，控制生成哪一种立即数
Immout[31:0]	Output	指 EXT 符号扩展模块的输出结果

表 3.4 EXT（符号/立即数扩展）模块接口

3.6 RF（寄存器模块）

3.6.1 功能描述

控制寄存器堆中 32 个寄存器的读和写操作。对于读操作而言，无论是时钟周期的上升沿还是下降沿都可以通过两个输入端 A1 和 A2 来读取两个输入端所表示的指定编号的寄存器，并且将读取出来的值赋给 RD1 和 RD2 两个输出信号。

对于写操作而言，当时钟周期处于上升沿并且寄存器堆写使能信号（RFWr）有效时，才能进行写操作，由输入端 A3 的值决定将要写入的寄存器编号，并将输入端 WD 所输入的值写入该寄存器中。

3.6.2 模块接口

信号名	方向	描述
clk	Input	时钟信号
rst	Input	复位重置信号
RFWr	Input	寄存器堆写使能信号
A1[4:0]	Input	读寄存器的第一个寄存器编号
A2[4:0]	Input	读寄存器的第二个寄存器编号
A3[4:0]	Input	指写寄存器的编号
WD[31:0]	Input	需要写入寄存器的信号
RD1[31:0]	Output	第一个读寄存器的读出数据
RD2[31:0]	Output	第二个读寄存器的读出数据

表 3.5 RF（寄存器堆）模块接口

3.7 ALU（运算器模块）

3.7.1 功能描述

根据输入的信息，包括两个操作数，操作控制信号 ALUOp，和当前 PC 值来计算得到结果并且输出供其他模块使用。

3.7.2 模块接口

信号名	方向	描述
Signed A[31:0]	Input	第一个操作数，是有符号数
Signed B[31:0]	Input	第二个操作数，是有符号数
ALUOp[4:0]	Input	控制运算类型的运算控制信号
Signed C[31:0]	Input	运算后的结果，是有符号数
Zero	Output	判断运算结果是否为 0
PC[31:0]	Output	当前的 PC 信号，实现 AUIPC 指令的执行

表 3.6 ALU（运算器）模块接口

3.8 DM（数据存储器）

3.8.1 功能描述

根据输入的信号和写入模式信号，包括写入地址，写入数据等等，将相应内存地址内的数据写入或读出。对于读操作而言，同上述的 RF 模块，无论是时钟的上升沿还是下降沿都可以从数据存储器中读出数据。根据需要读出数据的类型 DMType 来决定是读一个字节，两个字节（半字）还是四个字节（一个字）。

而对于写信号而言，同样与 RF 模块相类似，只能在时钟的上升沿并且当写使能信号（DMWr）有效时才能进行写入。也是需要根据写入数据的类型 DMType 来决定往数据存储器中写入多少量的数据。

此外，还需要说明的是，数据在内存中采取的是小端存储方式，即是将数据的高位保存在高地址段中。还使用了按字节编址的方式，即每一个内存单元都是 8bits（一个字节），共设置了 128 个内存单元（字节）的空间。

3.8.2 模块接口

信号名	方向	描述
clk	Input	时钟信号
DMWr	Input	第二个操作数，是有符号数
addr[8:0]	Input	访问的内存地址（可访问 512 个内存单元）
din[31:0]	Input	需要写入数据存储器的数据
DMType[2:0]	Input	控制写入读出模式的控制信号
dout[31:0]	Output	从访问地址中读出的数据

表 3.7 DM（数据存储器）模块接口

3.9 IM（指令存储器）

3.9.1 功能描述

根据输入的地址信息从指令存储器中读出将要执行的指令并送往其他模块执行。由于在具体的实验板上实现时采用的是 vivado 内置的 Distributed Memory Generator IP 核来实现指令存储器，故没有该部分，该部分仅在仿真时采用。

3.9.2 模块接口

信号名	方向	描述
addr[8:2]	Input	需要读取的指令地址（NPC 模块中均使用的是字地址（例如 PC+4），但在该模块中指令是按照每次 +1 的顺序依次存放的，因此要对输入的 PC 值除以 4 以转化为指令存储器中指令真正存放的地址）
dout[31:0]	Output	从指令寄存器中取出的指令

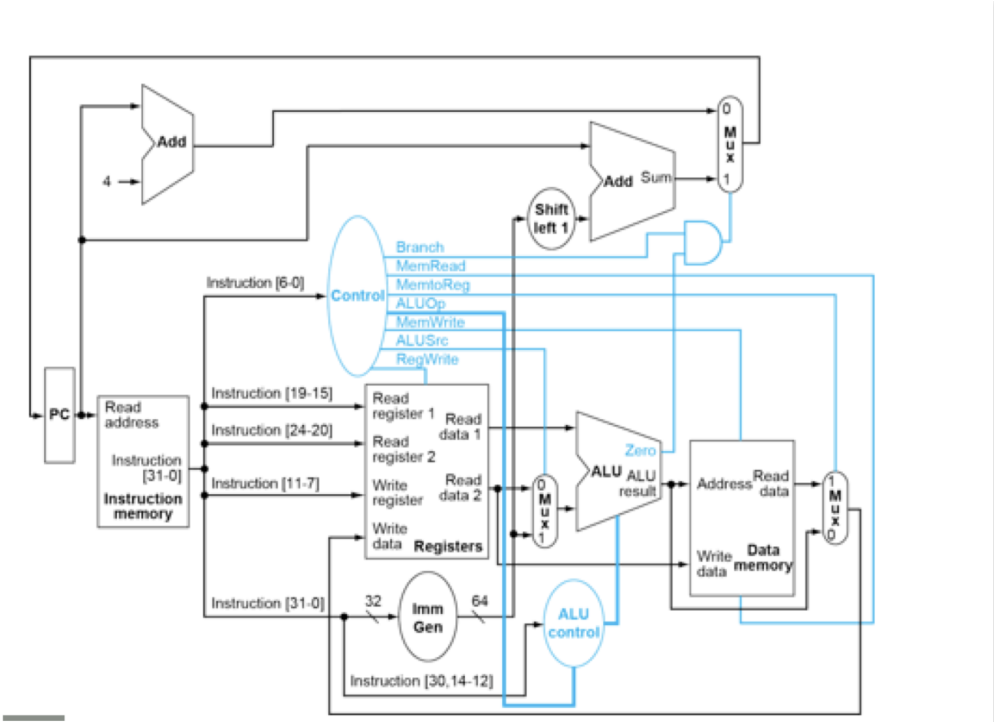
表 3.8 IM（指令存储器）模块接口

4　单周期 CPU 详细设计

4.1　CPU 总体结构

依据上学期所学的计算机组成与设计该门课的教材《计算机组成与设计:软/硬件接口 RISC-V（第五版）》中所描述的单周期 CPU 的原理图对于单周期 CPU 各个模块的输入，输出端口进行设计，包括各个指令所需要的控制信号，读写信号，功能信号等等。实验中的所有线路连接均是基于这个原理图来实现:

图 4.1　单周期 CPU 原理图



本实验中将 PC、NPC、EXT、RF、ALU、Ctrl 模块在 SCPU.v 文件中进行连接，然后再通过 sccomp.v 顶层模块文件将 SCPU 模块与 DM 和 IM 模块进行连接，实现完整的 CPU 结构。此处将 IM 与 DM 与其他模块分开独立例化放置的原因是因为在仿真时所使用的数据存储器和指令存储器分别是 DM 和 IM 模块，但是在 Nexys A7 实验板上具体实现时使用的是 vivado 内置的 block memory

generator 和 distributed memory generator IP 核来实现，因此分开放置这两个模块对于后续的导入来说更加方便。SCPU 的设计与实现如下：

在后续会依次附上在组成单周期 CPU 的各个模块的代码实现，需要注意的是，由于 vivado 套件暂不支持中文，使用中文注释在运行时会变为乱码，因此为保障提交的代码与本文中附上的代码的一致性，后续的注释均为英文注释。

```
1   `include "ctrl_encode_def.v"
2  module SCPU(
3      input     clk,           // clock
4      input     reset,         // reset
5      input [31:0] inst_in,    // instruction
6      input [31:0] Data_in,    // data from data memory
7      output       mem_w,     // output: memory write signal
8      output [31:0] PC_out,  // PC address
9        // memory write
10     output [31:0] Addr_out, // ALU output
11     output [31:0] Data_out,// data to data memory
12     input [4:0] reg_sel,   // register selection (for debug use)
13     output [31:0] reg_data, // selected register data (for debug use
          )
14     output [2:0] DMType
15 );
16     wire       RegWrite;   // control signal to register write
17     wire [5:0] EXTOp;      // control signal to signed extension
18     wire [4:0] ALUOp;      // ALU opertion
19     wire [2:0] NPCOp;      // next PC operation
20     wire [1:0] WDSel;      // (register) write data selection
21     wire [1:0] GPRSel;     // general purpose register selection
22     wire       ALUSrc;     // ALU source for A
23     wire       Zero;       // ALU ouput zero
24     wire [31:0] NPC;       // next PC
25     wire [4:0] rs1;        // rs
26     wire [4:0] rs2;        // rt
27     wire [4:0] rd;         // rd
28     wire [6:0] Op;         // opcode
29     wire [6:0] Funct7;     // funct7
30     wire [2:0] Funct3;     // funct3
31     wire [11:0] Imm12;     // 12-bit immediate
32     wire [31:0] Imm32;     // 32-bit immediate
33     wire [19:0] IMM;       // 20-bit immediate (address)
34     wire [4:0] A3;         // register address for write
35     reg [31:0] WD;         // register write data
36     wire [31:0] RD1,RD2;  // register data specified by rs
37     wire [31:0] B;         // operator for ALU B
38     wire [4:0] iimm_shamt;
39     wire [11:0] iimm,simm,bimm;
```

```
40      wire [19:0] uimm,jimm;
41      wire [31:0] immout;
42      wire [31:0] aluout;
43  assign Addr_out=aluout;
44  //the address for DM module to read/write from data memory
45  assign B = (ALUSrc) ? immout : RD2;
46  //decide the B operation is immediate or the data read from
        register
47  assign Data_out = RD2;
48   //the data for DM module to be saved in data memory
49      assign iimm_shamt =inst_in[24:20];
50      assign iimm     =inst_in[31:20];
51      assign simm     ={inst_in[31:25],inst_in[11:7]};
52      assign bimm ={inst_in[31],inst_in[7],inst_in[30:25],inst_in
            [11:8]};
53      assign uimm     =inst_in[31:12];
54      assign jimm={inst_in[31],inst_in[19:12],inst_in[20],inst_in
            [30:21]};
55      assign Op   = inst_in[6:0]; // instruction
56      assign Funct7 = inst_in[31:25]; // funct7
57      assign Funct3 = inst_in[14:12]; // funct3
58      assign rs1  = inst_in[19:15]; // rs1
59      assign rs2  = inst_in[24:20]; // rs2
60      assign rd   = inst_in[11:7]; // rd
61      assign Imm12 = inst_in[31:20];// 12-bit immediate
62      assign IMM  = inst_in[31:12]; // 20-bit immediate
63     // instantiation of control unit
64      ctrl U_ctrl(
65          .Op(Op), .Funct7(Funct7), .Funct3(Funct3), .Zero(Zero),
66          .RegWrite(RegWrite), .MemWrite(mem_w),
67          .EXTOp(EXTOp), .ALUOp(ALUOp), .NPCOp(NPCOp),
68          .ALUSrc(ALUSrc), .GPRSel(GPRSel), .WDSel(WDSel)
69          ,.DMType(DMType)
70      );
71   // instantiation of pc unit
72      PC U_PC(.clk(clk), .rst(reset), .NPC(NPC), .PC(PC_out) );
73      //instantiation of the NPC module
74      NPC U_NPC(.PC(PC_out), .NPCOp(NPCOp), .IMM(immout), .NPC(NPC), .
            aluout(aluout));
75      //instantiation of the EXT module
76      EXT U_EXT(
77          .iimm_shamt(iimm_shamt), .iimm(iimm),.simm(simm),.bimm(bimm)
                ,.uimm(uimm), .jimm(jimm),.EXTOp(EXTOp), .immout(immout)
78      );
79      //instantiation of the RF module
80      RF U_RF(
81          .clk(clk), .rst(reset),.RFWr(RegWrite), .A1(rs1), .A2(rs2), .
```

```
            A3(rd), .WD(WD), .RD1(RD1), .RD2(RD2)//.reg_sel(reg_sel)
            ,//.reg_data(reg_data)
82      );
83  // instantiation of alu unit
84      alu U_alu(.A(RD1), .B(B), .ALUOp(ALUOp), .C(aluout), .Zero(Zero),
            .PC(PC_out));
85  //please connnect the CPU by yourself
86  always @*
87  begin
88      case(WDSel)
89          `WDSel_FromALU: WD<=aluout;
90          `WDSel_FromMEM: WD<=Data_in;
91          `WDSel_FromPC: WD<=PC_out+4;
92      endcase
93  end
94  endmodule
```

4.2 Ctrl（控制模块）

```
1   module ctrl(Op, Funct7, Funct3, Zero,
2               RegWrite, MemWrite,
3               EXTOp, ALUOp, NPCOp,
4               ALUSrc, GPRSel, WDSel,DMType
5               );
6     input [6:0] Op;       // opcode
7     input [6:0] Funct7;   // funct7
8     input [2:0] Funct3;   // funct3
9     input       Zero;     //whether the result of alu is 0
10    output      RegWrite; // control signal for register write
11    output      MemWrite; // control signal for memory write
12    output [5:0] EXTOp; // control signal to signed extension
13    output [4:0] ALUOp; // ALU opertion
14    output [2:0] NPCOp; // next pc operation
15    output      ALUSrc;
16  // ALU source for B (whether it is from EXT module or from the
      register)
17    output [2:0] DMType;
18  // dm(read or write)operation ,choose read/write 1bytes/2bytes/4
      bytes
19    output [1:0] GPRSel; // general purpose register selection
20    output [1:0] WDSel;
21  // (register) write data selection,decide which data is to be
      writen to the register
22    // r format
23    wire rtype = ~Op[6]&Op[5]&Op[4]&~Op[3]&~Op[2]&Op[1]&Op[0]; //
        0110011
```

```
24    wire i_add = rtype& ~Funct7[6]&~Funct7[5]&~Funct7[4]&~Funct7[3]&~
          Funct7[2]&~Funct7[1]&~Funct7[0]&~Funct3[2]&~Funct3[1]&~Funct3
          [0]; // add 0000000 000
25    wire i_sub = rtype& ~Funct7[6]& Funct7[5]&~Funct7[4]&~Funct7[3]&~
          Funct7[2]&~Funct7[1]&~Funct7[0]&~Funct3[2]&~Funct3[1]&~Funct3
          [0]; // sub 0100000 000
26    wire i_or = rtype& ~Funct7[6]&~Funct7[5]&~Funct7[4]&~Funct7[3]&~
          Funct7[2]&~Funct7[1]&~Funct7[0]& Funct3[2]& Funct3[1]&~Funct3
          [0]; // or 0000000 110
27    wire i_and = rtype& ~Funct7[6]&~Funct7[5]&~Funct7[4]&~Funct7[3]&~
          Funct7[2]&~Funct7[1]&~Funct7[0]& Funct3[2]& Funct3[1]& Funct3
          [0]; // and 0000000 111
28    wire i_sll = rtype& ~Funct7[6]&~Funct7[5]&~Funct7[4]&~Funct7[3]&~
          Funct7[2]&~Funct7[1]&~Funct7[0]&~Funct3[2]&~Funct3[1]& Funct3
          [0]; // sll 0000000 001
29    wire i_slt = rtype& ~Funct7[6]&~Funct7[5]&~Funct7[4]&~Funct7[3]&~
          Funct7[2]&~Funct7[1]&~Funct7[0]&~Funct3[2]& Funct3[1]&~Funct3
          [0]; // slt 0000000 010
30    wire i_sltu= rtype& ~Funct7[6]&~Funct7[5]&~Funct7[4]&~Funct7[3]&~
          Funct7[2]&~Funct7[1]&~Funct7[0]&~Funct3[2]& Funct3[1]& Funct3
          [0]; // sltu 0000000 011
31    wire i_xor = rtype& ~Funct7[6]&~Funct7[5]&~Funct7[4]&~Funct7[3]&~
          Funct7[2]&~Funct7[1]&~Funct7[0]& Funct3[2]&~Funct3[1]&~Funct3
          [0]; // xor 0000000 100
32    wire i_srl = rtype& ~Funct7[6]&~Funct7[5]&~Funct7[4]&~Funct7[3]&~
          Funct7[2]&~Funct7[1]&~Funct7[0]& Funct3[2]&~Funct3[1]& Funct3
          [0]; // srl 0000000 101
33    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
34   // i format
35    wire itype_l = ~Op[6]&~Op[5]&~Op[4]&~Op[3]&~Op[2]&Op[1]&Op[0]; //
          0000011
36    wire i_lb = itype_l&~Funct3[2]&~Funct3[1]&~Funct3[0]; //lb 000
37    wire i_lh = itype_l&~Funct3[2]&~Funct3[1]& Funct3[0]; //lh 001
38    wire i_lw = itype_l&~Funct3[2]& Funct3[1]&~Funct3[0]; //lw 010
39    wire i_lbu = itype_l& Funct3[2]&~Funct3[1]&~Funct3[0]; //lbu 100
40    wire i_lhu = itype_l& Funct3[2]&~Funct3[1]& Funct3[0]; //lhu 101
41  // i format
42    wire itype_r = ~Op[6]&~Op[5]&Op[4]&~Op[3]&~Op[2]&Op[1]&Op[0]; //
          0010011
43    wire i_addi = itype_r&~Funct3[2]&~Funct3[1]&~Funct3[0]; // addi
          000
44    wire i_ori  = itype_r& Funct3[2]& Funct3[1]&~Funct3[0]; // ori 110
45    wire i_andi = itype_r& Funct3[2]& Funct3[1]& Funct3[0]; //andi 111
46    wire i_xori = itype_r& Funct3[2]&~Funct3[1]&~Funct3[0]; //xori 100
47    wire i_slti = itype_r&~Funct3[2]& Funct3[1]&~Funct3[0]; //slti 010
```

```
48   wire i_sltiu = itype_r&~Funct3[2]& Funct3[1]& Funct3[0]; //
        sltiu011
49   wire i_slli = itype_r&~Funct3[2]&~Funct3[1]& Funct3[0]; //slli 001
50   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
51   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
52  //jalr
53   wire i_jalr =Op[6]&Op[5]&~Op[4]&~Op[3]&Op[2]&Op[1]&Op[0];//jalr
        1100111
54   // s format
55   wire stype = ~Op[6]&Op[5]&~Op[4]&~Op[3]&~Op[2]&Op[1]&Op[0];//
        0100011
56   wire i_sw = stype&~Funct3[2]& Funct3[1]&~Funct3[0]; // sw 010
57   wire i_sh = stype&~Funct3[2]&~Funct3[1]& Funct3[0]; // sh 001
58   wire i_sb = stype&~Funct3[2]&~Funct3[1]&~Funct3[0]; // sb 000
59   // sb format
60    wire sbtype = Op[6]&Op[5]&~Op[4]&~Op[3]&~Op[2]&Op[1]&Op[0];//
         1100011
61    wire i_beq = sbtype&~Funct3[2]&~Funct3[1]&~Funct3[0]; // beq
62    wire i_bne = sbtype&~Funct3[2]&~Funct3[1]& Funct3[0]; // bne
63    wire i_blt = sbtype& Funct3[2]&~Funct3[1]&~Funct3[0]; // beq
64    wire i_bge = sbtype& Funct3[2]&~Funct3[1]& Funct3[0]; // beq
65    wire i_bltu = sbtype& Funct3[2]& Funct3[1]&~Funct3[0]; // beq
66    wire i_bgeu = sbtype& Funct3[2]& Funct3[1]& Funct3[0]; // beq
67  // j format
68    wire i_jal = Op[6]& Op[5]&~Op[4]& Op[3]& Op[2]& Op[1]& Op[0]; //
         jal 1101111
69   wire i_auipc = ~Op[6]&~Op[5]&Op[4]&~Op[3]&Op[2]&Op[1]&Op[0];
70   wire i_lui = ~Op[6]&Op[5]&Op[4]&~Op[3]&Op[2]&Op[1]&Op[0];
71   // generate control signals
72  assign RegWrite = rtype | itype_l | itype_r | i_jalr | i_jal |
       i_lui | i_auipc; // register write
73   assign MemWrite = stype;                          // memory write
74   assign ALUSrc   = itype_l | itype_r | stype | i_jal | i_jalr |
        i_auipc | i_lui; // ALU B is from instruction immediate
75   // signed extension
76   assign EXTOp[5] = i_slli | i_srli | i_srai;
77   assign EXTOp[4] = itype_l | i_addi | i_slti | i_sltiu | i_xori |
        i_ori | i_andi | i_jalr;
78   assign EXTOp[3] = stype;
79   assign EXTOp[2] = sbtype;
80   assign EXTOp[1] = i_auipc|i_lui;
81   assign EXTOp[0] = i_jal;
82   //dm read/write control
```

```
  assign DMType[0] = i_lb|i_lh|i_sb|i_sh;
  assign DMType[1] = i_lhu|i_lb|i_sb;
  assign DMType[2] = i_lbu;
  //select what is to be written to register
  assign WDSel[0] = itype_l;
  assign WDSel[1] = i_jal | i_jalr;
  //decide the next PC
  //when it is in the single cycle CPU,whether a SBtype instruction
      is to jump is decided in the ctrl module
  //while in the pipeline CPU,it is decided in the EX stage,NPC
     module
  assign NPCOp[0] = sbtype & Zero;
  assign NPCOp[1] = i_jal;
  assign NPCOp[2] =i_jalr;
  //decide what operation the ALU module will do
  assign ALUOp[0] = i_jal|i_jalr|itype_l|stype|i_addi|i_ori|i_add|
     i_or|i_bne|i_bge|i_bgeu|i_sltiu|i_sltu|i_slli|i_sll|i_sra|
     i_srai|i_lui;
  assign ALUOp[1] = i_jal|i_jalr|itype_l|stype|i_addi|i_add|i_and|
     i_andi|i_auipc|i_blt|i_bge|i_slt|i_slti|i_sltiu|i_sltu|i_slli|
     i_sll;
  assign ALUOp[2] = i_andi|i_and|i_ori|i_or|i_beq|i_sub|i_bne|i_blt|
     i_bge|i_xor|i_xori|i_sll|i_slli;//
  assign ALUOp[3] = i_andi|i_and|i_ori|i_or|i_bltu|i_bgeu|i_slt|
     i_slti|i_sltiu|i_sltu|i_xor|i_xori|i_sll|i_slli;
  assign ALUOp[4] = i_srl|i_sra|i_srli|i_srai;
endmodule
```

4.3 PC（程序计数器）

```
module PC( clk, rst, NPC, PC );
  input           clk;
  input           rst;
  input     [31:0] NPC;
  output reg [31:0] PC;
  always @(posedge clk, posedge rst)
    if (rst)
      PC <= 32'h0000_0000;
//      PC <= 32'h0000_3000;
    else
      PC <= NPC;
Endmodule
```

4.4 NPC（下一指令地址）

```
`include "ctrl_encode_def.v"
module NPC(PC, NPCOp, IMM, NPC,aluout); // next pc module
   input [31:0] PC;      // pc,use for AUIPC instruction
   input [2:0] NPCOp;    // next pc operation
   input [31:0] IMM;     // immediate
   input [31:0] aluout;  //result from alu module,for jalr
      instruction
   output reg [31:0] NPC; // next pc
   wire [31:0] PCPLUS4;
   assign PCPLUS4 = PC + 4; // pc + 4
   always @(*) begin
      case (NPCOp)
          `NPC_PLUS4: NPC = PCPLUS4;
          `NPC_BRANCH: NPC = PC+IMM; //for sbtype
          `NPC_JUMP:  NPC = PC+IMM; //for jal
          `NPC_JALR:  NPC = aluout; //for jalr instruction
           default:   NPC = PCPLUS4; //for the default condition
      endcase
   end // end always
endmodule
```

4.5 EXT（符号扩展模块）

```
`include "ctrl_encode_def.v"
module EXT(
input  [4:0]          iimm_shamt,
//use for SLLI,SRLI,SRAI instruction
    input  [11:0]         iimm,     //instr[31:20], 12 bits
    input  [11:0]         simm,     //instr[31:25, 11:7], 12 bits
input  [11:0]         bimm,
//instrD[31], instrD[7], instrD[30:25], instrD[11:8], 12 bits
    input  [19:0]         uimm,     //use for lui instruction
    input  [19:0]         jimm,     //use for jal instruction
input  [5:0]          EXTOp,
//to decide how to extend immediate
    output reg [31:0]     immout); //result of immediate extension
always @(*)
     case (EXTOp)
     `EXT_CTRL_ITYPE_SHAMT: immout<={27'b0,iimm_shamt[4:0]};
     `EXT_CTRL_ITYPE: immout <= {{{32-12}{iimm[11]}}, iimm[11:0]};
     `EXT_CTRL_STYPE: immout <= {{{32-12}{simm[11]}}, simm[11:0]};
     `EXT_CTRL_BTYPE: immout <= {{{32-13}{bimm[11]}}, bimm[11:0], 1'
        b0};
```

```
    `EXT_CTRL_UTYPE: immout <= {uimm[19:0], 12'b0}; //
       ???????????12??0
    `EXT_CTRL_JTYPE: immout <= {{{32-21}{jimm[19]}}, jimm[19:0], 1'
       b0};
      default:     immout <= 32'b0;
    endcase
endmodule
```

4.6 RF（寄存器模块）

```
module RF(   input        clk,
             input        rst,
             input        RFWr,
             input [4:0] A1, A2, A3,
             input [31:0] WD,
             output [31:0] RD1, RD2);
  //32 registers, each register has 32bits
  reg [31:0] rf[31:0];
  integer i;
  always @(posedge clk, posedge rst)
    if (rst) begin  // reset
      for (i=1; i<32; i=i+1)
        rf[i] <= 0; // i;
    end
    else
      if (RFWr) begin
        rf[A3] <= WD;
        //此处省略仿真时所用的所有display语句
      end
  //if it needs to read data from the number 0 register, the output
      value is 0
  assign RD1 = (A1 != 0) ? rf[A1] : 0;
  assign RD2 = (A2 != 0) ? rf[A2] : 0;
  //assign reg_data = (reg_sel != 0) ? rf[reg_sel] : 0;
endmodule
```

4.7 ALU（运算器模块）

```
include "ctrl_encode_def.v"
module alu(A, B, ALUOp, C, Zero,PC);
   input signed [31:0] A, B; //two operation number for alu module
   input        [4:0] ALUOp;//alu operation control signal, from Ctrl
       module
```

```
5     input        [31:0] PC; //the PC of the current instrcution, use
          for the AUIPC instruction
6     output signed [31:0] C; //the result of alu
7     output             Zero; //judge whether the result is 0
8     reg [31:0] C;
9     integer   i;
10    always @( * ) begin
11       case ( ALUOp )
12  `ALUOp_nop: C=A;
13  `ALUOp_lui: C=B;
14  `ALUOp_auipc:C=PC+B;
15  `ALUOp_add: C=A+B;
16  `ALUOp_sub: C=A-B;
17  `ALUOp_bne: C={31'b0,(A==B)};
18  `ALUOp_blt: C={31'b0,(A>=B)};
19  `ALUOp_bge: C={31'b0,(A<B)};
20  `ALUOp_bltu: C={31'b0,($unsigned(A)>=$unsigned(B))};
21  `ALUOp_bgeu: C={31'b0,($unsigned(A)<$unsigned(B))};
22  `ALUOp_slt: C={31'b0,(A<B)};
23  `ALUOp_sltu: C={31'b0,($unsigned(A)<$unsigned(B))};
24  `ALUOp_xor: C=A^B;
25  `ALUOp_or:  C=A|B;
26  `ALUOp_and: C=A&B;
27  `ALUOp_sll: C=A<<B;
28  `ALUOp_srl: C=A>>B;
29  `ALUOp_sra: C=A>>>B;
30       endcase
31      // Zero=(C==32'b0);
32    end // end always
33    assign Zero = (C == 32'b0);
34  endmodule
```

4.8 DM（数据存储器）

```
1   include "ctrl_encode_def.v"
2   // data memory
3   module dm(
4      input            clk,
5     input            DMWr,
6   //to decide whether write operation is permitted
7     input     [8:0]  addr,
8   //to decide where data is to be written,since the address is 9bits,
        it can only access 2^9=512 dm units
9     input     [31:0] din,  //the data to write
10    input     [2:0]  DMType,//decide write module is byte/2bytes/4
        bytes
```

```
11      output reg [31:0] dout //output of dm module
12      );
13      // declaration for data memory units, each unit contain 8bits
14      reg [7:0] dmem[127:0];
15      // so that if read/write for 1 byte       that's 1 dm unit
16      //        if read/write for 2 byte (half word) that's 2 dm unit
17      //        if read/write for 4 byte ( word ) that's 4 dm unit
18      //for the UNSIGNED/SIGNED question: when LOAD instruction SIGNED
            is the same as the UNSIGNED
19      //when SAVE instruction:
20      //while dout is 32 bits register
21      //SIGNED :if data is less than 32 bits, the left bits are filled
            with the signed bit
22      //UNSIGNED:if data is less than 32 bits, the left bits are
           filled with "0"
23      //此模块中所有用于仿真输出的display语句均省略
24      always @(posedge clk)
25         if (DMWr) begin
26           case(DMType)
27    // according to the DMTYPE to choose which save module is used
28              `dm_byte:
29                       begin
30                       dmem[addr]<=din[7:0];
31    // 1 dm unit is 8 bits,1 byte
32                       end
33              `dm_halfword:
34                       Begin
35    //1 dm unit is 8 bits, halfword need 2 continous dm units
36                       dmem[addr]<=din[7:0];
37                       dmem[addr+1]<=din[15:8];
38                       end
39              `dm_word:
40                       Begin
41    //1 dm unit is 8 bits, word need 4 continous dm units
42                       dmem[addr]<=din[7:0];
43                       dmem[addr+1]<=din[15:8];
44                       dmem[addr+2]<=din[23:16];
45                       dmem[addr+3]<=din[31:24];
46                       end
47              `dm_byte_unsigned:
48              begin//the same as before byte talk about
49              dmem[addr]<=din[7:0];
50              end
51              `dm_halfword_unsigned:
52                       begin//the same as before halfword talk about
53                       dmem[addr]<=din[7:0];
54                       dmem[addr+1]<=din[15:8];
```

```
                    end
          endcase
            //$display("dmem[0x%8X] = 0x%8X,", addr, din);
      end
   always@(*)begin
          case(DMType)
//according to DMTYPE to decide how much data is read out from dm
              `dm_byte:dout={{24{dmem[addr][7]}},dmem[addr][7:0]};
                 //   byte signed, so 24bits signal+8 bits data
              `dm_halfword:dout={{16{dmem[addr+1][7]}},dmem[addr
                 +1][7:0],dmem[addr][7:0]};
//halfword signed, so 16bits signal+16bits data
              `dm_word:dout={dmem[addr+3][7:0],dmem[addr+2][7:0],dmem
                 [addr+1][7:0],dmem[addr][7:0]};
//    word signed, so 0 bits signal+32bits data
              `dm_byte_unsigned:dout<={24'b0,dmem[addr]};
              //   byte unsigned, so 24bits 0+8 bits data
              `dm_halfword_unsigned:dout<={16'b0,dmem[addr+1],dmem[
                 addr]};
              //halfword unsigned, so 16bits 0+16bits data
                    //default:dout=32'hFFFFFFFF;
          endcase
  end
   //assign dout = dmem[addr[8:2]];
endmodule
```

4.9 IM（指令存储器）

```
dule im(input [8:2] addr,
          output [31:0] dout );
  reg [31:0] ROM[127:0];
  assign dout = ROM[addr]; // word aligned
endmodule
```

5 流水线 CPU 概要设计

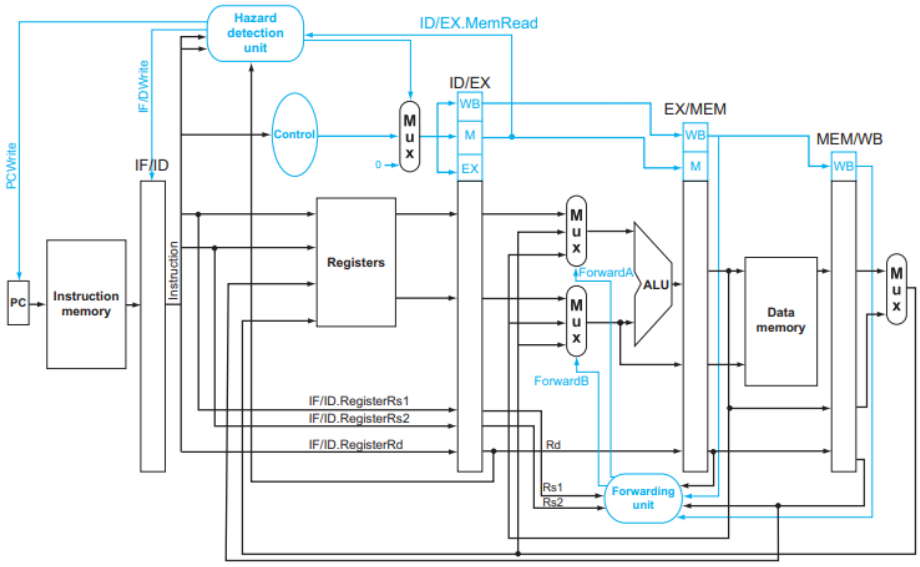
5.1 CPU 总体设计

流水线 CPU 由 PC（程序计数器）、Ctrl（控制模块）、EXT（符号扩展模块）、RF（寄存器模块）、HazardDectectionUnit（冒险检测单元）、GRE、_array（流水线寄存器模块）、ForwardingUnit（旁路前递模块）、Alu（运算器单元）、NPC（下一条指令地址模块）、DM（数据存储器模块）、IM（指令存储器模块）等部分组成。其中，将 PC、Ctrl、EXT、RF、HazardDectectionUnit、GRE_array、ForwardingUnit、ALU、NPC 模块放在 SCPU.v 中进行连线并成为一个完整的流水线 CPU。而 DM、IM 模块在仿真时独立于上述模块放在 sccomp.v 文件中，独立于 SCPU 模块进行连接。在真正实验板上实现时分别使用 vivado 套件中的 Block Memory Generator 和 Distributed Memory Generator 两个 IP 核来替代。

RISC-V 中的流水线分为五级，分别是 IF（取指令）级、ID（指令译码）级、EX（运算）级、MEM（访存）级和 WB（写回）级。每两级之间都采用一个流水线寄存器进行连接并暂时存放该级所得到的结果。但为了防止控制冒险，数据冒险和结构冒险的出现，引入了冒险探测单元、旁路前递单元和控制冒险静态预测阻塞单元。不管是仿真中还是最终实验板上实现时，有 DM、IM 模块和 IP 核的保证，不会有结构冒险的出现。

此外，同单周期 CPU 中，ctrl_encode_def.v 文件中存放着与流水线 CPU 相关的所有控制信号的宏定义。

图 5.1　流水线 CPU 大致原理图



5.2　PC（程序计数器）

5.2.1　功能描述

取指令发生在 IF 级，因此直接根据该模块生成的 PC 信息取指令，不需要从流水线寄存器中获得信息。根据输入的信号，包括当前 PC 值，将要执行的下一周期 NPC 值和是否阻塞信息 IS_STALL 来决定下一周期执行指令的 PC 地址。

5.2.2　模块接口

clk	Input	时钟信号
rst	Input	复位重置信号
IS_STALL	Input	来自冒险检测单元，判断是否阻塞的信号
NPC[31:0]	Input	下一周期需要执行的指令的 PC
PC[31:0]	Output	当前周期将要执行的指令 PC

表 5.1　流水线 CPU 中 PC（程序计数器）模块接口

5.3 Ctrl（控制模块）

5.3.1 功能描述

控制信号的生成在 ID 级完成，故所有信息都需要从 IF/ID 流水线寄存器中获得。Ctrl 模块根据从 IF/ID 流水线寄存器中读出的指令机器码信息和 PC 信息对指令进行解码。生成后续其他模块需要的控制信号。

5.3.2 模块接口

信号名	方向	描述
Op[6:0]	Input	IF/ID 流水线寄存器中存放的指令 [7:0] 位，即指令的 opcode
Funct7[6:0]	Input	IF/ID 流水线寄存器中存放的指令 [31:25] 位，即指令的 Funct7 字段
Funct3[2:0]	Input	IF/ID 流水线寄存器中存放的指令 [14:12] 位，即指令的 Funct3 字段
RegWrite	Output	IF/ID 流水线寄存器中存放指令的寄存器堆写使能信号
MemWrite	Output	IF/ID 流水线寄存器中存放指令的数据存储器写使能信号
EXTOp[5:0]	Output	IF/ID 流水线寄存器中存放指令所生成的符号扩展单元控制信号，用于生成该指令需要的立即数
ALUOp[4:0]	Output	IF/ID 流水线寄存器中存放指令需要进行的计算操作控制信号，用于在下一个时钟周期 EX 级进行该指令需要的计算
ALUSrc	Output	IF/ID 流水线寄存器中存放指令的操作数选择信号，用于在下一个时钟周期 EX 级为 ALU 选择第二个操作数
DMType[2:0]	Output	IF/ID 流水线寄存器中存放的指令需要读写数据存储器的数据类型，用于在下下个时钟周期的 MEM 级进行访存操作
GPRSel[1:0]	Output	备用端口，无特殊作用

转下一页

接上一页

信号名	方向	描述
WDsel[1:0]	Output	IF/ID 流水线寄存器中存放的指令需要写回寄存器的内容的选择控制信号，用于三个周期后的 WB 级选择写回寄存器堆的数据来源
MemRead	Output	IF/ID 流水线寄存器中存放的指令是否需要读数据存储器的信号，用于识别是否是 LOAD 类指令，用于数据冒险单元的阻塞判断

表 5.2　流水线 CPU 中 Ctrl（控制模块）的接口信息

5.4　EXT（符号扩展模块）

5.4.1　功能描述

符号扩展模块在 ID 级进行，因此一切所使用到的信息都需要从 IF/ID 流水线寄存器中获取。EXT 模块根据输入的指令各个立即数字段的值和立即数生成控制信号 EXTOp 来生成当前指令所需要的立即数并输出到运算器模块或 NPC 模块。

5.4.2　模块接口

信号名	方向	描述
Iimm_shamt[4:0]	Input	从 IF/ID 流水线中获取的指令中第 [24:20] 位，用于三条移位指令（slli，srli，srai）
Iimm[11:0]	Input	从 IF/ID 流水线中获取的指令中第 [31:20] 位，用于 I 型指令
Simm[11:0]	Input	从 IF/ID 流水线中获取的指令中的 [31:25,11:7] 位，用于 S 型指令
Bimm[11:0]	Input	从 IF/ID 流水线中获取的指令中的 [31,7,30:25,11:8] 位，用于 SB 型指令
Uimm[19:0]	Input	从 IF/ID 流水线中获取的指令中的 [31:12] 位，用于 U 型指令（即 lui 指令）

转下一页

接上一页

信号名	方向	描述
Jimm[19:0]	Input	从 IF/ID 流水线中获取的指令中的 [31,19:12,20,30:21] 位，用于 J 型指令（即 jal 指令）
EXTOp[5:0]	Input	来自 Ctrl 模块，控制生成哪一种立即数
RegWrite	Output	IF/ID 流水线寄存器中存放指令的寄存器堆写使能信号

表 5.3　流水线 CPU 中 EXT（符号/立即数扩展模块）模块接口

5.5　RF（寄存器模块）

5.5.1　功能描述

该模块中实现的是对寄存器堆中 32 个寄存器的读操作和写操作。对于读操作而言，无论是时钟周期的上升沿还是下降沿都是可以从寄存器中读出数据的。根据需要读出的寄存器编号 A1,A2 输入端来获取相应编号寄存器中的值并赋给 RD1,RD2 输出。

对于写操作而言，本实验中采用的是 RF 寄存器模块在下降沿写入，DM 和其他流水线寄存器均采取上升沿写入。因为 RF 的写入需要再 WB 级时决定写入的内容，而 WB 级的所有信息又需要从 MEM/WB 流水线寄存器中取出，因此为避免时序错误，选择流水线在上升沿写，寄存器在下降沿写来实现在上升沿先得到写入数据，在下降沿将该数据写入寄存器。通过 A3 决定写入寄存器的编号，再奖 WD 中的数据写入对应编号的寄存器。

5.5.2　模块接口

信号名	方向	描述
clk	Input	时钟信号
rst	Input	复位重置信号

转下一页

接上一页

信号名	方向	描述
RFWr	Input	MEM/WB 流水线寄存器中存储的该条指令是否写寄存器的寄存器堆写使能信号
A1[4:0]	Input	IF/ID 流水线寄存器中获得的指令读寄存器的第一个寄存器编号
A2[4:0]	Input	IF/ID 流水线寄存器中获得的指令读寄存器的第二个寄存器编号
A3[4:0]	Input	MEM/WB 流水线寄存器中存储的该条指令写寄存器的编号
WD[31:0]	Input	WB 级所决定需要写入寄存器的数据内容
RD1[31:0]	Output	第一个读寄存器的读出数据
RD2[31:0]	Output	第二个读寄存器的读出数据

表 5.4　流水线 CPU 中 RF（寄存器堆模块）的模块接口

5.6　HazardDectectionUnit（冒险检测单元）

5.6.1　功能描述

为解决数据冒险问题，我们引入了阻塞和旁路前递的两个方法，其中，旁路前递的问题将在后文旁路前递模块中讨论。冒险检测单元解决的问题是一条 LOAD 的指令要写回的寄存器与其后一条指令要读的寄存器为同一个寄存器的问题。为了让后一条指令读的数据就是 I 型指令写入的数据，保证数据一致性，引入 HazardDetectUnit 模块判断是否需要阻塞一个周期等待 I 型指令将数据写回寄存器。

5.6.2　模块接口

信号名	方向	描述
EX_MemRead	Input	来自 ID/EX 流水线寄存器中该指令是否需要读数据存储器的信号
ID_rs1	Input	来自 IF/ID 流水线寄存器中的需要读的第一个寄存器编号

转下一页

接上一页

信号名	方向	描述
ID_rs2	Input	来自 IF/ID 流水线寄存器中的需要读的第二个寄存器编号
EX_rd	Input	来自 ID/EX 流水线寄存器中该指令需要写回的寄存器编号
stall_signal	Output	用于判断是否阻塞的阻塞信号

表 5.5 流水线 CPU 中 HazardDetectUnit（冒险检测单元）的模块接口

5.7 GRE_array（流水线寄存器模块）

5.7.1 功能描述

流水线寄存器用于存放刚刚执行完的一级所得到的结果信息，也用于存放后续级可能要用到的控制信号，选择信号等信息。

由于要与 RF 寄存器堆实现数据的同步和一致性，因此需要先从流水线寄存器中取出信息完成 WB 级以决定写回寄存器的数据。因此流水线寄存器选择上升沿写入的方式。

5.7.2 模块接口

信号名	方向	描述
Clk	Input	时钟信号
Rst	Input	复位重置信号
write_enable	Input	写使能信号，用于决定是否允许修改流水线寄存器中的值
flush	Input	清空信号，用于决定是否要清空流水线寄存器中的所有值以阻止指令的执行
in[180:0]	Input	流水线寄存器的输入数据，包含之后的级需要的数据信息，控制信号，选择信号等等
out[180:0]	Output	流水线寄存器的输出，作为下一级执行的数据来源

表 5.6 流水线 CPU 中 GRE_array（流水线寄存器模块）的模块接口

5.8 ForwardingUnit（旁路前递模块）

5.8.1 功能描述

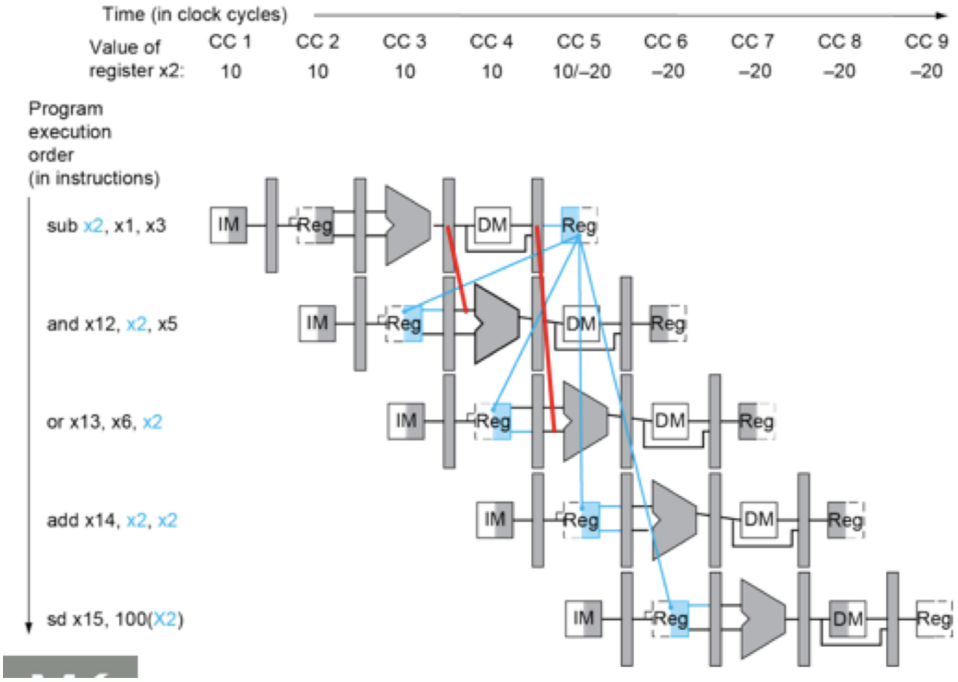
前文已经叙述过，为解决数据冒险，引入了冒险阻塞和旁路前递两种方式。旁路前递模块就是用于实现旁路前递功能。

本模块解决的问题是数据冒险中当前处在 EX 级的指令所需要的两个操作数或是其中一个操作数是其上一条指令或是上上条指令的运算结果或是需要写回寄存器的内容，因而通过建立旁路的方式将其直接送到 ALU 模块中进行运算，保证了数据的同步性。若出现以下四种情况，那么就需要进行旁路前递：

1. EX/MEM.RegisterRd = ID/EX.RegisterRs1（MEM->EX 旁路）
2. EX/MEM.RegisterRd = ID/EX.RegisterRs2（MEM->EX 旁路）
3. MEM/WB.RegisterRd = ID/EX.RegisterRs1（WB->EX 旁路）
4. MEM/WB.RegisterRd = ID/EX.RegisterRs2（WB->EX 旁路）

以下是流水线 CPU 中可能会发生的出现旁路前递的情况：

图 5.2　流水线 CPU 中可能出现旁路前递的情况



此外，还需要进行说明的是旁路前递单元的输出与 ALU 操作数来源的对应关

系如下：

ForwardSignal[1:0]的取值	数据来源	说明
00	ID/EX 流水线寄存器中读出的寄存器中的值	不进行旁路前递，操作数正常来自于 ID/EX 流水线寄存器
10	EX/MEM 流水线寄存器中存储的 ALU 运算结果	ALU 操作数从 MEM 阶段前递过来
01	WB 阶段等待写回寄存器的 WD 的值	ALU 操作数从 WB 阶段前递过来

表 5.7　流水线 CPU 中 GRE_array（流水线寄存器模块）的模块接口

5.8.2　模块接口

信号名	方向	描述
MEM_RegWrite	Input	来自 EX/MEM 流水线寄存器中该指令是否需要写寄存器的寄存器写使能信号
MEM_rd[4:0]	Input	来自 EX/MEM 流水线寄存器中该指令需要写回的寄存器编号
WB_RegWrite	Input	来自 MEM/WB 流水线寄存器中该指令是否需要写寄存器的寄存器写使能信号
WB_rd[4:0]	Input	来自 MEM/WB 流水线寄存器中该指令需要写回的寄存器编号
EX_rs[4:0]	Input	来自 ID/EX 中需要读出数据的寄存器编号
ForwardSignal[1:0]	Output	旁路前递单元输出信号，与数据来源的对应关系见上表

表 5.8　流水线 CPU 中 ForwardingUnit（旁路前递单元）的模块接口

5.9 ALU（运算器单元）

5.9.1 功能描述

ALU 在 EX 级执行，因此所有与 ALU 模块相关的操作数信息全部需要从 ID/EX 流水线寄存器中读取。ALU 根据输入的信息，包括两个操作数，操作控制信号 ALUOp，和当前 PC 值来计算得到结果并且输出供其他模块使用。

5.9.2 模块接口

信号名	方向	描述
Signed A[31:0]	Input	第一个操作数，来自旁路数据选择器 1，是有符号数
Signed B[31:0]	Input	第二个操作数，来自旁路选择器 2 的结果和立即数选选择器的结果，是有符号数
ALUOp[4:0]	Input	从 ID/EX 流水线寄存器中获取的在 ID 级生成的控制运算类型的运算控制信号
Signed C[31:0]	Input	运算后的结果，是有符号数
Zero	Output	判断运算结果是否为 0
PC[31:0]	Output	从 ID/EX 流水线寄存器中获取的当前的 PC 信号，用于实现 AUIPC 指令的执行

表 5.9 流水线 CPU 中 ALU（运算器单元）模块的模块接口

5.10 NPC（下一条指令地址模块）

5.10.1 功能描述

本实验中，决定下一周期需要执行的指令 PC 在 EX 级进行。也就是说，对于解决控制冒险的问题，判断转跳指令是否发生转跳发生在 EX 级。因此所有与 NPC 模块相关的输入信号均需要从 ID/EX 流水线寄存器中获取。该模块根据输入的信号，包括当前指令 PC（EX_PC)、下一周指令地址选择信号 NPCOp 以及立即数和来自 ALU 模块的运算结果来计算得到下一周期需要执行的指令 PC，并在下一周期时送往程序计数器模块（PC)。

5.10.2 模块接口

信号名	方向	描述
PC[31:0]	Input	从 ID/EX 流水线寄存器中获取的当前正在执行的指令地址
NPCOp[2:0]	Input	从 ID/EX 流水线寄存器中获取的在 ID 级产生选择下一条指令地址的控制信号
IMM[31:0]	Input	从 ID/EX 流水线寄存器中获取的在 ID 级产生的来自 EXT 模块的立即数
Aluout[31:0]	Input	当前 alu 运算器的运算结果
NPC[31:0]	Output	下一时钟周期的指令地址

表 5.10 流水线 CPU 中 NPC（下一周期指令 PC）模块接口

5.11 DM（数据存储器模块）

5.11.1 功能描述

对于数据存储器的访存在 MEM 级进行，因此所有跟数据存储器相关的输入信息都需要从 EX/MEM 流水线寄存器中获取。本模块根据输入的信号和写入模式信号，包括写入地址，写入数据等等，将相应内存地址内的数据写入或读出。对于读操作而言，同上述的 RF 模块，无论是时钟的上升沿还是下降沿都可以从数据存储器中读出数据。根据需要读出数据的类型 DMType 来决定是读一个字节，两个字节（半字）还是四个字节（一个字）。

而对于写信号而言，同样与 RF 模块相类似，只能在时钟的上升沿并且当写使能信号（DMWr）有效时才能进行写入。其原因是：一切与访存相关的输入端均来自 EX/MEM 流水线寄存器，不存在像 RF 模块一样的时序问题，因此可以选择上升沿写入。其写入时也是需要根据写入数据的类型 DMType 来决定往数据存储器中写入多少量的数据。

此外，还需要说明的是，数据在内存中采取的是小端存储方式，即是将数据的高位保存在高地址段中。还使用了按字节编址的方式，即每一个内存单元都是 8bits（一个字节），共设置了 512 个内存单元（字节）的空间。

在使用 Nexys A7 开发板实现时，用 vivado 内置的 Block Memory Generator

IP 核来代替该模块。

5.11.2 模块接口

信号名	方向	描述
clk	Input	时钟信号
DMWr	Input	来自 EX/MEM 流水线寄存器中存储的当前指令是否需要写寄存器的写使能信号
addr[8:0]	Input	来自 EX/MEM 流水线寄存器中存储的 EX 级的运算结果作为访问的内存地址（可访问 512 个内存单元）
DMType[2:0]	Input	来自 EX/MEM 流水线寄存器中存储的在 ID 级生成的控制写入读出模式的控制信号
dout[31:0]	Output	从访问地址中读出的数据

表 5.11 流水线 CPU 中 DM（数据存储器）模块的模块接口

5.12 IM（指令存储器模块）

5.12.1 功能描述

根据输入的地址信息从指令存储器中读出将要执行的指令并送往其他模块执行。由于在具体的实验板上实现时采用的是 vivado 内置的 Distributed Memory Generator IP 核来实现指令存储器，故没有该部分，该部分仅在仿真时采用。

5.12.2 模块接口

信号名	方向	描述
addr[8:2]	Input	根据 PC 产生的当前指令地址 PC 来读取需要读取的指令地址（NPC 模块中均使用的是字地址（例如 PC+4），但在该模块中指令是按照每次 +1 的顺序依次存放的，因此要对输入的 PC 值除以 4 以转化为指令存储器中指令真正存放的地址）
转下一页		

接上一页		
信号名	方向	描述
dout[31:0]	Output	从指令寄存器中取出的指令

表 5.12　流水线 CPU 中 IM（指令寄存器）模块的模块接口

6 流水线 CPU 详细设计

6.1 CPU 总体结构

在后续会依次附上在组成单周期 CPU 的各个模块的代码实现，需要注意的是，由于 vivado 套件暂不支持中文，使用中文注释在运行时会变为乱码，因此为保障提交的代码与本文中附上的代码的一致性，后续的注释均为英文注释。

SCPU 模块的设计将五级流水线的每一级都集中实现，其具体设计如下所示：

```
1     `include "ctrl_encode_def.v"
2  module SCPU(
3      input         clk,          // clock
4      input         reset,        // reset
5      input [31:0] inst_in,   // instruction
6      input [31:0] Data_in,   // data from data memory
7      output        mem_w,     // output: memory write signal
8      output [31:0] PC_out,  // PC address
9        // memory write
10     output [31:0] Addr_out, // ALU output
11     output [31:0] Data_out,// data to data memory
12     //input [4:0] reg_sel, // register selection (for debug use)
13    // output [31:0] reg_data, // selected register data (for debug
         use)
14     output [2:0] DMType
15 );
16     //assign Addr_out=aluout;
17         //assign B = (ALUSrc) ? immout : RD2;
18         //assign Data_out = RD2;
19     //declaration of pipeline registers IN/OUT
20     //given that the ID/EX pipeline register requires 160bits,
          larger than 128bits,so 181bits for all pipeline register IN
          and OUT
21     //declaration of relevent virables of IF/ID register
22     wire [180:0]IF_ID_IN;
23         wire [180:0]IF_ID_OUT;
24     wire        IF_ID_flush;
25     //declaration of relevent virables of ID/EX register
26     wire [180:0]ID_EX_IN;
27     wire [180:0]ID_EX_OUT;
28     wire        ID_EX_flush;
```

```
29      wire       ID_EX_writeE;
30      //declaration of relevent virables of EX/MEM register
31      wire [180:0]EX_MEM_IN;
32      wire [180:0]EX_MEM_OUT;
33      wire       EX_MEM_flush;
34      wire       EX_MEM_writeE;
35      //declaration of relevent virables of MEM/WB register
36      wire [180:0]MEM_WB_IN;
37      wire [180:0]MEM_WB_OUT;
38      wire       MEM_WB_flush;
39      wire       MEM_WB_writeE;
40      //declaration virables for ID STAGE
41      wire      stall_signal;
42      wire [31:0]ID_PC;
43      wire      ID_MemWrite;
44      wire [2:0] ID_DMType;
45      wire       RegWrite;   // control signal to register write
46      wire [5:0] EXTOp;      // control signal to signed extension
47      wire [4:0] ALUOp;      // ALU opertion
48      wire [2:0] NPCOp;      // next PC operation
49      wire [1:0] WDSel;      // (register) write data selection
50      wire [1:0] GPRSel;     // general purpose register selection
51      wire       ALUSrc;     // ALU source for A
52      wire [4:0] rs1;         // rs
53      wire [4:0] rs2;         // rt
54      wire [4:0] rd;         // rd
55      wire [6:0] Op;         // opcode
56      wire [6:0] Funct7;      // funct7
57      wire [2:0] Funct3;      // funct3
58      wire [11:0] Imm12;     // 12-bit immediate
59      wire [31:0] Imm32;     // 32-bit immediate
60      wire [19:0] IMM;       // 20-bit immediate (address)
61      wire [4:0] A3;         // register address for write
62      wire [31:0] RD1,RD2;      // register data specified by rs
63      wire       MemRead;
64          wire [4:0] iimm_shamt;
65          wire [11:0] iimm,simm,bimm;
66          wire [19:0] uimm,jimm;
67          wire [31:0] immout;
68      wire [31:0] aluout;
69      //declaration of virables for EX stage
70      wire      ID_EX_MemRead;
71      wire [4:0] ID_EX_rd;
72      wire      ID_EX_RegWrite;
73      wire      ID_EX_MemWrite;
74      wire [4:0] ID_EX_ALUOp;
75      wire [2:0] ID_EX_NPCOp;
```

```
76      wire      ID_EX_ALUSrc;
77      wire [2:0] ID_EX_DMType;
78      wire [1:0] ID_EX_WDSel;
79      wire [31:0]EX_immout;
80      wire [31:0]EX_RD1;
81      wire [31:0]EX_RD2;
82      wire [4:0] EX_rs1;
83      wire [4:0] EX_rs2;
84      wire [31:0]EX_PC;
85      wire [31:0]selResult;
86      wire      Branch_or_jump;
87      wire [1:0] FORWARD1;//=2'b0;
88      wire [1:0] FORWARD2;//=2'b0;
89      wire       Zero;       // ALU ouput zero
90      wire [31:0] NPC;       // next PC
91      wire [31:0] B;         // operator for ALU B
92      wire [31:0] A;
93      //declaration of virables for MEM stage
94      wire      MEM_RegWrite;
95      wire [4:0] MEM_rd;
96      wire [31:0]MEM_MemRead;
97      wire [1:0] MEM_WDSel;
98      wire [2:0] MEM_DMType;
99      wire [2:0] MEM_NPCOp;
100     wire [31:0]MEM_PC;
101     wire      MEM_MemWrite;
102     wire [31:0]MEM_immout;
103     wire      MEM_Zero;
104     wire [31:0]EX_MEM_aluout;
105     wire [31:0]DataToDM;
106     wire [31:0]dm_dout;
107     //declaration of virables for WB stage
108     wire [1:0] WB_WDSel;
109     wire [31:0]WB_PC;
110     wire      WB_RegWrite;
111        wire [4:0] WB_rd;
112     wire [31:0]DataToReg;
113     wire [31:0]MEM_WB_aluout;
114     reg [31:0] WD;         // register write data
115   //IF STAGE
116   //use PC to fetch the instruction for the clock cycle to execute
117   // instantiation of pc unit
118       PC U_PC(.clk(clk), .rst(reset), .NPC(NPC), .PC(PC_out),.
          IS_STALL(stall_signal) );
119   //assignment of the input of IF/ID pipeline register.
120   //PC_out represent for the current PC.
121   //inst_in represent for the current instruction.
```

```
122     assign IF_ID_IN={117'b0,PC_out[31:0],inst_in[31:0]};
123     //assignment of the IF/ID pipeline register
124     //clk&reset are the same to the input of the SCPU module
125     //about write_enable: when the pipeline is stalled,
126     //        given that virable stall_signal is decided when
            current instruction is in IF stage and the next instruction
            is in ID stage
127     //        the next instruction should be stalled, so IF/ID
            pipeline register should NOT be written
128     //about flush(ID_EX_flush): flush information is used for JUMP
            or BRANCH instruction and whether BRANCH instruction junps is
             decided in EX stage
129     //   when jump occurs, 2 instructions has entered pipeline,
            which need to be flushed
130     //   so use ID_EX_flush from EX stage to decide whether IF/ID is
             flushed
131     GRE_array IF_ID
132     (
133      .Clk(clk),
134      .Rst(reset),
135      .write_enable(~stall_signal),
136      .flush(ID_EX_flush),
137      .in(IF_ID_IN),
138      .out(IF_ID_OUT)
139     );
140     //ID STAGE
141     // instantiation of control unit
142         assign iimm_shamt=IF_ID_OUT[24:20];
143         assign iimm    =IF_ID_OUT[31:20];
144         assign simm    ={IF_ID_OUT[31:25],IF_ID_OUT[11:7]};
145         assign bimm    ={IF_ID_OUT[31],IF_ID_OUT[7],IF_ID_OUT[30:25],
                IF_ID_OUT[11:8]};
146         assign uimm    =IF_ID_OUT[31:12];
147         assign jimm    ={IF_ID_OUT[31],IF_ID_OUT[19:12],IF_ID_OUT
                [20],IF_ID_OUT[30:21]};
148      assign Op      = IF_ID_OUT[6:0]; // instruction
149      assign Funct7  = IF_ID_OUT[31:25]; // funct7
150      assign Funct3  = IF_ID_OUT[14:12]; // funct3
151      assign rs1     = IF_ID_OUT[19:15]; // rs1
152      assign rs2     = IF_ID_OUT[24:20]; // rs2
153      assign rd      = IF_ID_OUT[11:7]; // rd
154      assign Imm12   = IF_ID_OUT[31:20];// 12-bit immediate
155      assign IMM     = IF_ID_OUT[31:12]; // 20-bit immediate
156      assign ID_PC   = IF_ID_OUT[63:32];
157      //assignments of control signals
158         ctrl U_ctrl(
159                .Op(Op), .Funct7(Funct7), .Funct3(Funct3), /*.Zero(
```

```
                Zero), */
160             .RegWrite(RegWrite), .MemWrite(ID_MemWrite),
161             .EXTOp(EXTOp), .ALUOp(ALUOp), .NPCOp(NPCOp),
162             .ALUSrc(ALUSrc), .GPRSel(GPRSel), .WDSel(WDSel)
163     ,.DMType(ID_DMType),.MemRead(MemRead)
164     );
165 //assignments of immediate value
166     EXT U_EXT(
167             .iimm_shamt(iimm_shamt), .iimm(iimm), .simm(simm), .
                bimm(bimm),
168             .uimm(uimm), .jimm(jimm),
169             .EXTOp(EXTOp), .immout(immout)
170     );
171     //Module instantiation of RF
172     RF U_RF(
173             .clk(clk), .rst(reset),
174             .RFWr(WB_RegWrite),
175             .A1(rs1), .A2(rs2), .A3(WB_rd),
176             .WD(WD),
177             .RD1(RD1), .RD2(RD2)
178             //.reg_sel(reg_sel),
179             //.reg_data(reg_data)
180     );
181 //assignments of stall_signal
182 HazardDectectionUnit U_HazardDectectionUnit
183 (
184     .EX_MemRead(ID_EX_MemRead)
185     ,.ID_rs1(rs1)
186     ,.ID_rs2(rs2)
187     ,.EX_rd(ID_EX_rd)
188     ,.stall_signal(stall_signal)
189 );
190 //assignment of input of ID/EX pipeline register
191 //assign ID_EX_IN={21'b0,RegWrite,mem_w,ALUOp[4:0],NPCOp[2:0],
    ALUSrc,DMType[2:0],WDSel[1:0],MemRead,ID_PC[31:0],immout
    [31:0],RD1[31:0],RD2[31:0],rs1[4:0],rs2[4:0],rd[4:0]};
192 //if the pipeline is stalled,all control signals should be
    flushed
193 assign ID_EX_IN[142:0] ={ID_PC[31:0],immout[31:0],RD1[31:0],RD2
    [31:0],rs1[4:0],rs2[4:0],rd[4:0]};
194 assign ID_EX_IN[143] =(stall_signal)?0:MemRead;
195 assign ID_EX_IN[145:144]=(stall_signal)?0:WDSel[1:0];
196 assign ID_EX_IN[148:146]=(stall_signal)?0:ID_DMType[2:0];
197 assign ID_EX_IN[149] =(stall_signal)?0:ALUSrc;
198 assign ID_EX_IN[152:150]=(stall_signal)?0:NPCOp[2:0];
199 assign ID_EX_IN[157:153]=(stall_signal)?0:ALUOp[4:0];
200 assign ID_EX_IN[158] =(stall_signal)?0:ID_MemWrite;
```

```
201     assign ID_EX_IN[159] =(stall_signal)?0:RegWrite;
202     assign ID_EX_IN[180:160]=21'b0;
203     assign ID_EX_writeE=1;
204     //assign ID_EX_flush=0;
205     //information about ID_EX_flush has mentioned before
206     //about write_enalbe: as has mentioned before, when stall_signal
            is positive,only 1 irrelvent instruction enters pipeline
207     //only IF/ID pipeline register should not be modified,so ID/EX,
           EX/MA,MA/WB register's write_enable is 1
208     GRE_array ID_EX
209     (
210     .Clk(clk),
211     .Rst(reset),
212     .write_enable(ID_EX_writeE),
213     .flush(ID_EX_flush),
214     .in(ID_EX_IN),
215     .out(ID_EX_OUT)
216     );
217     //EX STAGE
218     //decode essential information from ID/EX register
219     // wire [4:0] EX_rd;
220     assign ID_EX_rd         =ID_EX_OUT[4:0];
221     assign EX_rs2           =ID_EX_OUT[9:5];
222     assign EX_rs1           =ID_EX_OUT[14:10];
223     assign EX_RD2           =ID_EX_OUT[46:15];
224     assign EX_RD1           =ID_EX_OUT[78:47];
225     assign EX_immout        =ID_EX_OUT[110:79];
226     assign EX_PC            =ID_EX_OUT[142:111];//for NPC module to
           decide the next PC
227     assign ID_EX_MemRead    =ID_EX_OUT[143];
228     assign ID_EX_WDSel      =ID_EX_OUT[145:144];
229     assign ID_EX_DMType     =ID_EX_OUT[148:146];
230     assign ID_EX_ALUSrc     =ID_EX_OUT[149];
231     // assign ID_EX_NPCOp[2:1] =ID_EX_OUT[152:151];
232     assign ID_EX_ALUOp     =ID_EX_OUT[157:153];
233     assign ID_EX_MemWrite  =ID_EX_OUT[158];
234     assign ID_EX_RegWrite  =ID_EX_OUT[159];
235     //about NPCOp:decide whether sbtype instructions jump
236     //NPCOp[0] is sbtype , so it depends on Zero&sbtype to deside
           whether it jummps
237     assign ID_EX_NPCOp={ID_EX_OUT[152:151],ID_EX_OUT[150] & Zero};
238     //2 forwarding unit to solve data hazard
239     //2 op number of alu can be selected from data read from RF from
            EX stage/aluout of previous instruction from MEM stage/data
           write bcak to RF from WB stage
240     ForwardingUnit ForwardA
241     (
```

```
242     .MEM_RegWrite(MEM_RegWrite)
243     ,.MEM_rd(MEM_rd)
244     ,.WB_rd(WB_rd)
245     ,.WB_RegWrite(WB_RegWrite)
246     ,.Ex_rs(EX_rs1)
247     ,.ForwardSignal(FORWARD1)
248    );
249     ForwardingUnit ForwardB
250    (
251     .MEM_RegWrite(MEM_RegWrite)
252     ,.MEM_rd(MEM_rd)
253     ,.WB_rd(WB_rd)
254     ,.WB_RegWrite(WB_RegWrite)
255     ,.Ex_rs(EX_rs2)
256     ,.ForwardSignal(FORWARD2)
257    );
258   //select 2 op number for alu
259   //for A: forwardA=00:from rf
260   //      forwardA=01:from WB stage
261   //      forwardA=10:from MEM stage
262   //for B: immidate value is selected afer forwarding unit
263   //      ALUSRC=1: immediate from EXT module
264   //      ALUSRC=0: result of forwarding unit
265   //      forwardB=00: from rf
266   //      forwardB=01: from WB stage
267   //      forwardB=10: from EX stage
268    assign A=(FORWARD1[0])?WD:((FORWARD1[1])?EX_MEM_aluout:EX_RD1);
269    assign B=(ID_EX_ALUSrc) ? EX_immout:((FORWARD2[0])?WD:((FORWARD2
          [1])?EX_MEM_aluout:EX_RD2));
270    //assign Zero=0;
271    //selResult is the result of the forwarding unit
272    //it may serve as data to be written to dm for dm module
273    assign selResult=(FORWARD2[0])?WD:((FORWARD2[1])?EX_MEM_aluout:
          EX_RD2);
274    // instantiation of alu unit
275    alu U_alu(.A(A), .B(B), .ALUOp(ID_EX_ALUOp), .C(aluout), .Zero(Zero
          ),.PC(EX_PC));
276    //decide whether to jump
277    assign Branch_or_jump= ID_EX_NPCOp[2]|ID_EX_NPCOp[1]|ID_EX_NPCOp
          [0];
278    assign ID_EX_flush= stall_signal | Branch_or_jump;
279    //decide the next pc
280    NPC U_NPC(.PC(PC_out), .NPCOp(ID_EX_NPCOp), .IMM(EX_immout), .NPC(
          NPC), .aluout(aluout),.EX_PC(EX_PC));
281    assign EX_MEM_IN={36'b0,EX_PC[31:0],ID_EX_RegWrite,ID_EX_MemWrite,
          ID_EX_NPCOp[2:0],ID_EX_DMType[2:0],ID_EX_WDSel[1:0],
          ID_EX_MemRead,ID_EX_rd[4:0],selResult[31:0],Zero,aluout[31:0],
```

```
    EX_immout[31:0]};
282  assign EX_MEM_flush =0;
283  assign EX_MEM_writeE=1;
284  GRE_array EX_MEM
285   (
286    .Clk(clk),
287    .Rst(reset),
288    .write_enable(EX_MEM_writeE),
289    .flush(EX_MEM_flush),
290    .in(EX_MEM_IN),
291    .out(EX_MEM_OUT)
292   );
293  //MEM STAGE
294  //decode information for MEM stage from EX/MEM register
295  assign MEM_immout =EX_MEM_OUT[31:0];
296  assign EX_MEM_aluout=EX_MEM_OUT[63:32];
297  assign MEM_Zero   =EX_MEM_OUT[64];
298  assign DataToDM   =EX_MEM_OUT[96:65];
299  assign MEM_rd     =EX_MEM_OUT[101:97];
300  assign MEM_MemRead =EX_MEM_OUT[102];
301  assign MEM_WDSel  =EX_MEM_OUT[104:103];
302  assign MEM_DMType =EX_MEM_OUT[107:105];
303  assign MEM_MemWrite =EX_MEM_OUT[111];
304  assign MEM_RegWrite =EX_MEM_OUT[112];
305  assign MEM_PC     =EX_MEM_OUT[144:113];
306  //prepare parameters for DM module(it is instrantiated in the top
       module)
307  assign mem_w      =MEM_MemWrite;
308  assign Data_out   =DataToDM;
309  assign DMType     =MEM_DMType;
310  assign Addr_out   =EX_MEM_aluout;
311  assign MEM_WB_IN  ={77'b0,MEM_PC[31:0],MEM_RegWrite,MEM_WDSel[1:0],
       MEM_rd[4:0],EX_MEM_aluout[31:0],Data_in[31:0]};
312  assign MEM_WB_flush =0;
313  assign MEM_WB_writeE=1;
314  GRE_array MEM_WB
315   (
316    .Clk(clk),
317    .Rst(reset),
318    .write_enable(MEM_WB_writeE),
319    .flush(MEM_WB_flush),
320    .in(MEM_WB_IN),
321    .out(MEM_WB_OUT)
322   );
323 // WB STAGE
324 //decode information for WB stage from MEM/WB pipeline register
325 assign WB_PC      =MEM_WB_OUT[103:72];
```

```
assign WB_RegWrite =MEM_WB_OUT[71];
assign WB_WDSel   =MEM_WB_OUT[70:69];
assign DataToReg  =MEM_WB_OUT[31:0];
assign MEM_WB_aluout=MEM_WB_OUT[63:32];
assign WB_rd      =MEM_WB_OUT[68:64];
//please connnect the CPU by yourself
//select which is to be written to register
always @*
begin
       case(WB_WDSel)
              `WDSel_FromALU: WD<=MEM_WB_aluout;
              `WDSel_FromMEM: WD<=DataToReg;//WD<=Data_in;/
              `WDSel_FromPC: WD<=WB_PC+4;//PC_out+4;
       endcase
end
endmodule
```

6.2 PC（程序计数器）

```
   module PC( clk, rst, NPC, PC,IS_STALL);
  input            clk;
  input            rst;
  input            IS_STALL;
  input      [31:0] NPC;
  output reg [31:0] PC;
  always @(posedge clk, posedge rst)
    if (rst)
      PC <= 32'h0000_0000;
//      PC <= 32'h0000_3000;
    else
      begin
        if(IS_STALL)PC<=PC;
//if the current instrcution is stalled, the pc should not change
      else
        begin
           PC <= NPC;
//if the current instruction is not stalled, the pc change to next
   pc
        end
      end
endmodule
```

6.3 Ctrl（控制模块）

```
1     // `include "ctrl_encode_def.v"
2
3   //123
4   module ctrl(Op, Funct7, Funct3, Zero,
5               RegWrite, MemWrite,
6               EXTOp, ALUOp, NPCOp,
7               ALUSrc, GPRSel, WDSel,DMType
8               ,MemRead
9               );
10    input [6:0] Op;      // opcode
11    input [6:0] Funct7;  // funct7
12    input [2:0] Funct3;  // funct3
13    input       Zero;
14  //useful in the single cycle cpu to judge whether sbtpye
        instruction is about to jump
15  //BUT IT IS USELESS in the pipeline cpu beceuse whether SBTYPE
        instruction is about to jump is decided in EX stage not ID stage
         where control module is in
16    output      RegWrite; // control signal for register write
17    output      MemWrite; // control signal for memory write
18    output [5:0] EXTOp; // control signal to signed extension
19    output [4:0] ALUOp; // ALU opertion
20    output [2:0] NPCOp; // next pc operation
21    output      ALUSrc;  // ALU source for B
22    output [2:0] DMType; // used for module DM(in the singlecycle cpu
          ,),or RAM_B(in the pipeline cpu)
23    output [1:0] GPRSel; // general purpose register selection
24    output [1:0] WDSel; // (register) write data selection
25    output      MemRead; // added for pipeline for hazard detect Unit
          to detect whether the instruction is to be stalled
26    // r format
27    wire rtype = ~Op[6]&Op[5]&Op[4]&~Op[3]&~Op[2]&Op[1]&Op[0]; //
          0110011
28    wire i_add = rtype& ~Funct7[6]&~Funct7[5]&~Funct7[4]&~Funct7[3]&~
          Funct7[2]&~Funct7[1]&~Funct7[0]&~Funct3[2]&~Funct3[1]&~Funct3
          [0]; // add 0000000 000
29    wire i_sub = rtype& ~Funct7[6]& Funct7[5]&~Funct7[4]&~Funct7[3]&~
          Funct7[2]&~Funct7[1]&~Funct7[0]&~Funct3[2]&~Funct3[1]&~Funct3
          [0]; // sub 0100000 000
30    wire i_or = rtype& ~Funct7[6]&~Funct7[5]&~Funct7[4]&~Funct7[3]&~
          Funct7[2]&~Funct7[1]&~Funct7[0]& Funct3[2]& Funct3[1]&~Funct3
          [0]; // or 0000000 110
31    wire i_and = rtype& ~Funct7[6]&~Funct7[5]&~Funct7[4]&~Funct7[3]&~
          Funct7[2]&~Funct7[1]&~Funct7[0]& Funct3[2]& Funct3[1]& Funct3
          [0]; // and 0000000 111
32    wire i_sll = rtype& ~Funct7[6]&~Funct7[5]&~Funct7[4]&~Funct7[3]&~
          Funct7[2]&~Funct7[1]&~Funct7[0]&~Funct3[2]&~Funct3[1]& Funct3
```

```
    [0]; // sll 0000000 001
33  wire i_slt = rtype& ~Funct7[6]&~Funct7[5]&~Funct7[4]&~Funct7[3]&~
    Funct7[2]&~Funct7[1]&~Funct7[0]&~Funct3[2]& Funct3[1]&~Funct3
    [0]; // slt 0000000 010
34  wire i_sltu= rtype& ~Funct7[6]&~Funct7[5]&~Funct7[4]&~Funct7[3]&~
    Funct7[2]&~Funct7[1]&~Funct7[0]&~Funct3[2]& Funct3[1]& Funct3
    [0]; // sltu 0000000 011
35  wire i_xor = rtype& ~Funct7[6]&~Funct7[5]&~Funct7[4]&~Funct7[3]&~
    Funct7[2]&~Funct7[1]&~Funct7[0]& Funct3[2]&~Funct3[1]&~Funct3
    [0]; // xor 0000000 100
36  wire i_srl = rtype& ~Funct7[6]&~Funct7[5]&~Funct7[4]&~Funct7[3]&~
    Funct7[2]&~Funct7[1]&~Funct7[0]& Funct3[2]&~Funct3[1]& Funct3
    [0]; // srl 0000000 101
37  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
38  // i format
39  wire itype_l = ~Op[6]&~Op[5]&~Op[4]&~Op[3]&~Op[2]&Op[1]&Op[0]; //
    0000011
40  wire i_lb   = itype_l&~Funct3[2]&~Funct3[1]&~Funct3[0]; //lb  000
41  wire i_lh   = itype_l&~Funct3[2]&~Funct3[1]& Funct3[0]; //lh  001
42  wire i_lw   = itype_l&~Funct3[2]& Funct3[1]&~Funct3[0]; //lw  010
43  wire i_lbu  = itype_l& Funct3[2]&~Funct3[1]&~Funct3[0]; //lbu 100
44  wire i_lhu  = itype_l& Funct3[2]&~Funct3[1]& Funct3[0]; //lhu 101
45  // i format
46  wire itype_r = ~Op[6]&~Op[5]&Op[4]&~Op[3]&~Op[2]&Op[1]&Op[0]; //
    0010011
47  wire i_addi = itype_r&~Funct3[2]&~Funct3[1]&~Funct3[0]; // addi
    000
48  wire i_ori  = itype_r& Funct3[2]& Funct3[1]&~Funct3[0]; // ori 110
49  wire i_andi = itype_r& Funct3[2]& Funct3[1]& Funct3[0]; //andi 111
50  wire i_xori = itype_r& Funct3[2]&~Funct3[1]&~Funct3[0]; //xori 100
51  wire i_slti = itype_r&~Funct3[2]& Funct3[1]&~Funct3[0]; //slti 010
52  wire i_sltiu = itype_r&~Funct3[2]& Funct3[1]& Funct3[0]; //
    sltiu011
53  wire i_slli = itype_r&~Funct3[2]&~Funct3[1]& Funct3[0]; //slli 001
54  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
55  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
56  //jalr
57  wire i_jalr =Op[6]&Op[5]&~Op[4]&~Op[3]&Op[2]&Op[1]&Op[0];//jalr
    1100111
58  // s format
```

```
59    wire stype = ~Op[6]&Op[5]&~Op[4]&~Op[3]&~Op[2]&Op[1]&Op[0];//
          0100011
60    wire i_sw = stype&~Funct3[2]& Funct3[1]&~Funct3[0];
61    wire i_sh = stype&~Funct3[2]&~Funct3[1]& Funct3[0];
62    wire i_sb = stype&~Funct3[2]&~Funct3[1]&~Funct3[0];
63    // sb format
64     wire sbtype = Op[6]&Op[5]&~Op[4]&~Op[3]&~Op[2]&Op[1]&Op[0];//
          1100011
65     wire i_beq = sbtype&~Funct3[2]&~Funct3[1]&~Funct3[0];
66     wire i_bne = sbtype&~Funct3[2]&~Funct3[1]& Funct3[0];
67     wire i_blt = sbtype& Funct3[2]&~Funct3[1]&~Funct3[0];
68     wire i_bge = sbtype& Funct3[2]&~Funct3[1]& Funct3[0];
69     wire i_bltu = sbtype& Funct3[2]& Funct3[1]&~Funct3[0];
70     wire i_bgeu = sbtype& Funct3[2]& Funct3[1]& Funct3[0];
71   // j format
72     wire i_jal = Op[6]& Op[5]&~Op[4]& Op[3]& Op[2]& Op[1]& Op[0]; //
          jal 1101111
73    wire i_auipc = ~Op[6]&~Op[5]&Op[4]&~Op[3]&Op[2]&Op[1]&Op[0];
74    wire i_lui  = ~Op[6]& Op[5]&Op[4]&~Op[3]&Op[2]&Op[1]&Op[0];
75    // generate control signals
76   assign RegWrite = rtype | itype_l | itype_r | i_jalr | i_jal |
         i_lui | i_auipc; // register write
77    assign MemWrite = stype;                          // memory write
78    assign ALUSrc   = itype_l | itype_r | stype | i_jal | i_jalr |
         i_auipc | i_lui; // ALU B is from instruction immediate
79    //decide the extension of the immediate
80    assign EXTOp[5] = i_slli | i_srli | i_srai;
81    assign EXTOp[4] = itype_l | i_addi | i_slti | i_sltiu | i_xori |
         i_ori | i_andi | i_jalr;
82    assign EXTOp[3] = stype;
83    assign EXTOp[2] = sbtype;
84    assign EXTOp[1] = i_auipc|i_lui;
85    assign EXTOp[0] = i_jal;
86    //decide the read/write of data memory
87    assign DMType[0] = i_lb | i_lh | i_sb | i_sh;
88    assign DMType[1] = i_lhu | i_lb | i_sb;
89    assign DMType[2] = i_lbu;
90    //decide what data is to be written to the register
91    assign WDSel[0] = itype_l;
92    assign WDSel[1] = i_jal | i_jalr;
93    //decide the next PC
94    assign NPCOp[0] = sbtype; //& Zero;
95  //the "&"operation is executed in the EX stage
96    assign NPCOp[1] = i_jal;
97    assign NPCOp[2] = i_jalr;
98    //decide the ALU operation
```

```
99    assign ALUOp[0] = i_jal|i_jalr|itype_l|stype|i_addi|i_ori|i_add|
          i_or|i_bne|i_bge|i_bgeu|i_sltiu|i_sltu|i_slli|i_sll|i_sra|
          i_srai|i_lui;
100   assign ALUOp[1] = i_jal|i_jalr|itype_l|stype|i_addi|i_add|i_and|
          i_andi|i_auipc|i_blt|i_bge|i_slt|i_slti|i_sltiu|i_sltu|i_slli|
          i_sll;
101   assign ALUOp[2] = i_andi|i_and|i_ori|i_or|i_beq|i_sub|i_bne|i_blt|
          i_bge|i_xor|i_xori|i_sll|i_slli;//
102   assign ALUOp[3] = i_andi|i_and|i_ori|i_or|i_bltu|i_bgeu|i_slt|
          i_slti|i_sltiu|i_sltu|i_xor|i_xori|i_sll|i_slli;
103   assign ALUOp[4] = i_srl|i_sra|i_srli|i_srai;
104   // assignmemt of the MemRead signal for HAZARD DETECT UNIT
105   assign MemRead=itype_l;
106 endmodule
```

6.4 EXT（符号扩展模块）

```
1     `include "ctrl_encode_def.v"
2   module EXT(
3   input  [4:0]          iimm_shamt,
4   //use for SLLI,SRLI,SRAI instruction
5      input  [11:0]         iimm,     //instr[31:20], 12 bits
6      input  [11:0]         simm,     //instr[31:25, 11:7], 12 bits
7   input  [11:0]         bimm,
8   //instrD[31], instrD[7], instrD[30:25], instrD[11:8], 12 bits
9      input  [19:0]         uimm,     //use for lui instruction
10     input  [19:0]         jimm,     //use for jal instruction
11  input  [5:0]          EXTOp,
12  //to decide how to extend immediate
13     output reg [31:0]     immout); //result of immediate extension
14  always @(*)
15      case (EXTOp)
16         `EXT_CTRL_ITYPE_SHAMT: immout<={27'b0,iimm_shamt[4:0]};
17         `EXT_CTRL_ITYPE:  immout <= {{{32-12}{iimm[11]}}, iimm[11:0]};
18       `EXT_CTRL_STYPE:immout <= {{{32-12}{simm[11]}}, simm[11:0]};
19       `EXT_CTRL_BTYPE:immout <= {{{32-13}{bimm[11]}}, bimm[11:0], 1'
            b0};
20       `EXT_CTRL_UTYPE:immout <= {uimm[19:0], 12'b0};
21       `EXT_CTRL_JTYPE:immout <= {{{32-21}{jimm[19]}}, jimm[19:0], 1'
            b0};
22        default:     immout <= 32'b0;
23      endcase
24  endmodule
```

6.5 RF（寄存器模块）

本部分中省去了在流水线仿真中所用到的所有 display 语句，具体 display 语句的用法可见附件中流水线仿真的具体代码.

```
   module RF( input        clk,
              input        rst,
              input        RFWr,
              input [4:0] A1, A2, A3,
              input [31:0] WD,
              output [31:0] RD1, RD2);
  //32 registers, each register has 32bits
  reg [31:0] rf[31:0];
  integer i;
  //initialization for every RF unit
  integer j;
  initial begin
  for(j=0;j<32;j=j+1)
  rf[j]<=0;
  end
  always @(negedge clk, posedge rst)
    if (rst) begin  // reset
      for (i=1; i<32; i=i+1)
        rf[i] <= 0; // i;
    end
    else
    //the number 0 register has the static value 0,which cant be
       changed
      if (RFWr&&A3!=0) begin
//if the target register is the number 0 register, the write
   request is banned
        rf[A3] <= WD[31:0];
      end
    //if it needs to read data from the number 0 register, the
       output value is 0
  assign RD1 = (A1 != 0) ? rf[A1] : 0;
  assign RD2 = (A2 != 0) ? rf[A2] : 0;
  //assign reg_data = (reg_sel != 0) ? rf[reg_sel] : 0;
endmodule
```

6.6 HazardDectectionUnit（冒险检测单元）

```
   module HazardDectectionUnit
(
 input     EX_MemRead, //MemRead signal from ID/EX pipeline register
```

```
 input [4:0]ID_rs1,   //rs1 signal from IF/ID pipeline register
 input [4:0]ID_rs2,   //rs2 signal from IF/ID pipeline register
 input [4:0]EX_rd,    //rd signal from ID/EX pipeline register
 output    stall_signal //stall signal
);
//the HAZARD DECTECT UNIT is used for LOAD instruction such as ld, lb,lbu..... TO SOLVE DATA HAZARD
//whether the register which previous instruction is to write is equal to the register which current instruction is read
//rd=rs1 or rd=rs2
assign stall_signal=EX_MemRead&(~(|(EX_rd^ID_rs1))|~(|(EX_rd^ID_rs2)));
endmodule
```

6.7 GRE_array（流水线寄存器模块）

```
    define WIDTH_PIPELINE 128
    module GRE_array(
    input Clk,Rst,write_enable,flush,
    input    [180:0] in,
    output reg[180:0] out
    );
    always@(posedge Clk,posedge Rst)
    begin
        if(Rst) begin out = 0; end
        else begin
            if(write_enable)
//whether write to an pipeline register is permitter
            begin
                if(flush)//whether it is to be flushed
                    out=0;
                else
                    out =in;
            end
        end
   end
endmodule
```

6.8 ForwardingUnit（旁路前递模块）

```
    module ForwardingUnit(
input     MEM_RegWrite,
// EX/MEM RegWrite ,RegWrite signal from EX/MEM pipeline register
input [4:0]MEM_rd,
```

```
5   // EX/MEM RegisterRd,rd signal from EX/MEM pipeline register
6   input     WB_RegWrite,
7   // MEM/WB RegWrite, RegWrite signal from MEM/WB pipeline register
8   input [4:0]WB_rd,
9   // MEM/WB RegisterRd,rd signal from MEM/WB pipeline register
10  input [4:0]Ex_rs,
11  // EX/MEM RegisterRs1/2,rs1 and rs2 signal from EX/MEM pipeline
        register
12  output[1:0]ForwardSignal
13  // 00:from regfile,10:from MEM_aluout,01:from WB_WD
14  );
15  //the FORWARD UNIT is used for DATA HAZARD
16  //to choose two operation number for ALU module
17  wire MEM_Forward;
18  wire WB_Forward;
19  assign MEM_Forward =~(|(MEM_rd^Ex_rs))&MEM_RegWrite;
20  //decide whether the register which previous 1 instruction is to
        write is equal to the register which current instruction is to
        read
21  assign WB_Forward =~(|(WB_rd^Ex_rs))&WB_RegWrite&~MEM_Forward;
22  //decide whether the register which previous 2 instruction is to
        write is equal to the register which current instruction is to
        read
23  assign ForwardSignal ={MEM_Forward,WB_Forward};
24  endmodule
```

6.9 ALU（运算器单元）

```
1     include "ctrl_encode_def.v"
2   module alu(A, B, ALUOp, C, Zero,PC);
3     input signed [31:0] A, B;
4     input        [4:0] ALUOp;
5     input        [31:0] PC;
6     output signed [31:0] C;
7     output             Zero;
8     reg [31:0] C;
9     integer   i;
10    always @( * ) begin
11       case ( ALUOp )
12  `ALUOp_nop: C=A;
13  `ALUOp_lui: C=B;
14  `ALUOp_auipc:C=PC+B;
15  `ALUOp_add: C=A+B;
16  `ALUOp_sub: C=A-B;
17  `ALUOp_bne: C={31'b0,(A==B)};
18  `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
    // Zero=(C==32'b0);
  end // end always
  assign Zero = (C == 32'b0);
endmodule
```

6.10 NPC（下一条指令地址模块）

```
    `include "ctrl_encode_def.v"
module NPC(PC, NPCOp, IMM, NPC,aluout,EX_PC); // next pc module
  input [31:0] PC;      // pc
  input [2:0] NPCOp;    // next pc operation
  input [31:0] IMM;     // immediate
  input [31:0] aluout;
  input [31:0]EX_PC;
  output reg [31:0] NPC; // next pc
  wire [31:0] PCPLUS4;
  assign PCPLUS4 = PC + 4; // pc + 4
  always @(*) begin
     begin
     case (NPCOp)
         `NPC_PLUS4: NPC = PCPLUS4;
         `NPC_BRANCH: NPC = EX_PC+IMM;
//if NPCOp for branch is true, the next PC should be the current PC
    (EX_PC) add immediate
         `NPC_JUMP:  NPC = EX_PC+IMM;
//if NPCOp for jal is true, the next PC should be the current PC(
    EX_PC) add immediate
         `NPC_JALR:  NPC = aluout;
//if NPCOp for jalr is true,the next PC should be value from
    register add immediate which is aluout
          default:   NPC = PCPLUS4;
     endcase
     end
  end // end always
```

```
25  endmodule
```

7 测试结果及分析

7.1 仿真代码及分析

```
ori x5, x0, 0x435#for i-type
lui x6, 0x1
or x5,x5,x6
lui x6, 0x98765
addi x7, x5, 0x578
addi x8, x6, -1024
xori x9, x5, 0x7bc
sltiu x3, x7, 0x19
sltiu x4, x5, -1
andi x18, x9, 0x765
slti x20, x6, 0x123
sub x19, x6, x5  #for r-type
xor x21, x20, x6
add x22, x21, x20
add x22, x22, x5
sub x23, x22, x6
or  x25, x23, x22
and x26, x23, x22
slt x27, x25, x26
sltu x28, x25, x26
addi x3, x3, 4 #for shift
sll x27, x26, x3
srl x28, x25, x3
sra x29, x25, x3
slli  x27, x19, 24
srli x28, x19, 4
srai x29, x19, 4
addi x3, x0, 0
addi x5,x0, 0xEF
#for store
sw x19, 0(x3)
sw x21, 4(x3)
sw x23, 8(x3)
sh x26, 4(x3)
sh x19, 10(x3)
sb x5, 7(x3)
sb x5, 9(x3)
sb x5, 8(x3)
```

```
#for load test & data hazard test
       lw  x5, 0(x3)
       sw x5, 12(x3)
       lh x7, 2(x3)
       sw x7, 16(x3)
       lhu x7, 2(x3)
       sw x7, 20(x3)
       lb x8, 3(x3)
       sw x8, 24(x3)
       lbu x8, 3(x3)
       sw x8, 28(x3)
       lbu x8, 1(x3)
       sw x8, 32(x3)
#for sb-type test
       sw x0, 0(x3)
       and x9, x0, x9
       bne x5, x7, _lb1
       addi x9, x9, 2
_lb1:  bge x5, x7, _lb2
       addi x9, x9, 7
_lb2:  bgeu x5, x7, _lb3
       addi x9, x9,5
_lb3:  blt x5, x7, _lb4
       addi x9, x9, 3
_lb4:  bltu x5, x7, _lb4
       addi x9, x9, 6
_lb5:  beq x7, x8, _lb6
       addi x9, x9, 1
_lb6:  sw x9, 0(x3)
# for jal & jalr test
       lw x10, 0(x3)
       jal x1, F_Test_JAL
       addi x10, x10, 3
       sw x10, 0(x3)
F_Test_JAL:
       ori x10, x10, 0x7a1
       sw x10, 0(x3)
       jalr x0, x1, 0
```

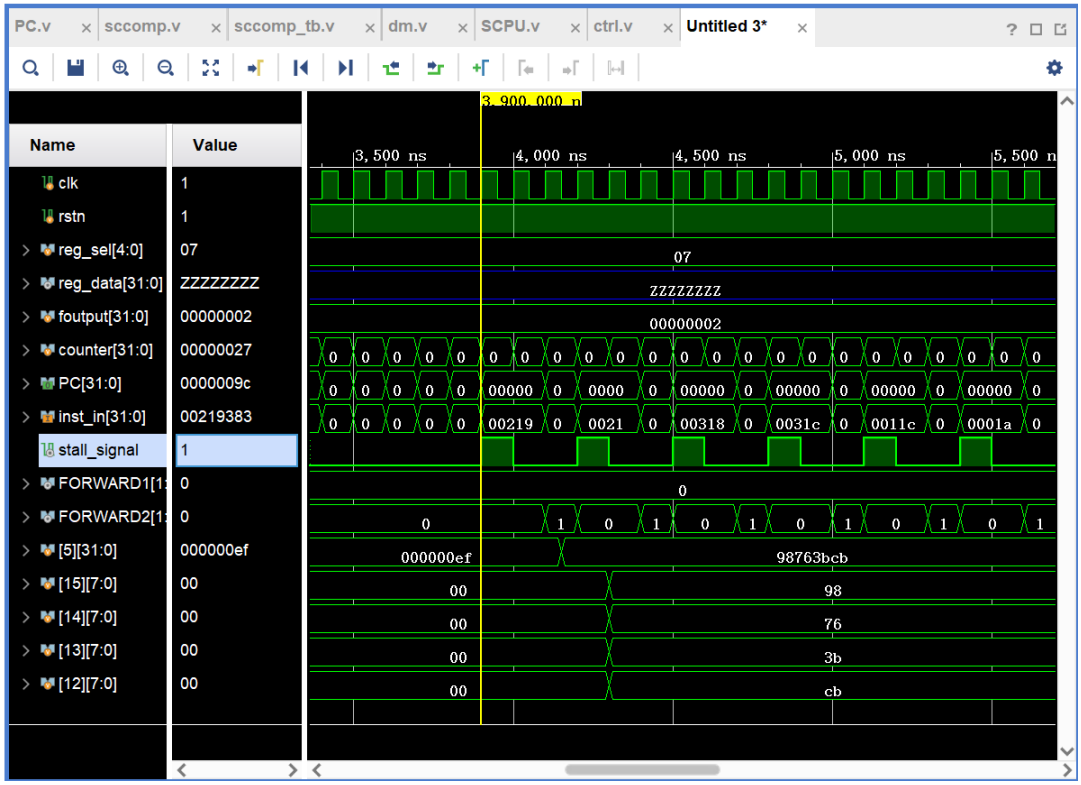
以上的仿真代码共分为七个部分，分别是对于 I 型、R 型、移位型（特殊 I 型）、S 型、SB 型、J 型和 U 型的测试指令集合。

7.2 仿真测试结果

7.2.1 数据冒险

代码执行到第 41 行时出现了数据冒险，当第 41 行代码（sw 指令）执行到 ID 级时，需要读取寄存器 x5 中的值，但是此时第 40 行代码（lw 指令）的 lw 指令还在 EX 级，还没有将数据写回 x5 寄存器，因此需要对第 41 行代码的 sw 进行阻塞。

图 7.1 检测到 ld-use 结构并开始阻塞



而到了第 40 行代码的 MEM 级结束时，已经从内存中读出了相应的值并要在该周期下降沿写回寄存器，（如图 7.3 所示，可以看到此时 x5 的值已经变为了 0x98763bcb）因此直接通过旁路 2（此时 forward2=1）将其传回 sw 指令的 EX 级，在下一时钟周期的上升沿可以看到其内存单元已经变为 0x98763bcb（如图 7.4 所

示），由于是 sw 指令因此是 32 位占用四个 dm 单元。

图 7.2　在 lw 指令 WB 级时通过旁路传回寄存器值

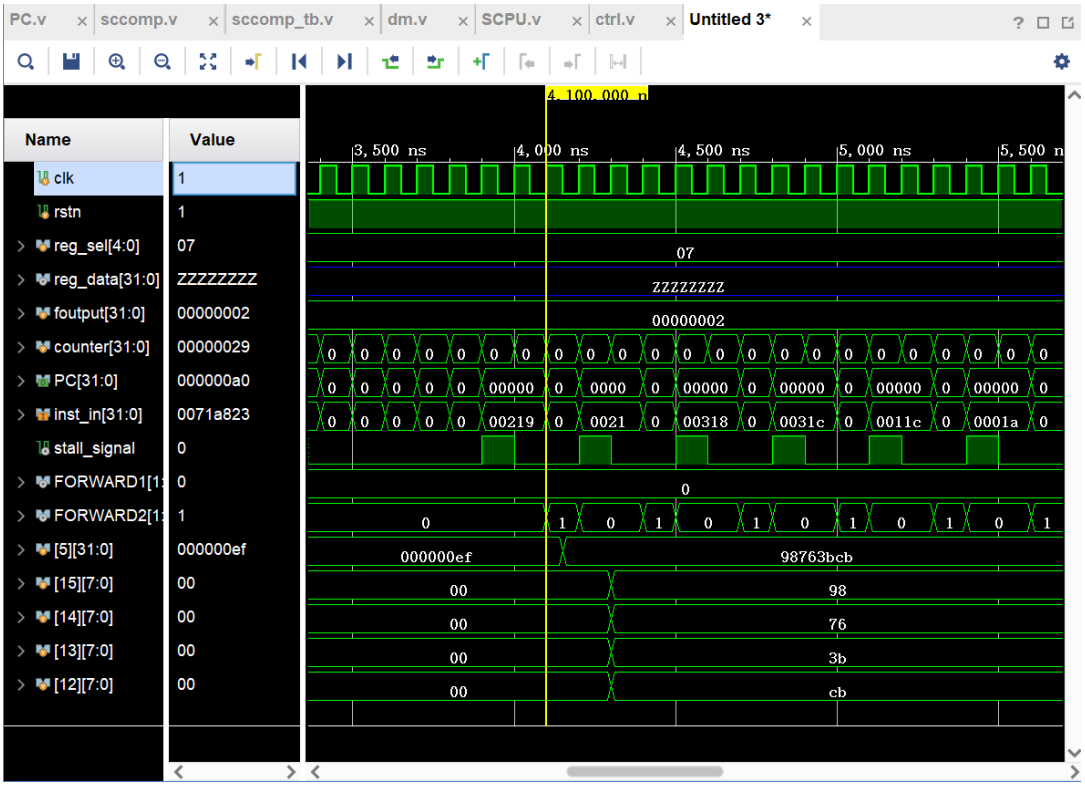


图 7.3　在 lw 指令 WB 级下降沿将数据写回寄存器

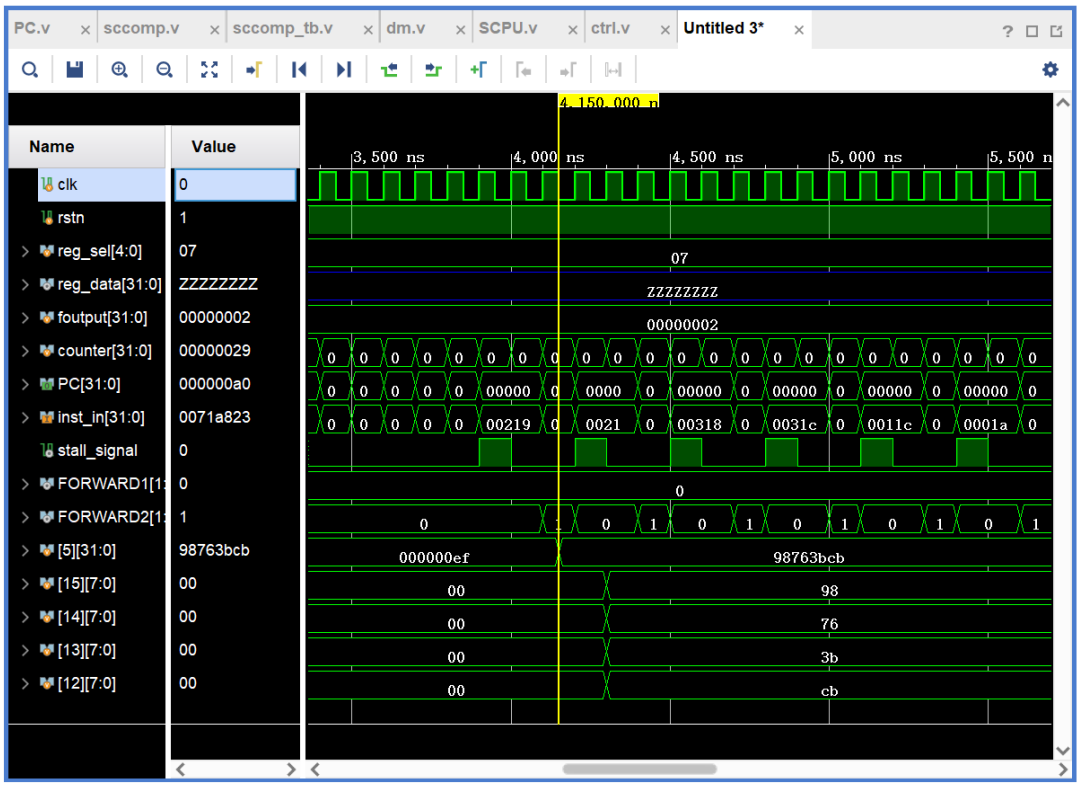
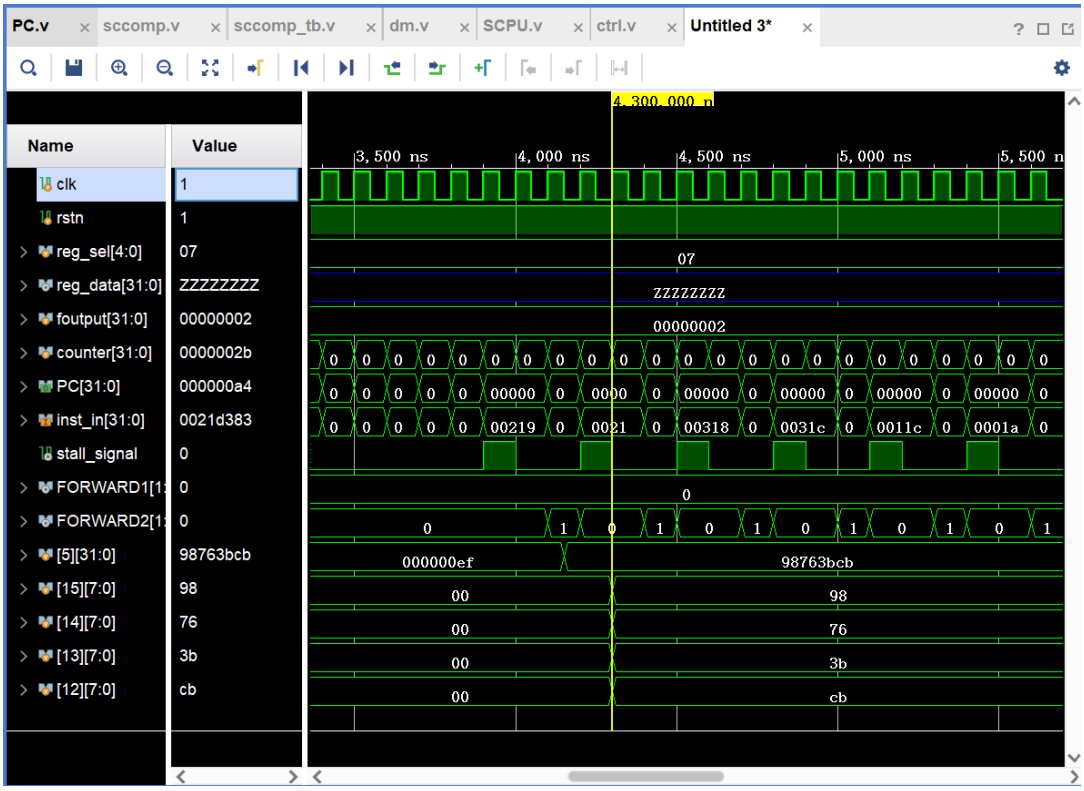


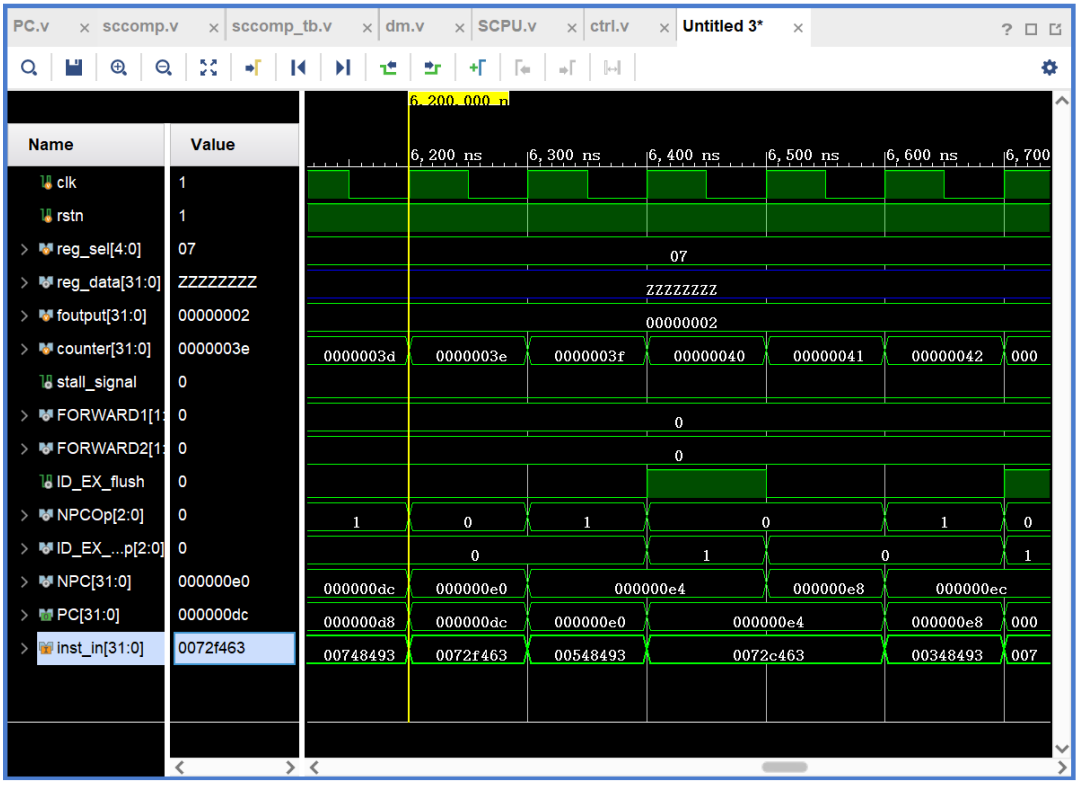
图 7.4　在 sw 指令的 MEM 级上升沿写入 dm



7.2.2 控制冒险（对于 SB 型指令）

当代码执行到第 59 行（bgeu 指令）时，第 57 行代码的 bge 指令正处在 EX 级，其计算结果满足 bge 指令的条件，需要发生转跳，但是目前 58 行的代码（addi 指令）正在 ID 级，59 行的代码（bgeu 指令）正在 IF 级，因此需要同时清空 IF/ID,ID/EX 两个流水线寄存器（如图 7.5，7.6 所示）来保证转跳的顺利进行。

图 7.5　bge 指令的 IF 级



在 bge 指令的 MEM 级，如图 7.7 所示，我们可以看到转跳已经发生，由于在 bge 指令的 EX 级周期处于 IF 级的正好是转跳的目标 bgeu 指令，因此图中所示清空流水线起存器前后的输入指令相同，由于控制下一 PC 的 NPCOp 变量为 1，，可以知道转跳已经发生。

图 7.6　bge 指令的 EX 级

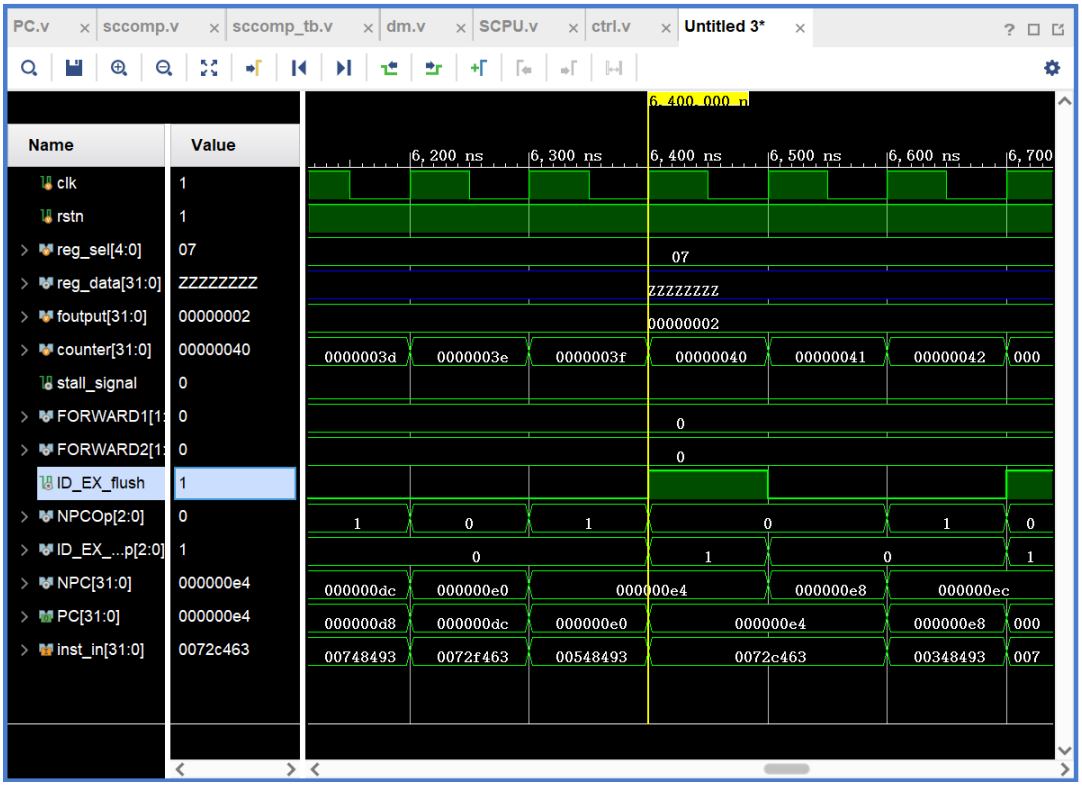
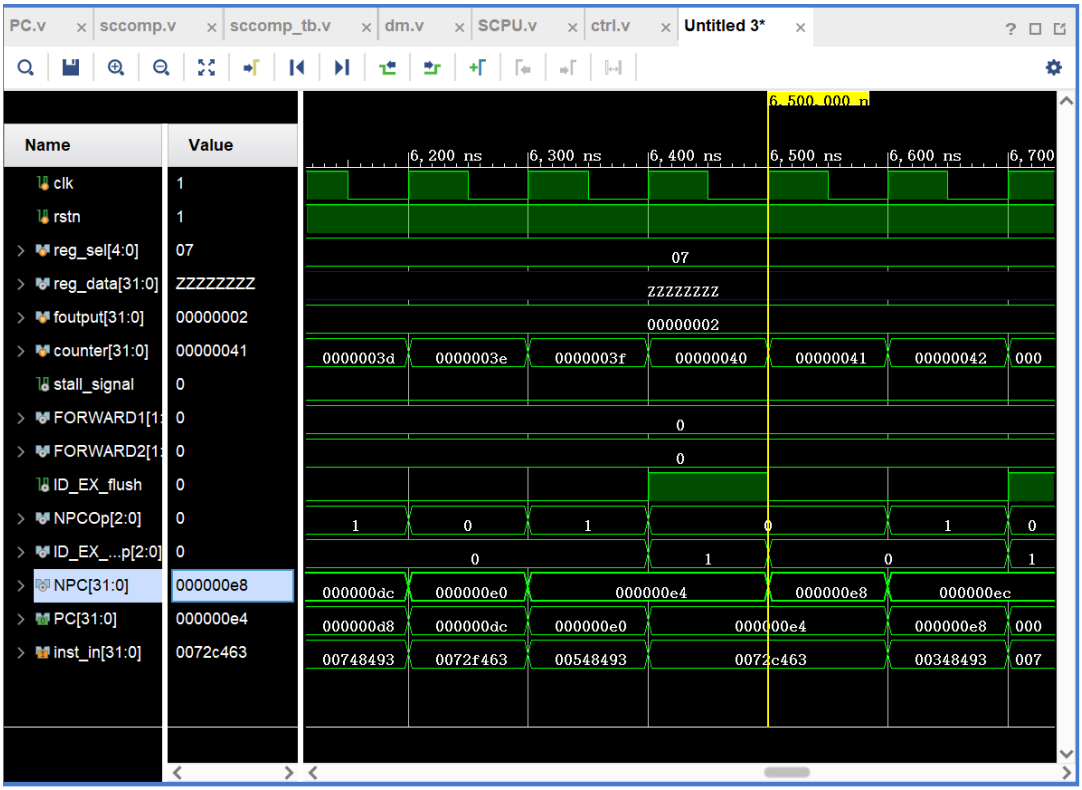


图 7.7　发生转跳



7.2.3 控制冒险（对于 J 型指令 jal）

当代码执行到第 72 行（sw 指令）时，此时是该指令的 IF 级，同时也是第 70 行代码（jal 指令）的 EX 级，下一指令模块已经计算出了下一周期的 PC 位置。因此此时需要清空 IF/ID,ID/EX 两个流水线寄存器并在下一周期转跳至目标位置。（如图 7.8 和图 7.9 所示）

图 7.8　jal 指令的 IF 级

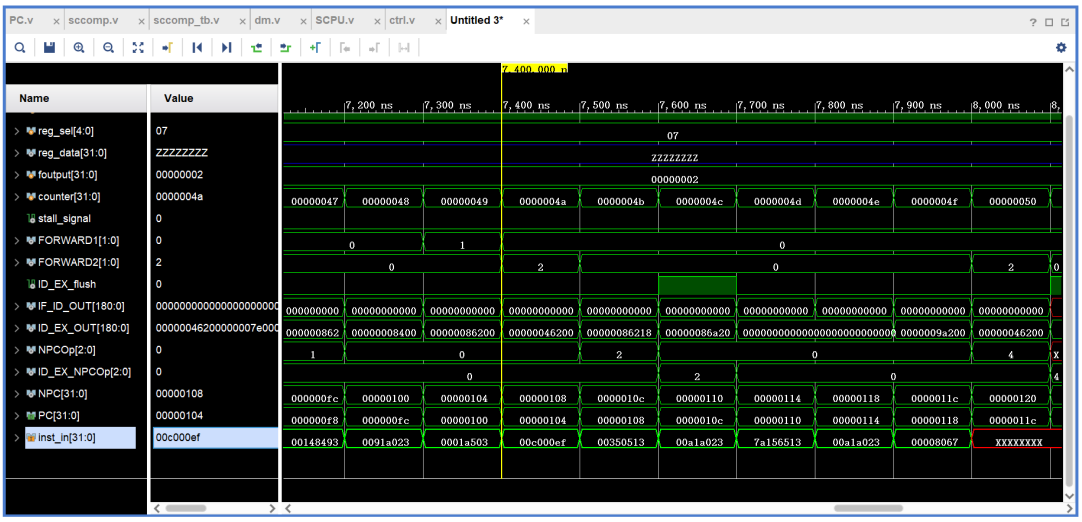


图 7.9　jal 指令的 EX 级

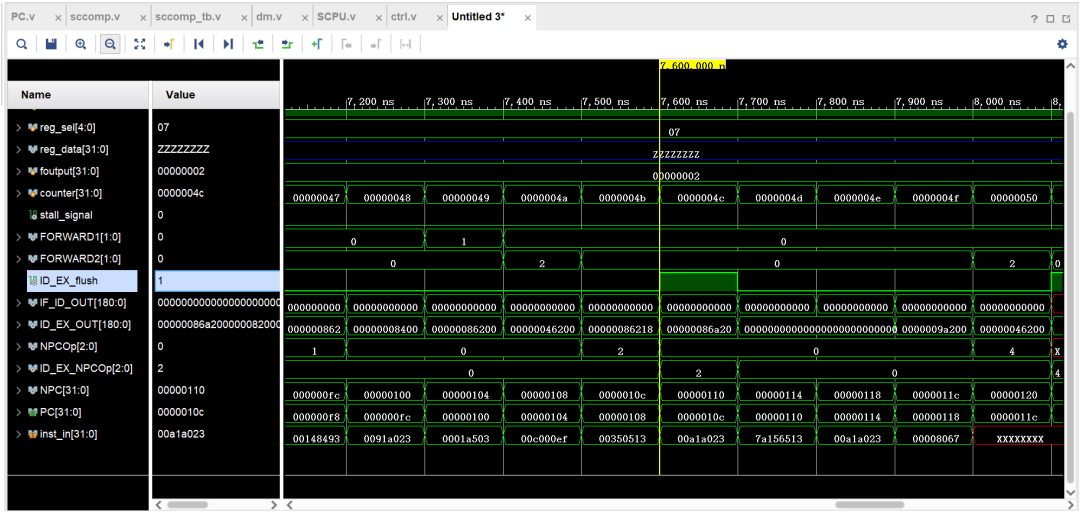
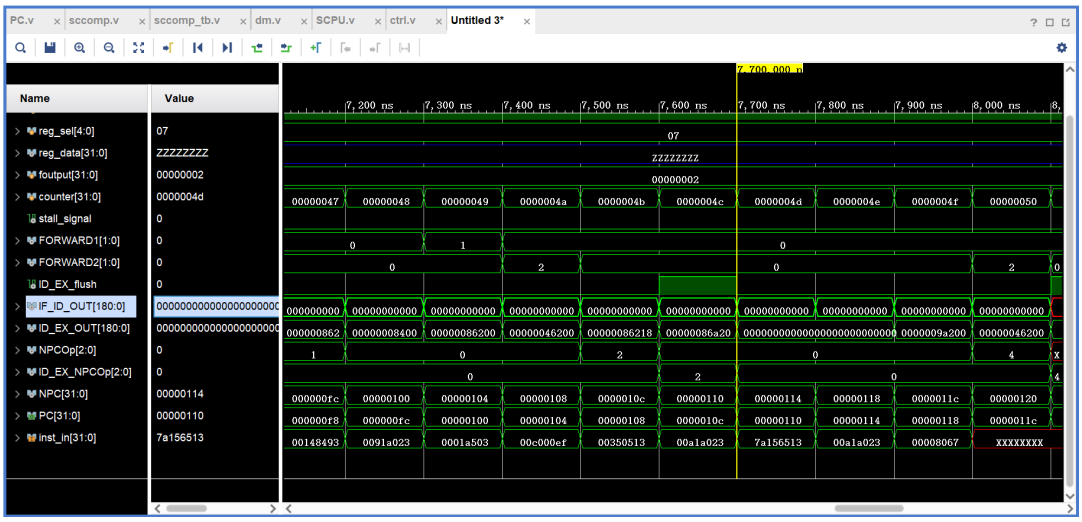


图 7.10　发生跳转



如图 7.10 所示，IF/ID,ID/EX 两个流水线寄存器都被清空并且指令已经根据 NPCOp（其值为 2 代表 jal 指令发生转跳）变量的控制跳转至第 74 行（ori 指令）处，并从此处往后继续执行。

7.3　下载测试代码分析

测试汇编的代码在 nn.asm 文件中。本文件实现了通过拨码开关输入一个数字 n 并且在数码管上输出其平方后的结果。

在将指令导入到实验板上时，基于 MIO_BUS.v,top.v,MULTI_CH32.v 等文件，修改了文件中的输入和输出以及显示模块，使其能够支持拨码开关的输入和数码管上的数字输出。

本实验中使用开关 15 来控制图形/数字显示模块，通过 10-14 号开关来输入需要计算的 n，将其储存在寄存器 x6 中、并将计算的结果储存在寄存器 x7 中。通过 0-2 号开关来展示输出结果的值以及计数器中存储的值。计算及使用的方式为：在拨码开关上输入 n 后，根据 n 个 n 循环加的方式得到平方数的结果，并在数码管上循环显示结果的值。

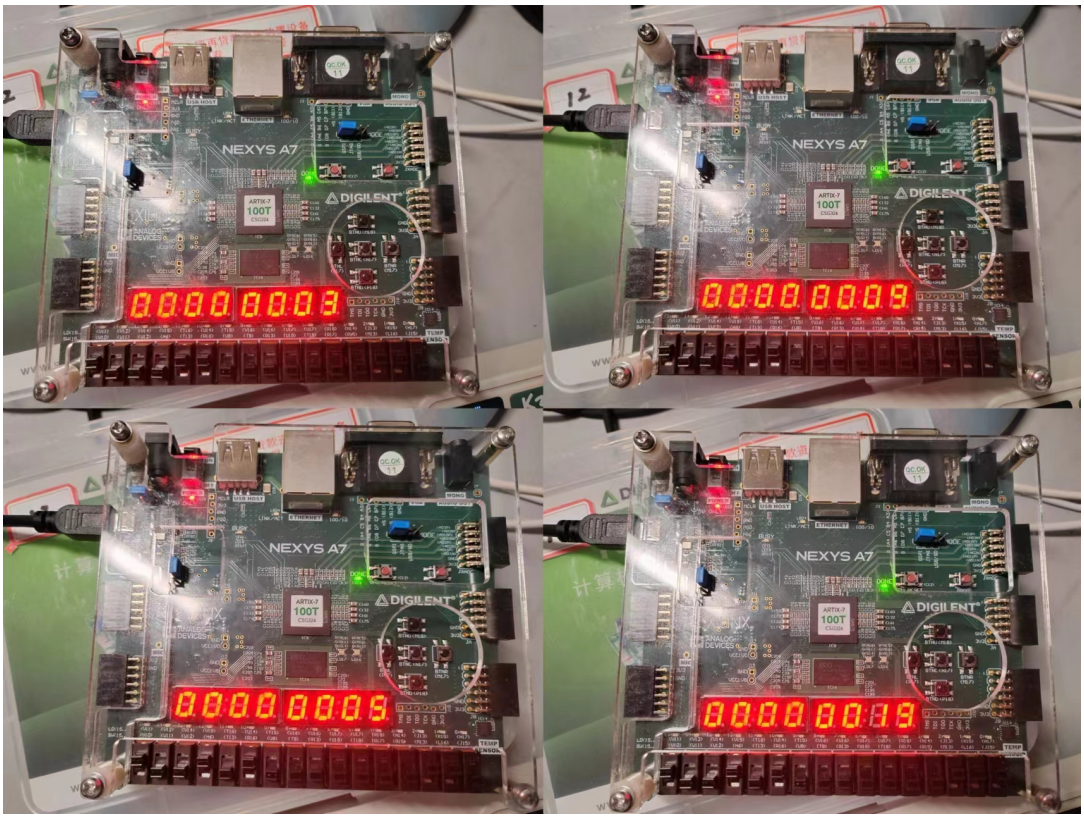
7.4 下载测试结果

如下图所示，分别在数码管上输入 3 和 5 两个值，可以看到数码管上显示的结果寄存器中的值分别为 0x9 和 0x19（25）；通过输入其他的值也可以得到相应的结果。

经过调试，计算器的部分计算正确，数码管上也可以正确的显示结果的十六进制表示。

完成本计算器的代码中通过 SB 型指令、jal 指令、jalr 指令来进行循环的转跳，并且也通过存取内存和寄存器来测试数据冒险和 ld-use 类型的冒险，均能够正常执行，因此认为可以正常并正确处理数据冒险、控制冒险等问题。

图 7.11 调试结果



7.5 实验中所遇到的问题

7.5.1 流水线寄存器多重赋值

原流水线寄存器模块的实现方法如下图所示：

图 7.12　原流水线寄存器模块

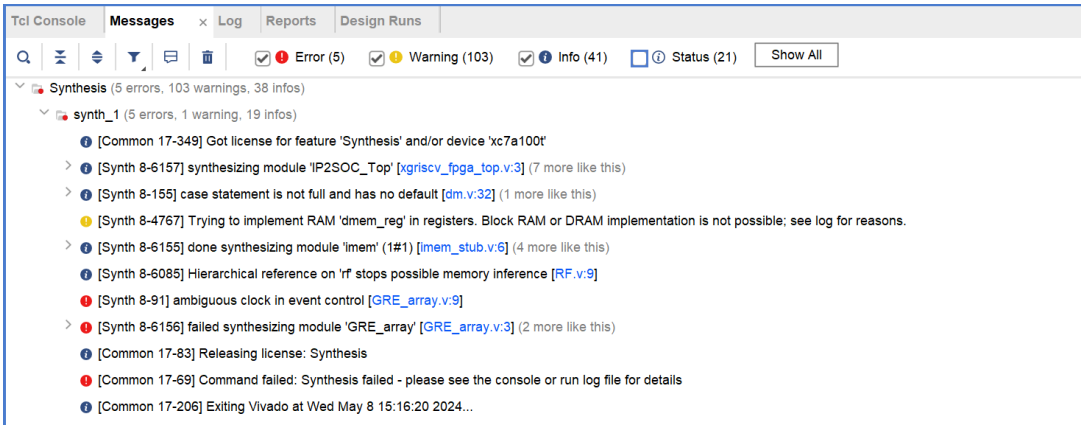
```
module GRE_array(
    input Clk,Rst,write_enable,flush,
    input     [180:0] in,
    output reg[180:0] out
    );

    always@(posedge Clk,posedge Rst)
    begin
//        if(Rst) begin out = 0; end
//        else
        begin
            if(write_enable)//whether write to an pipeline register is permitter
            begin
                if(flush)//whether it is to be flushed
                    out=0;
                else
                    out =in;
            end
        end
    end
    always@(posedge Rst)
    begin
    out=0;
    end

endmodule
```

其写法在仿真中能够正常执行，但是在下载到实验板的综合过程中会报出如下的错误:

图 7.13　原流水线寄存器模块错误



该错误经检查意为在多个 always 语句块中对同一个变量 out 进行赋值，因此改进方法是通过添加 if 语句将其写入同一个 always 语句块中实现。其实现代码如下图所示:

图 7.14　改进后的流水线寄存器模块

```
module GRE_array(
    input Clk,Rst,write_enable,flush,
    input     [180:0] in,
    output reg[180:0] out
    );

    always@(posedge Clk,posedge Rst)
    begin
       if(Rst) begin out = 0; end
        else
        begin
            if(write_enable)//whether write to an pipeline register is permitter
            begin
                if(flush)//whether it is to be flushed
                    out=0;
                else
                    out =in;
            end
        end
   end


endmodule
```

7.5.2　转跳指令下一周期 PC 的决定位置

根据上学期《计算机组成与设计》课程中教材上所写的，转跳指令下一指令地址需要在该指令的 MEM 级（alu 计算出 zero 变量之后）决定。但是考虑到若在 MEM 级决定那么其后就将会有三条指令进入流水线，如果转跳发生，那么需要同时清空 IF/ID,ID/EX,EX/MA 三个流水线寄存器，也同时意味着在 EX 级产生清空控制信号（ID_EX_flush）需要经过 EX/MA 流水线寄存器传输到下一周期再进行清空。

因此，为避免同时清空三个流水线寄存器带来的软硬件开销并简化流水线中的数据通路，本实验中选择将 Zero 值的计算和转跳指令下一周期 PC 值的决定都设置在 EX 级进行。具体实现的代码如下：

```
assign Branch_or_jump= ID_EX_NPCOp[2]|ID_EX_NPCOp[1]|ID_EX_NPCOp[0];
assign ID_EX_flush= stall_signal | Branch_or_jump;
```

由于将下一周期 PC 决定设置在 EX 级，因此在该 PC 决定时仅有两条可能无关的指令进入流水线，故如前文展示的处理控制冒险的方法一样，只需要同时清空 IF/ID,ID/EX 流水线寄存器中的信号即可。

8 实验总结

8.1 实验总结

本次实验是基于 RISC-V 语言架构来设计的单周期和流水线 CPU，同时根据 Nexys A7 实验开发板的输入和输出功能开发了其他的交互式的内容。一方面，本次实验使用了一门不同于其他高级语言的 Verilog 作为开发语言，另一方面，在流水线 CPU 中面对增加的四个流水线寄存器和每一级都要使用的控制信号使得数据通路的传输、搭建和调试变的较为困难，这两个方面都是得本次实验并不易于完成。

单周期 CPU 的实现在老师所给代码的框架下的实现较为容易，但是将其改造成能够处理包括 ld-use 类型的控制冒险在内的冒险以及控制冒险就并非易事。数据并不只是像单周期那样简单的进行模块对接的传输方式。时钟是上升沿写、下降沿写数据异或是什么时间读数据，阻塞与非阻塞的判断，清空与不清空的操作都可能对于有时序特性的数字电路带来不小的影响。因此能够完整的完成本实验也是对于自我的一大挑战。

在最终将测试代码在实验板上实现的过程中，对于 Nexys A7 实验板以及其背后的 FPGA 控制的原理、输入输出特性、每个按键的功能和操作方式往往需要数十小时乃至数天的摸索来不断实验，在此期间来自老师和同学间的交流和对资料文献的查阅能力就显得十分重要，但是最主要的还是能够亲自动手在实验板上不断探索实验。

最终，虽然仅仅是实现了求给定数字平方的简单功能，但是在此过程中也遇到并解决了不少于 vivado 软件和 verilog 以及 risc-v 汇编语言相关的问题。代码的正常执行，数码管显示的计算结果正确，寄存器和内存内容符合 venus 网站上的模拟结果说明实验的成功完成，

8.2 实验取得的收获

在为期六周的课程时间中，本实验的四个任务无疑给我的动手实践能力，自助学习能力，自主调试能力带来了不小的提升，也将上学期中所学的《计算机组成与设计》课程中的知识融汇贯通，并巩固了使用 RISC-V 汇编语言的程序设计和使用 Verilog 语言编程的能力。

在完成流水线 CPU 的设计和搭建的过程中，除了前文中所提到的两个主要的问题外，其实还有大大小小不少关于软件和硬件的问题。例如仿真时经常出现的大红 X、或是大蓝 Z；又或是看不懂的英文报错等等。对此，我增加的 display 语句对重要的变量进行输出，并且依据数据通路的连接方式追根溯源得到高阻（X）状态的产生与初始化，未定义，值错误等许多问题有关。并且不断在约束文件增加内容以解决报错问题。在此之后，我解决问题的速度和能力也在不断提升。

在实验中遇到的问题同时也在不断启发着我，例如流水线 CPU 的结构是否还能简化，数据通路的连接是否太过于冗杂，代码的实现有没有逻辑漏洞等。这也促使着我不断追寻着软硬件中、各模块所遇到的问题的根源并将其解决。

经历了四五天的连续探索，让我成功完成本次实验的同时也带给了我不小的成就感和满足感。同样也让我理解到计算机不只是软件，也不只是硬件，只有通过软硬件的交叉结合，计算机才能发挥其作用。

参考文献

[1] 张学镇, 汪西虎, 董嗣万, 等. 五级流水线 RISC-V 微处理器的研究与设计[J/OL]. 计算机工程: 1-8. DOI: 10.19678/j.issn.1000-3428.0068146.

[2] 计算机科学丛书: 计算机组成与设计: 硬件/软件接口[M/OL]. 机械工业出版社, 2020. https://books.google.de/books?id=YeZbzgEACAAJ.

[3] 杜岚, 王裕, 刘向峰, 等. 一种基于 RISC-V 架构的高性能嵌入式处理器设计[J/OL]. 小型微型计算机系统, 2023, 44(12): 2865-2871. DOI: 10.20009/j.cnki.21-1106/TP.2022-0253.

[4] 赵博涵. RISC-V 标量处理器的应用与优化分析[J/OL]. 集成电路应用, 2024, 41(03): 40-43. DOI: 10.19339/j.issn.1674-2583.2024.03.016.

[5] 倪光南. 以全球视野谋划和推动开源 RISC-V 生态发展[J]. 科技导报, 2024, 42(02): 1-2.

[6] 潘树朋, 刘有耀. RISC-V 微处理器以及商业 IP 的综述[J]. 单片机与嵌入式系统应用, 2020, 20(06): 5-8+12.

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,并给出百分制评分。)