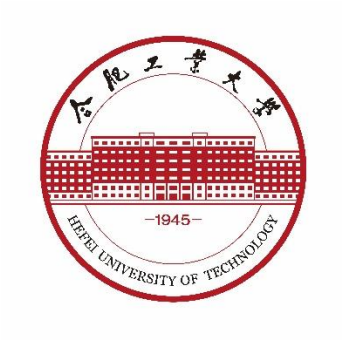
合肥工業大學



2021~2022 学年 第一学期

《系 统 硬 件 综 合 设 计》

设 计 报 告

班　级	计算机 19-2 班	学　号	2019217192
姓　名	周布伟	成　绩	

2022 年 1 月

目　录

1　方案选定与设计

本节将介绍此次《系统硬件综合设计》课程的设计目的与要求，针对设计要求，我们调研整理了三套候选方案，经过比较，我们最终确定使用 ARMv8 指令集架构作为设计蓝本，并对此制定理论学习和开发设计两阶段的实现方案。最后，根据实现方案，我们将对此次课程设计的开发环境与实现情况进行简单介绍。

1.1 设计的目的与要求

《系统硬件综合设计》基于先修课程，根据系统设计思想，使用硬件描述语言设计实现一款基于 MIPS32，ARM，RISC-V 或者自定义指令集的微处理器（CPU），以此贯穿《数字逻辑》《计算机组成原理》《计算机体系结构》课程，实现从逻辑门至完整 CPU 处理器的设计。

实现上，两人一组，完成单周期 CPU 设计，或多周期 CPU 设计，或五级流水线 CPU 设计，难度依次提升。所有学生必须至少完成单周期 CPU 的设计工作，并将设计的 CPU 下载至 FPGA 开发板（EGO-1）上运行。

1.2 方案的选定

正式课程设计开始之前，经过预先调研，如表 1-1 所示，我们将 MIPS32，ARM，RISC-V 作为候选方案，并拟采用流水线的设计方案。

MIPS32 指令集架构因为简单，曾被作为《计算机体系结构》课程的案例而重点讲解，同时，本次设计的推荐教材《自己动手写 CPU》[1]详细地阐述了 MIPS32 指令集架构各种指令的实现步骤，还提供了教学版和实践版的实现方案，随书附赠了完整的工程代码。由此，MIPS32 指令集架构的实现门槛较小，若要增加难度，需要额外考虑数据前推、乱序执行、高速缓存等优化方案。

ARM 指令集架构曾被作为《嵌入式系统原理》课程案例介绍，但并未对指令集架构本身进行讲解，仅停留在汇编指令与硬件部分。同时了解到，ARM 指令集架构经过多次迭代，衍生出很多的版本，每个版本之间可能存在较大的差异，目前

最新的版本是 ARMv8，相较于 ARMv7，其做出了较大的改变，引入 64 位架构的支持，包括 64 位和 32 位两种执行状态，并支持三个主要指令集。而《嵌入式系统原理》列举的是 ARMv4 左右的版本进行着重介绍。此外，目前发现并未有相关书籍对 ARM 指令集架构的 CPU 设计实现进行介绍，只有少数如《计算机组成与设计：硬件/软件接口》[2]对 ARM 指令集架构设计的基本原理进行系统介绍。同时，网络上虽然有部分设计代码，但由于指令集版本的不同而存在很多的差异，并且并未有具体的实现步骤作指导。由此，ARM 指令集架构具有一定的难度，还需要投入较多的时间进行理论学习。

RISC-V 指令集架构并未作为课程样例进行介绍，其是较新的指令集架构，采用模块化的设计思想，开源授权，指令数量较少，实现较为容易。目前，针对该指令集架构，有《手把手教你设计 CPU——RISC-V 处理器篇》等书籍介绍了其实现步骤，并开源了全部的设计代码。由此，RISC-V 指令集架构虽然需要一定的学习成本，但设计难度较低。

表 1-1 CPU 设计候选方案比较

候选方案	是否熟悉	相关教材	开源代码	实现难度
MIPS32	是	《自己动手写 CPU》	书籍配套，过程详细	易
ARM	是	暂无	较少，无开发指导	较难
RISC-V	否	《手把手教你设计 CPU》	书籍配套，过程详细	较易

综合考虑难度与时间开销，最终，我们确定 ARMv8 指令集架构作为《系统硬件综合设计》的实现方案。

1.3 实现思路

本次课程设计拟采用 ARMv8 指令集架构作为设计的蓝本，同时采用五级流水线，有一定的难度，且需要投入较多的时间进行理论学习和开发设计。对此，如图 1-1 所示，实现的思路如下：

第一阶段：理论学习，包括 Verilog 语言回顾、CPU 设计的实现流程、ARMv8 指令集架构的特点、开发板的使用等方面。由于本专业并未专门为 Verilog 语言开设

相应的课程，只是在《数字逻辑》和《计算机组成原理》中有过一些介绍和简单的实验，因此在正式设计之前，需要对 Verilog 语言做简单的回顾，可以通过《自己动手写 CPU》[1]中的理论部分作为参考，对于设计过程中出现的问题，可以直接通过网络搜索解决，无需做过多系统性的学习。由于先前尚未经历过 CPU 设计开发的项目，对具体的开发过程并没有一个明确的认识，故在正式设计之前，需要通过 CPU 设计实现的教材和慕课（如东南大学的《计算机系统综合设计》）来对 CPU 设计有一个直观的理解。由于本次设计面向 ARMv8，在先修的课程中并未有太多的介绍，需要通过《计算机组成与设计：硬件/软件接口》[2]学习流水线、数据通路等理论知识，在设计的过程中可以查阅手册。由于此前并未接触过 EGO-1 类型的开发板，缺乏相应的调试经验，需要对照 EGO-1 的相应例程和实验指导书进行学习。

第二阶段：开发设计，包括数据通路设计、流水线优化、开发板烧录和调试等方面。本次课程设计两人一组，为了提高开发效率，分工以并行的方式进行。首先，针对 ORR 指令，两人按照流水线的五个阶段分别实现相应阶段的硬件模块，并整合形成基本的数据通路，随后，根据指令类型，两人分别实现算术逻辑运算指令、访存指令、跳转指令等相应类型的指令，并对流水线进行优化，然后交由一人合并，最后设计演示指令和演示程序，进行开发板的烧录和调试。

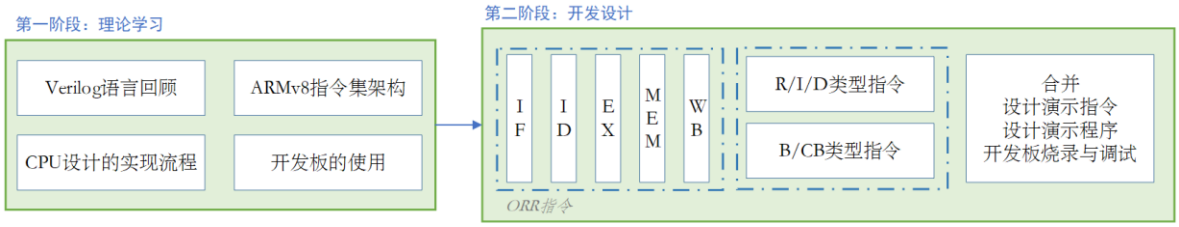


图 1-1 实现流程

1.4 开发环境与完成情况

本次课程设计采用 Xilinx 公司的 Vivado2020.1 开发环境，编写.v 文件设计硬件模块，进行顶层封装，通过电路图检查信号分配，编写 testbench 模拟仿真测试信号波形图。同时搭配依元素科技基于 Xilinx Artix-7 FPGA 研发的 EGO1 开发板展示设计效果，通过 Vivado 对.v 文件进行综合实现，根据开发板的引脚对模块输入输出的

信号进行分配，最后生成 Bitstream 文件烧录至开发板中用于演示。

本次课程设计实现了基于 ARMv8 指令集架构五级流水线 CPU 设计，其支持三十余条指令，其中包括多种算术逻辑运算指令、访存指令、跳转指令等，并通过数据前推解决真相关（Read-After-Write, RAW）可能造成的冲突，最终能够通过开发板的数码管展示寄存器堆中部分寄存器和程序计数器（Program Counter, PC）的内容。

2 相关理论概述

在第 1 节中我们将本次课程设计分为理论学习和开发设计两个阶段，本节将介绍理论学习阶段通过调研和部分实践所取得的学习成果，其中包括 Verilog 语言回顾、CPU 设计的实现流程、ARMv8 指令集架构、EGO-1 开发板的使用等内容。出于篇幅受限，本节仅总结 CPU 设计的实现流程、ARMv8 指令集架构、EGO-1 开发板的使用，且仅是与此次开发高度相关的内容，若需了解更多相关内容，可以查阅书籍或手册。

2.1 CPU 设计的实现流程

在第 1 节中，我们简单介绍了开发设计的实现思路，在本节中，我们将详细展开介绍关于开发设计部分的具体思路及其原因。通过调研相关面向 CPU 设计实践的参考文献，本次 CPU 设计的实现流程确定为由一条数据通路出发[1]，设计相关的硬件模块，然后两人分工实现不同的指令，最后交由一人汇总，并烧录至开发板调试。

2.1.1 设计第一条指令（ORR 指令）的数据通路

之所以选择 ORR 指令，是因为它是指令集架构中的常规操作，涉及数据通路较广，也较为简单，以此为基础能够完成大部分硬件的初始化，并能够很快地进行仿真测试验证。在此次设计过程中，为了提高开发效率，分工将按照流水线所划分的取指阶段（Instruction Fetch, IF）、译码阶段（Instruction Decode, ID）、执行阶段（execute, EX）、访存阶段（memory, MEM）、写回阶段（Write Back, WB），我将负责 IF、ID、EX、MEM 阶段的 ORR 指令数据通路设计，并最后汇总测试。在源码中，模块的编写者均已注明。

2.1.2 以第一条数据通路为基础，分工设计不同的指令

经过调研可知，ARMv8 指令集架构包含多种类型的指令，如 R-指令、I-指令、D-指令、B-指令、CB-指令、IW-指令等，相同指令的数据通路类似，由此，我们可以根据指令的类型进行分工。考虑到方案实现的目标与实现的难度，本次选取

ARMv8 指令集架构中 R-指令、I-指令、D-指令、B-指令、CB-指令中的子集进行设计。如表 2-1 所示，我将负责设计 R-指令、I-指令与 D-指令，包括基本算术逻辑运算指令、含立即数的算术逻辑运算指令、设置标志位的算术运算指令、设置标志位的含立即数的算术运算指令、存数取数和其他指令等在内的 21 条指令，由于指令之间的相似性，每实现一条指令，该指令所在类型的所有指令都能够很容易地实现。进一步而言，在指令数据通路设计过程中涉及到很多的控制信号，在本次设计中，每一条数据通路所涉及到的控制信号在测试时先赋固定值，控制单元和流水线寄存器信号的调整将在汇总时进行。

表 2-1 本次设计负责实现的指令

实现的指令分类	实现的指令	种类	数量
基本算术逻辑运算指令	LSR、LSL、ADD、SUB、AND、ORR、EOR	R	7
含立即数的算术逻辑运算指令	ADDI、SUBI、ANDI、ORRI、EORI	I	5
设置标志位的算术运算指令	ANDS、ADDS、SUBS	R	3
标志位、立即数的算术运算指令	ANDIS、ADDIS、SUBIS	I	3
存数取数和其他指令	LDUR、STUR、NOP	D	3

2.1.3 汇总合并，烧录至开发板并调试

该阶段由我负责，由于本次项目基于模块化的设计思想，指令的扩充与合并比较容易，实际开发过程中，因为有了第一条数据通路为基础，只需要在改动过的模块中合理标注改动内容，通过观察模块输入输出的接口种类、数量和模块改动标注，即可找到需要改动的模块，由于扩充指令，并未对原有的数据通路做过多的修改，故合并时大多情况下只需添加代码即可，无需做过多的修改。合并后还需要定义相关的控制信号，编写和扩充控制单元，与各个模块所需要的控制信号引脚进行连接，由于此次开发设计的控制信号是统一译码，并随着流水线逐级流动的，因此还需要修改流水线寄存器的输入与输出，确保每一级流水线都能有相应的控制信号。由于本人有 51 单片机开发的经验，对 LED、数码管等开发板外部设备较为熟悉，故由我进行烧录至开发板并调试，但与 51 单片机开发不同的是，此次设计中的开发板旨在反映指令执行过程中相关寄存器的变化，需要基于合并后的指令，编写演示程序，

通过执行相关指令改变寄存器内部的值，交由开发板上数码管显示。同时，开发板的 Bitstream 文件烧录前需要大量的时间进行综合与实现，为了保证开发效率与结果的准确性，在综合与实现之前需要进行仿真测试。此外，其他的相关内容我们将在 2.3 中讨论。

2.2 ARMv8 指令集架构

我们在第 1 节中确定 ARMv8 指令集架构作为《系统硬件综合设计》的实现方案，横向看，ARM 系列的指令集发展迅速，“2015 年，ARM 处理器芯片的产量超过 140 亿片，这使得 ARM 成为世界上最流行的指令集”[2]，可见其受到很多人的欢迎，并得到了广泛的应用，有一定的学习和实现的意义；纵向看，ARM 系列的指令集经过多代的更迭变得复杂，在实现过程中需要提前调研相关资料，在设计实现上也只能实现它的子集。相较前代，ARMv8 指令集架构有了很大的改动，“在我们看来，颇具讽刺意味的是，ARMv8 比 ARMv7 更接近于 MIPS”[2]，由此，值得庆幸，这对于学过 MIPS 而初学 ARM 的人更加友好。为了能够实现 ARMv8 指令集架构，我们预先进行了相关的调研，以下将从硬件支持、操作数与指令、与其他指令集之间的区别等方面着重介绍，且之后的开发设计也将基于这些理论展开，由于部分 ARM 的设计原理较为复杂，后期的开发设计对部分内容进行了简化处理，在此会进行简要的说明。

2.2.1 硬件支持

真正的 ARMv8 指令集架构存在两种执行状态，在 AArch32 下，寄存器为 32 位宽，在 AArch64 下，寄存器为 64 位宽。第一种状态支持 A32 和 T32 指令集，第二种状态支持 A64 指令集。在本次设计中，为了简化处理，将采用 64 位宽。

ARMv8 指令集架构中，含有 32 个 64 位寄存器，31 号寄存器 XZR 通过硬件连线恒置为 0，然而，31 号寄存器在绝大多数指令中表示 XZR，即恒为 0 的寄存器，而在其他指令中表示栈指针 SP，栈从高地址向低地址增长。这在设计实现过程中容易造成歧义，这种歧义不仅会令用户混淆，还会增加数据通路设计的复杂性。对此，本次设计中，将 31 号寄存器总是与 XZR 一致，而栈指针 SP 另外指定寄存器。此外，在过程调用中，X0~X7 作为八个参数寄存器，用于传递参数或返回结果，X30

作为返回地址寄存器，用于返回原始调用点。

2.2.2 操作数与指令

ARMv8 指令集架构按照字节寻址，视 32 位为字，64 位双字。既支持大端模式，也支持小端模式，还提供了大小端互换的指令，由于只有在考虑以双字方式还是以字节方式访问同一个数据时，“端”的顺序才起作用，其他多数情况下并不需要关注该问题，对此，为了简化处理，本次设计中的指令存储器和数据存储器在存储时统一采用大端的模式。此外，由于需要设计五级流水线，为了避免指令和数据访问造成的冲突，本次设计采用哈佛架构，即将指令和数据通过两个存储器分别存储。

ARMv8 架构对于大多数数据传输指令，支持对普通存储器的非对齐访问，但栈访问和取指令必须遵守对齐限制。对此，本次设计统一采用对齐访问的方式。

ARMv8 架构采用二进制补码表示有符号数，与一般的指令集架构相同，在此次实际开发实现时交由开发平台实现，从结果可以看出，进行相关运算时，均采用补码的方式进行，且结果正确。

ARMv8 指令集架构主要采用以下 4 种寻址方式，这些寻址方式将直接体现在指令中，并通过数据通路予以实现。

- 立即数寻址，操作数是位于指令自身中的常数；
- 寄存器寻址，操作数是寄存器；
- 基址寻址或偏移寻址，操作数在内存中，其地址是一个寄存器和指令中常数的和；
- PC 相对寻址，地址是 PC 与指令中常数的和。

ARMv8 采用 32 位指令，并固定长度编码，这样做能够降低译码复杂度，通过不同格式的指令，能够完成不同类型的操作，其中 load 和 store 是访问内存的唯一途径。本设计中，涉及到三种指令格式，即 R-指令、I-指令、D-指令，分别实现算术逻辑运算、带立即数的算术逻辑运算、存数取数操作，在之后的开发设计中将对这三种指令进行实现。如图 2-1 所示，opcode 表示操作码，Rm 表示第二个源操作数寄存器，shamt 表示位移量，Rn 表示第一个源操作数寄存器，Rd 表示目的操作数寄存器，存放操作结果，ALU_immediate 表示立即数，DT_address 表示地址字段，op 表示操作码字段扩展。三种指令均为 32 位宽，有公共的 Rn 和 Rd 部分。三种指令格式由第一个字段的值来区分，尽管长度存在一定的差异，但每种格式的 opcode

都赋给不同的值，所在的位宽范围内均是唯一确定的，以便让计算机硬件知道如何处理指令的剩余部分。

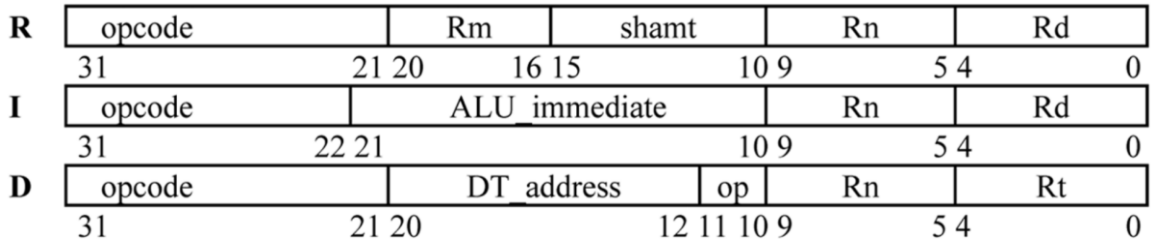


图 2-1 ARMv8 指令集架构的部分指令格式[2]

真正的 ARMv8 指令集没有采用助记符 ADDI 表示立即数加，而是和普通加法一样用 ADD 表示，由汇编器负责选择正确的操作码，用相同的助记符表示不同的操作可能会引起混淆，因此为了便于区分，在本次报告中用不同的助记符对这两种加法进行表示。

2.2.3 与其他指令集的对比

在部分先修课程中，我们曾经接触过 MIPS、x86、ARMv4 等指令集，为了能够在设计过程中融汇贯通，减少错误的发生，我们对其他指令集也进行了相关的调研，并将 ARMv8 与其他指令集进行简单地对比，以下将列举 MIPS、ARMv7 指令集的调研部分结果。

MIPS 因为简单，而被作为很多教材介绍指令集架构的例子，“MIPS 和 ARMv8 设计理念相同，但 MIPS 的问世要比 ARMv8 早 25 年。”[2] MIPS 指令集提供 32 位地址和 64 位地址两个版本，两种指令集几乎完全相同，区别地址空间的位宽。

- MIPS 和 ARMv8 两种体系结构的指令都是 32 位的，都含有 32 个通用寄存器，并且其中一个寄存器的值由硬件恒置为 0。
- load 和 store 指令都是访问内存的唯一途径。
- 与 MIPS 的区别之一是条件分支指令，ARMv8 指令集采用条件码，而 MIPS 采用比较指令的方式，根据比较结果而将寄存器的值进行设置，然后根据寄存器的数值进行分支跳转。
- 在指令格式上，MIPS 支持的指令格式较 ARMv8 更少。

ARMv7 指令集是 ARMv8 指令集的先序版本，在第 1 节也曾提到，ARMv8 指

令集相较 ARMv7 有了较大的改变，以下将主要阐述两者之间的区别。

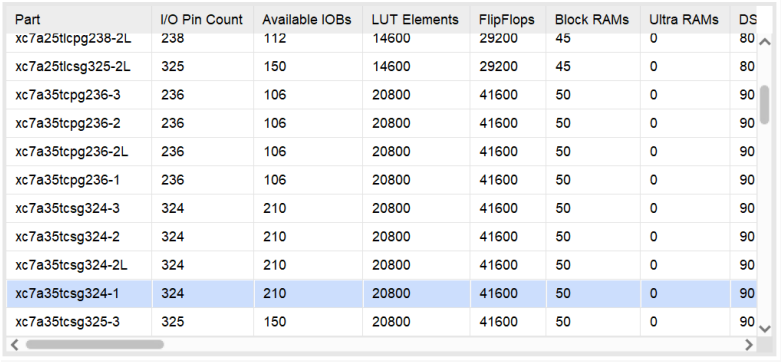
- ARMv7 指令集都只有 15 个通用寄存器，这增加了编译器的开发难度，同时，寄存器中并没有将某个寄存器硬件置 0，因而需要额外的指令才能完成 ARMv8 直接通过 XZR 寄存器就能完成的操作。
- ARMv7 有取多个数和存多个数的指令，即允许几个寄存器同时从内存中移入移出数据，而 ARMv8 没有。
- ARMv7 寻址方式并不适用于所有的数据大小，而 ARMv8 可以。
- ARMv7 将 PC 划分到了寄存器堆里，在实际操作中需要额外对寄存器进行判断，否则会遇到不可预料的分支，而在 ARMv8 中的 PC 不是 32 个寄存器之一，而被单独拿出。

从 ARMv8 和 MIPS、ARMv7 的比较中不难看出，ARMv8 的这次改动，使得其与 MIPS 更加类似，这样印证了“ARMv8 比 ARMv7 更接近于 MIPS”[2]这句话。同时，这也为实际的开发减少了一定的难度。

2.3 EGO-1 开发板的使用

本次设计的最后会将各个硬件模块和基于设计指令的程序通过综合实现，然后烧录至开发板中，以展示设计指令的执行情况。对于开发板，以下将介绍实际开发过程中的注意事项，开发的详细过程将在第 3.1 节中阐述。

本次开发使用的开发板是依元素科技基于 Xilinx Artix-7 FPGA 研发的 EGO1 开发板。如图 2-2 所示，在新建项目时需要选择 xc7a35tcsg324-1 类型的开发板。编程开发过程中，需要逐步设计模块，并进行顶层封装，封装的引脚需要映射为实际开发板上的引脚信号，对此可以查阅依元素科技提供的 EGO-1 参考手册[4]。



Part	I/O Pin Count	Available IOBs	LUT Elements	FlipFlops	Block RAMs	Ultra RAMs	DS
xc7a25tlcpg238-2L	238	112	14600	29200	45	0	80
xc7a25tlcsg325-2L	325	150	14600	29200	45	0	80
xc7a35tcpg236-3	236	106	20800	41600	50	0	90
xc7a35tcpg236-2	236	106	20800	41600	50	0	90
xc7a35tcpg236-2L	236	106	20800	41600	50	0	90
xc7a35tcpg236-1	236	106	20800	41600	50	0	90
xc7a35tcsg324-3	324	210	20800	41600	50	0	90
xc7a35tcsg324-2	324	210	20800	41600	50	0	90
xc7a35tcsg324-2L	324	210	20800	41600	50	0	90
xc7a35tcsg324-1	324	210	20800	41600	50	0	90
xc7a35tcsg325-3	325	150	20800	41600	50	0	90

图 2-2 开发板型号的选择

开发板只提供了 100M 的晶振，如果直接作为各个模块的时钟输入，将会以一个非常快的速度执行指令，对此可以加入分频器模块或将按键作为时钟的输入。

本次开发设计需要显示不同寄存器的内容，数码管需要采用动态显示的策略，利用人眼的视觉暂留现象，快速位选扫描不同的数码管，同时显示不同的数值，即可在同一时刻看到不同寄存器的内容。此外，数码管并不是输入相应的数值就能显示内容，而需要根据数值点亮数码管上的段号，本次开发板上给的是八段数码管，其中包含一个小数点，故在将数值输入数码管之前需要进行译码，并输出译码段选信号和位选信号。

3　指令设计与流水线优化

本节主要在第 2 节调研学习的基础上，介绍此次课程设计的重点，主要分为两个方面：指令设计、流水线优化。其中，指令设计将分别从流水线各阶段涉及的模块和每一条指令的实现两个维度进行介绍，由浅入深。流水线优化主要涉及用数据前推解决真相关引起的冒险。

3.1 数据通路中的模块设计

本次设计将要实现基于 ARMv8 架构的五级流水线 CPU，在设计开发的过程中，我们采用了分而治之的模块化设计思想设计各个模块，为了便于说明各指令的数据通路，以下将先介绍几个最主要的数据通路模块，其中，由于各个阶段的流水线寄存器实现的功能类似，只是输入输出的引脚信号的区别，限于篇幅，以下将不再介绍。

3.1.1　取指阶段模块设计

取指模块主要包括 1 个程序计数器和 1 个指令存储器，其相互关系与引脚定义分别如图 3-1 以及表 3-1 所示，指令存储器只提供读访问，任意时刻的输出都反映了输入地址指向的内容，不需要读控制信号；程序计数器（PC）是 64 位寄存器，每个时钟周期结束都会被写入，不需要写控制信号；为了便于实现，PC 内部还集成有加法器，总是将两个输入相加。

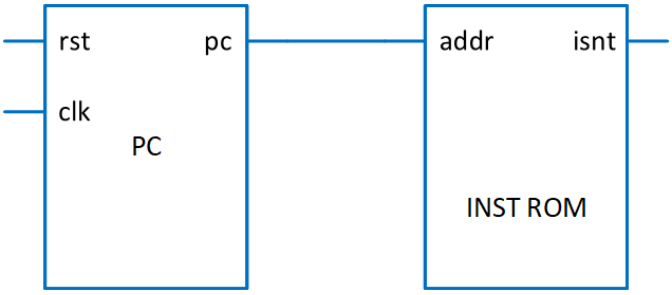


图 3-1 取指阶段内主要模块及其相互关系

表 3-1 取指阶段各个模块的引脚定义

模块	输入/输出	接口名称	宽度	作用
PC	输入	rst	1	复位信号
	输入	clk	1	时钟信号
	输出	pc	64	要读取的指令地址
INST ROM	输入	addr	64	要读取的指令地址
	输出	inst	32	读出的指令

程序计数器 PC 模块主要进行程序指令的计数，用于控制指令的执行。其代码如下，需要说明的是，为了使代码具有更好的可扩展性且便于修改，模块中的部分信号通过宏定义的方式实现，由于工程代码是由两人共同完成，每个模块中会有个别引脚与功能实现不在此次介绍的范围内。在 PC 模块中，主要实现每一个时钟信号后能够使计数加 1，由于一条指令是 4 个字节，故在此将计数器以此加 4。

```
module pc_reg(

    input    wire                          clk,          // 时钟信号
    input    wire                          rst,          // 复位信号

    input wire PCSrc,   // 增加
    input wire[`InstAddrBus] pc_in,

    output   reg      [`InstAddrBus]       pc            // 要读取的指令地址

);

    always @ (posedge clk) begin
        if (rst == `RstEnable) begin
            pc <= 64'h0000000000000000;
        end else begin
            if (PCSrc == 1'b1) begin
                pc <= pc_in;
            end
            else begin
                pc <= pc + 4;
            end
        end
    end
```

```
endmodule
```

指令存储器 INST ROM 主要用于存储需要执行的指令，根据 PC 模块输出的 PC 信号进行取指并输出，以下存储按照字节寻址，并采用了大端的方式。初始化的指令将通过系统函数 readmemh 从文件中读取。

```
module inst_rom(

    input   wire                        rst,            // 复位信号，高电平有效
    input   wire    [`InstAddrBus]      addr,   // 要读取的指令地址
    output reg      [`InstBus]          inst    // 读出的指令

);

    reg[`BYTESIZE-1:0]  inst_mem[0:`InstMemSize-1];

    initial $readmemh (`INSTFILE, inst_mem );

    always @ (*) begin
        if (rst == `RstEnable) begin
            inst <= `InstZeroWord;
        end else begin
            inst[7:0]   = inst_mem[addr + 3];
            inst[15:8]  = inst_mem[addr + 2];
            inst[23:16] = inst_mem[addr + 1];
            inst[31:24] = inst_mem[addr];
        end
    end

endmodule
```

3.1.2 译码阶段模块设计

译码模块主要包括 1 个寄存器堆和 1 个符号扩展器，此外还有控制单元与选择器，由于控制单元涉及全部数据通路的信号，比较器实现较为简单，这两个模块将在全局原理图中予以给出。译码模块的相互关系与引脚定义分别如图 3-2 和表 3-2 所示，寄存器堆是译码阶段中十分重要的模块，内部包含了 32 个 64 位寄存器，主要保存了有关的数据；符号扩展器用于处理带立即数的算术逻辑运算的指令或跳转指令等。二者通过取指阶段读取的指令进行关联。

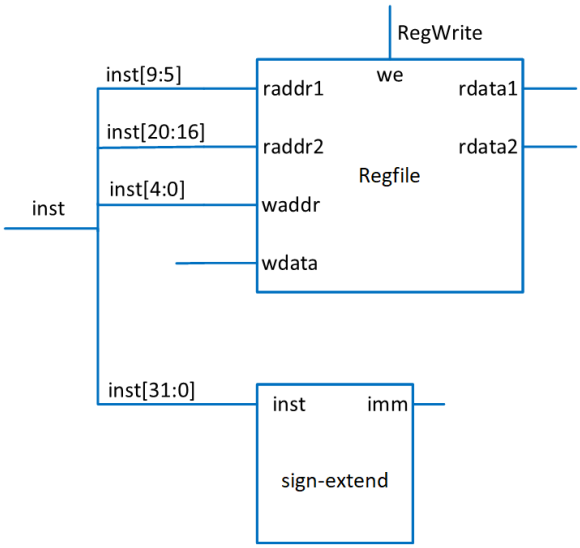


图 3-2 译码阶段内主要模块及其相互关系

表 3-2 译码阶段各个模块的引脚定义

模块	输入/输出	接口名称	宽度	作用
Regfile	输入	rst	1	复位信号，高电平有效
	输入	clk	1	时钟信号
	输入	raddr1	5	第一个读寄存器端口要读取的寄存器的地址
	输出	rdata1	64	第一个读寄存器端口输出的寄存器值
	输入	raddr2	5	第二个读寄存器端口要读取的寄存器的地址
	输出	rdata2	64	第二个读寄存器端口输出的寄存器值
	输入	we	1	写使能信号
	输入	waddr	5	要写入的寄存器地址
	输入	wdata	64	要写入的数据
sign_extend	输入	inst	32	读出的指令
	输出	imm	64	符号扩展后的指令字段

寄存器堆 Regfile 模块负责数据的读取和写入，内部的 32 个 64 位寄存器中，故通过 5 位二进制数即可唯一确定寄存器堆内的寄存器，其中，31 号寄存器通过硬件恒置为 0。如果要读出一个数据字，要输入需要读的寄存器号，无需控制信号，最终会输出一个读出 64 位的值；如果要写入一个数据字，需要输入两个要写的寄存器

号、要写的数据和写控制信号控制，写操作在时钟边沿产生，为了避免数据相关造成的部分冒险，本寄存器堆在时钟上升沿写入，下降沿读取。相关代码如下，其中输出信号 o1-o4 用于之后的开发板演示，寄存器堆内的寄存器内容初始化为寄存器编号，第 31 号寄存器初始化为 0。

```
module regfile(

    input   wire                        clk,        // 复位信号，高电平有效
    input   wire                        rst,        // 时钟信号

    //读端口 1
    input   wire    [`RegAddrBus]   raddr1,// 第一个读寄存器端口要读取的寄存器的地址
    output reg      [`RegBus]       rdata1,     // 第一个读寄存器端口输出的寄存器值

    //读端口 2
    input   wire    [`RegAddrBus]   raddr2,// 第二个读寄存器端口要读取的寄存器的地址
    output reg      [`RegBus]       rdata2,     // 第二个读寄存器端口输出的寄存器值

    //写端口
    input   wire                        we,         // 写使能信号
    input   wire    [`RegAddrBus]   waddr,      // 要写入的寄存器地址
    input   wire    [`RegBus]       wdata,      // 要写入的数据

    output wire     [`DataBus]      o1,
    output wire     [`DataBus]      o2,
    output wire     [`DataBus]      o3,
    output wire     [`DataBus]      o4

);

    reg[`RegBus]    regs[0:`RegNum-1];

    integer initCount;

    assign   o1 = regs[0];
    assign   o2 = regs[2];
    assign   o3 = regs[3];
    assign   o4 = regs[4];

    initial begin

        for (initCount = 0; initCount < `RegNum; initCount = initCount + 1) begin
            regs[initCount] = initCount;
        end
```

```
        regs[31] = 64'h00000000;
    end

    // 读端口 1 操作
    always @ (negedge clk) begin
        if(rst == `RstEnable) begin
            rdata1 <= `ZeroWord;
        // 31 号寄存器的值始终是数值 0
        end else if(raddr1 == `RegNumLog2'd31) begin
            rdata1 <= `ZeroWord;
        end else begin
            rdata1 <= regs[raddr1];
        end
    end

    // 读端口 2 操作
    always @ (negedge clk) begin
        if(rst == `RstEnable) begin
            rdata2 <= `ZeroWord;
        // 31 号寄存器的值始终是数值 0
        end else if(raddr2 == `RegNumLog2'd31) begin
            rdata2 <= `ZeroWord;
        end else begin
            rdata2 <= regs[raddr2];
        end
    end

    // 写端口操作
    always @ (posedge clk) begin
        if((we == `WriteEnable) && (waddr != `RegNumLog2'd31)) begin
            regs[waddr] <= wdata;
        end
    end

endmodule
```

符号扩展器主要用于将指令中的带符号偏移字段符号扩展为 64 位的带符号值，输入 32 位的指令，输出指令中的立即数字段扩展后的 64 位数据。相关代码如下，其中包括了 I-指令和 D-指令。通过识别指令中的操作码，即可对指令中的立即数进行扩展，这里直接利用大括号{}进行扩展。

```
module sign_extend(

    input wire[`InstBus] inst,
```

```
    output reg [`ImmBus] imm

    );

    always @ (*) begin
        imm[`ImmBus] = `ZeroWord;
        if (inst[31:26] == 6'b000101) begin                                    // B
            imm[25:0] = inst[25:0];
            imm[63:26] = {64{imm[25]}};
        end else if (inst[31:24] == 8'b10110100) begin                         // CBZ
            imm[19:0] = inst[23:5];
            imm[63:20] = {64{imm[19]}};
        end else if (inst[31:24] == 8'b10110101) begin                         // CBNZ
            imm[19:0] = inst[23:5];
            imm[63:20] = {64{imm[19]}};
        end else begin                                                         // I/D
            case (inst[31:21])
                11'b11111000010 : imm = {{55{inst[20]}}, inst[20:12]};       // LDUR
                11'b11111000000 : imm = {{55{inst[20]}}, inst[20:12]};       // STUR
            endcase
            case (inst[31:22])
                10'b1001000100 : imm = {52'b0, inst[21:10]};       // ADDI
                10'b1001001000 : imm = {52'b0, inst[21:10]};       // ANDI
                10'b1011001000 : imm = {52'b0, inst[21:10]};       // ORRI
                10'b1101000100 : imm = {52'b0, inst[21:10]};       // SUBI
                10'b1101001000 : imm = {52'b0, inst[21:10]};       // EORI
                10'b1011000100 : imm = {52'b0, inst[21:10]};       // ADDIS
                10'b1111000100 : imm = {52'b0, inst[21:10]};       // SUBIS
                10'b1111001000 : imm = {52'b0, inst[21:10]};       // ANDIS
            endcase
        end
    end
endmodule
```

3.1.3 执行阶段模块设计

执行模块主要包括 2 个算术逻辑运算单元（Arithmetic and Logic Unit, ALU）和 1 个算术逻辑运算控制单元，其中 1 个运算器用于跳转指令地址的计算，此外还有比较器。执行模块的相互关系与引脚定义分别如图 3-3 和表 3-3 所示，ALU 负责处理算术逻辑运算指令，包括普通、带立即数、带符号位设置的指令；ALU 由 ALU 控制单元控制，它与译码阶段的主控制单元一同构成多级译码的形式，即主控制单元生成 ALUOp 作为 ALU 控制单元的输入，再由 ALU 控制单元生成真 ALU 控制信

号，通过多级控制能够减小主控制单元的规模，使用多个控制单元也可能减少控制单元的延迟。由于设计的指令涉及到对标志位的操作，ALU 控制单元识别指令后将输出控制信号和单独的控制信号 setflags 用于设置 ALU 的运算以及对标志位的影响。

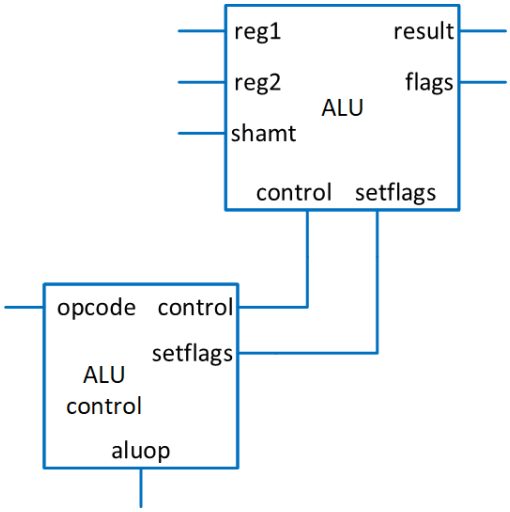


图 3-3 执行阶段内主要模块及其相互关系

表 3-3 执行阶段各个模块的引脚定义

模块	输入/输出	接口名称	宽度	作用
ALU	输入	rst	1	复位信号，高电平有效
	输入	reg1	64	参与运算的源操作数 1
	输入	reg2	64	参与运算的源操作数 2
	输入	control	4	ALU 控制信号
	输入	shamt	6	移位量
	输入	setflags	1	是否设置标志位
	输出	result	64	ALU 运算结果
	输出	flags	4	结果为 0 标志位
ALU control	输入	opcode	11	操作码字段，inst[31:21]
	输入	aluop	2	ALU control 控制信号
	输出	setflags	1	是否设置标志位
	输出	control	4	ALU 控制信号

算术逻辑运算控制单元通过主控制单元所给的 ALUop 信号并根据指令的操作码输出对 ALU 的控制信号，在实际实现时主要涉及到对输入信号的判断，尽管输入的操作码可能长短不一，但每一个操作码在对应位宽下都是唯一确定的，故可以直接根据操作码输出相应的控制信号。各个指令对应的输出信号如表 3-4 所示。

```
module alu_control(

    input       wire [`AluOpBus]        aluop,
    input       wire [`OpCodeBus]       opcode,
    output      reg    [`ALUCtlBus]      control,
    output      reg                      setflags

);

    always @(aluop or opcode) begin
        setflags = 1'b0;
        control = 4'b1111;
        case (aluop)
            2'b00 : control = 4'b0010;                                  // ADD
            2'b01 : begin control = 4'b0111; setflags = 1'b1; end                                  // PASS B
            2'b10 : begin
                case (opcode)
                    11'b10001011000 : control = 4'b0010;          // ADD
                    11'b10101011000 : begin control = 4'b0010; setflags = 1'b1; end        // ADDS
                    11'b11001011000 : control = 4'b0110;          // SUB
                    11'b11101011000 : begin control = 4'b0110; setflags = 1'b1; end        // SUBS
                    11'b10001010000 : control = 4'b0000;          // AND
                    11'b11101010000 : begin control = 4'b0000; setflags = 1'b1; end        // ANDS
                    11'b10101010000 : control = 4'b0001;          // ORR
                    11'b11001010000 : control = 4'b1000;          // EOR
                    11'b11010011010 : control = 4'b1010;          // LSR
                    11'b11010011011 : control = 4'b1011;          // LSL
                endcase
                case (opcode[10:1])
                    10'b1001000100 : control = 4'b0010;          // ADDI
                    10'b1001001000 : control = 4'b0000;          // ANDI
                    10'b1011001000 : control = 4'b0001;          // ORRI
                    10'b1101000100 : control = 4'b0110;          // SUBI
                    10'b1101001000 : control = 4'b1000;          // EORI
                    10'b1011000100 : begin control = 4'b0010; setflags = 1'b1; end        // ADDIS
                    10'b1111000100 : begin control = 4'b0110; setflags = 1'b1; end        // SUBIS
                    10'b1111001000 : begin control = 4'b0000; setflags = 1'b1; end        // ANDIS
                endcase
            end
```

```
        endcase

    endmodule
```

表 3-4 各个指令对应的输出信号

指令	含义	control 信号	setflags 信号
LDUR / STUR	取数 / 存数	0010	
ADD	加	0010	
SUB	减	0110	
AND	与	0000	
ORR	或	0001	
EOR	异或	1000	
LSR	逻辑右移	1010	
LSL	逻辑左移	1011	
ADDS	带标志位加	0010	√
SUBS	带标志位减	0110	√
ANDS	带标志位与	0000	√
ADDI	带立即数加	0010	
SUBI	带立即数减	0110	
ANDI	带立即数与	0000	
ORRI	带立即数或	0001	
EORI	带立即数异或	1000	
ADDIS	带立即数和标志位加	0010	√
SUBIS	带立即数和标志位减	0110	√
ANDIS	带立即数和标志位与	0000	√

算术逻辑运算单元根据算术逻辑运算控制单元所输出的控制信号对操作数进行相关的运算，在运算的过程中记录运算的进位和结果，并根据控制信号输出相应的结果和标志位，其中标志位包括负数、零、溢出、进位，负数通过结果的最高位即可判断，零根据结果也可以很方便判断，溢出的判断依据是输入的两个操作数的符号与计算所得结果的符号不同，进位通过{carrybit, result}的形式获取结果溢出的位，若置 1 表示进位或者借位。标志位的设置需要在 setflags 控制信号置 1 的情况下才能实现。

```
module alu(

    input    wire    [`RegBus]        reg1_i,
    input    wire    [`RegBus]        reg2_i,
    input    wire    [`ShamtBus]      shamt,
    input    wire    [`ALUCtlBus]     control,
    input    wire                     setflags,
```

```
    output    reg       [`RegBus]       result,
    output    reg       [`FlagBus]      flags              //NZVC

);

    reg carrybit;

    always @(reg1_i or reg2_i or control) begin
      case (control)
        4'b0000 : {carrybit, result} = reg1_i & reg2_i;
        4'b0001 : {carrybit, result} = reg1_i | reg2_i;
        4'b0010 : {carrybit, result} = reg1_i + reg2_i;
        4'b0110 : {carrybit, result} = reg1_i - reg2_i;
        4'b0111 : {carrybit, result} = reg2_i;
        4'b1000 : {carrybit, result} = reg1_i ^ reg2_i;
        4'b1100 : {carrybit, result} = ~(reg1_i | reg2_i);
        4'b1010 : {carrybit, result} = reg1_i >>> shamt;
        4'b1011 : {carrybit, result} = reg1_i << shamt;
      endcase
    end

    always @(*) begin
      if (setflags == `True_v) begin
          // 结果为负数
          flags[3] <= result[`DWORDSIZE-1];
          // 结果为 0
          flags[2] <= result == 64'b0;
          // 结果溢出
          flags[1] <= (reg1_i[`DWORDSIZE-1] == reg2_i[`DWORDSIZE-1]) &&
(result[`DWORDSIZE-1] != reg1_i[`DWORDSIZE-1]);
          // 结果进位
          flags[0] <= carrybit;
      end else begin
          flags <= 4'b0000;
      end
    end

endmodule
```

3.1.4 访存阶段模块设计

访存阶段主要包括数据存储器，用于存储更多的数据，其图示与引脚定义分别如图 3-4 和表 3-5 所示，该模块将根据输入的信号对数据进行读写操作。

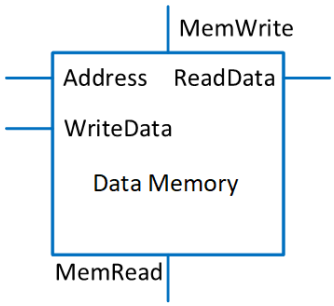


图 3-4 访存阶段的主要模块

表 3-5 访存阶段主要模块的引脚定义

模块	输入/输出	接口名称	宽度	作用
	输入	rst	1	复位信号，高电平有效
	输入	clk	1	时钟信号
	输入	addr	64	存储器的读写地址
DATA ROM	输入	wdata	64	要写入的数据
	输入	rden	1	读取使能
	输入	wren	1	写入使能
	输出	rdata	64	读取输出的数据

数据存储器的实现与指令存储器类似，只是输入输出引脚与其相比更少，在此同样采用按字节寻址和大端的方式。初始化通过 readmemh 系统函数实现。

```
module data_rom(

    input   wire                    rst,            // 复位信号，高电平有效
    input   wire                    clk,            // 时钟信号
    input   wire [`DataAddrBus]     addr,           // 存储器的读写地址
    input   wire [`DataBus]         wdata,          // 要写入的数据
    input   wire                    rden,           // 读取使能
    input   wire                    wren,           // 写入使能

    output reg [`DataBus]           rdata           // 读取输出的数据

    );

    reg [`BYTESIZE-1:0] data[0:`DataMemSize-1];
```

```
    initial begin
        $readmemh(`DATAFILE, data);
    end

    always @(posedge clk) begin
        if (rst == `RstEnable)
            rdata <= {`DWORDSIZE{1'b0}};
        else if (rden == `ReadEnable)
        begin
            rdata[63:56]    <=  data[addr]      ;
            rdata[55:48]    <=  data[addr + 1];
            rdata[47:40]    <=  data[addr + 2];
            rdata[39:32]    <=  data[addr + 3];
            rdata[31:24]    <=  data[addr + 4];
            rdata[23:16]    <=  data[addr + 5];
            rdata[15:8]     <=  data[addr + 6];
            rdata[7:0]      <=  data[addr + 7];
        end
    end

    always @(posedge clk) begin
        if (rst == `RstEnable)
            $readmemh(`DATAFILE, data);
        else if (wren == `WriteEnable)
        begin
            data[addr]          <=  wdata[63:56];
            data[addr + 1]      <=  wdata[55:48];
            data[addr + 2]      <=  wdata[47:40];
            data[addr + 3]      <=  wdata[39:32];
            data[addr + 4]      <=  wdata[31:24];
            data[addr + 5]      <=  wdata[23:16];
            data[addr + 6]      <=  wdata[15:8];
            data[addr + 7]      <=  wdata[7:0];
        end
    end

endmodule
```

3.1.5 写回阶段模块设计

写回阶段主要将结果写回寄存器堆，该阶段涉及的主要模块是选择器，由于实现较为简单，在此不再提及，相关的模块和数据通路将在后续的原理图中介绍。

3.1.6 顶层封装模块设计

为了使结构更加清晰，如图 3-5 所示，我们在普通模块的基础之上，根据流水线五个阶段，做了一层封装，由于写回模块较为简单，在此并未进行封装，此外还涉及 forwarding 模块，这将在后续流水线优化中介绍。限于篇幅，其他相关的模块可以参见附录一：各个主要模块的引脚定义，顶层封装的代码如下，其他模块的实现并不困难，在此不再给出，具体代码实现可以查看源码。数据通路可以参见附录二：主要的数据通路，其中包括了之后流水线优化所涉及的模块。

图 3-5 模块设计的结构

```
module datapath(

    input wire     clock,
    input wire     reset,

    output   wire          [`InstAddrBus]     pc,
    output   wire          [`DataBus]         o1,
    output   wire          [`DataBus]         o2,
    output   wire          [`DataBus]         o3,
    output   wire          [`DataBus]         o4

    );


    wire    [`InstAddrBus]   id_pc;              // 译码阶段取得的指令对应的地址
    wire    [`InstBus]       id_inst;           // 译码阶段取得的指令

    assign   pc = id_pc;

    // 增加
```

```
        wire    [`InstAddrBus]   mem_add_o;
        // 增加
        wire      [`FlagBus]                mem_flags_o;
        wire mem_isZeroBranch;
        wire mem_isUnconBranch;
        wire mem_isNZBranch;
        // 增加
        wire[`InstAddrBus] ex_pc_o;
        wire ex_isZeroBranch;
        wire ex_isUnconBranch;
        wire ex_isNZBranch;

        module_if module_if0(
            .clock(clock),
            .reset(reset),
            .if_pc_i(mem_add_o),
            .PCSrc(mem_isUnconBranch || (mem_flags_o[2] && mem_isZeroBranch) ||
(!mem_flags_o[2] && mem_isNZBranch)),
            .id_pc_o(id_pc),
            .id_inst_o(id_inst)
        );

        wire   [`OpCodeBus]          ex_opcode_o;
        wire   [`RegAddrBus]         ex_waddr_o;
        wire   [`RegBus]             ex_reg1_o;
        wire   [`RegBus]             ex_reg2_o;
        wire   [`ShamtBus]           ex_shamt_o;
        wire   [`ImmBus]             ex_imm_o;

        wire    [`RegAddrBus]        wb_waddr;
        wire    [`DataBus]           wb_out;

        wire                         ex_MemRead_o;
        wire                         ex_MemtoReg_o;
        wire                         ex_MemWrite_o;
        wire                         ex_ALUSrc_o;
        wire                         wb_RegWrite_o ;
        wire                         ex_RegWrite_o;
        wire    [`AluOpBus]          ex_ALUOp_o;
        wire    [`RegAddrBus]        ex_Rn_o;
        wire    [`RegAddrBus]        ex_Rm_o;

        module_id module_id0(
            .clock(clock),
            .reset(reset),
```

```
        .id_pc_i(id_pc),
        .id_inst_i(id_inst),
        .RegWrite_i(wb_RegWrite_o),
        .waddr(wb_waddr),
        .wdata(wb_out),

        .ex_opcode_o(ex_opcode_o),
        .ex_waddr_o(ex_waddr_o),
        .ex_reg1_o(ex_reg1_o),
        .ex_reg2_o(ex_reg2_o),
        .ex_shamt_o(ex_shamt_o),
        .ex_imm_o(ex_imm_o),
        .ex_MemRead_o(ex_MemRead_o),
        .ex_MemtoReg_o(ex_MemtoReg_o),
        .ex_MemWrite_o(ex_MemWrite_o),
        .ex_ALUSrc_o(ex_ALUSrc_o),
        .ex_RegWrite_o(ex_RegWrite_o),
        .ex_ALUOp_o(ex_ALUOp_o),
        .ex_Rn(ex_Rn_o),
        .ex_Rm(ex_Rm_o),

        .ex_pc_o(ex_pc_o),
        .ex_control_isZeroBranch(ex_isZeroBranch),
        .ex_control_isUnconBranch(ex_isUnconBranch),
        .ex_control_isNZBranch(ex_isNZBranch),

        .o1(o1),
        .o2(o2),
        .o3(o3),
        .o4(o4)
    );

    // output
    wire    [`RegBus]           mem_result_o;
    wire    [`RegBus]           mem_reg2_o;
    wire    [`RegAddrBus]       mem_waddr_o;
    wire                        mem_MemRead_o;
    wire                        mem_MemtoReg_o;
    wire                        mem_MemWrite_o;
    wire                        mem_RegWrite_o;

    wire    [`ForwardBus]       Fa;
    wire    [`ForwardBus]       Fb;

    module_ex module_ex0(
```

```
        .clock(clock),
        .reset(reset),
        .ex_opcode_i(ex_opcode_o),
        .ex_waddr_i(ex_waddr_o),
        .ex_reg1_i(ex_reg1_o),
        .ex_reg2_i(ex_reg2_o),
        .ex_shamt_i(ex_shamt_o),
        .ex_imm_i(ex_imm_o),
        .ex_MemRead_i(ex_MemRead_o),
        .ex_MemtoReg_i(ex_MemtoReg_o),
        .ex_MemWrite_i(ex_MemWrite_o),
        .ex_ALUSrc_i(ex_ALUSrc_o),
        .ex_RegWrite_i(ex_RegWrite_o),
        .ex_ALUOp_i(ex_ALUOp_o),
        .f_Fa(Fa),
        .f_Fb(Fb),
        .f_ALURes(mem_result_o),
        .f_Out(wb_out),

        .ex_pc_i(ex_pc_o),
        .ex_isZeroBranch_i(ex_isZeroBranch),
        .ex_isUnconBranch_i(ex_isUnconBranch),
        .ex_isNZBranch_i(ex_isNZBranch),

        .mem_flags_o(mem_flags_o),
        .mem_result_o(mem_result_o),
        .mem_reg2_o(mem_reg2_o),
        .mem_waddr_o(mem_waddr_o),
        .mem_MemRead_o(mem_MemRead_o),
        .mem_MemtoReg_o(mem_MemtoReg_o),
        .mem_MemWrite_o(mem_MemWrite_o),
        .mem_RegWrite_o(mem_RegWrite_o),

        .mem_add_o(mem_add_o),
        .mem_isZeroBranch_o(mem_isZeroBranch),
        .mem_isUnconBranch_o(mem_isUnconBranch),
        .mem_isNZBranch_o(mem_isNZBranch)

    );

    wire    [`DataBus]          wb_rdata;           // 写回阶段数据存储器读取输出的数据
    wire    [`RegBus]           wb_result;          // 写回阶段的 ALU 运算结果
    wire                        wb_MemtoReg_o ;

    module_mem module_mem0(
```

```
        reset,
        clock,
        mem_result_o,
        mem_reg2_o,
        mem_waddr_o,
        mem_result_o,
        mem_MemRead_o,
        mem_MemtoReg_o,
        mem_MemWrite_o,
        mem_RegWrite_o,

        wb_rdata,              // 写回阶段数据存储器读取输出的数据
        wb_result,             // 写回阶段的 ALU 运算结果
        wb_waddr,              // 写回阶段的指令要写入的目的寄存器地址，inst[4:0]
        wb_MemtoReg_o,
        wb_RegWrite_o
      );

    Mux mux0(
        wb_rdata,
        wb_result,
        wb_MemtoReg_o,
        wb_out
    );

    forwarding forwarding0(
        .Rn(ex_Rn_o),
        .Rm(ex_Rm_o),
        .ex_Rd(mem_waddr_o),
        .mem_Rd(wb_waddr),
        .ex_RegWrite(mem_RegWrite_o),
        .mem_RegWrite(wb_RegWrite_o),

        .Fa(Fa),
        .Fb(Fb)
        );

endmodule
```

3.2 指令的实现

至此，我们已经得到了各个阶段的硬件模块，并实现了数据通路，接下来，将

通过编写与各种类型指令对应的程序以对各个硬件模块进行检验，分别为算术逻辑运算指令、带立即数的算术逻辑运算指令、带立即数和标志位的算术逻辑运算指令、存数取数指令等。以下将通过将机器码转换为对应的十六进制数，录入文件作为指令存储器的初始值，最后观察波形图进行检验。主要的测试代码（testbench）如下，主要功能是一开始产生复位信号，持续300个时间单位后复位信号小时，随后的6000个单位内，每隔50个时间单位将发生时钟信号的翻转，即时钟周期为100个单位。

```
module datapath_sim();

    reg     clock       =   1'b1;
    reg     reset       =   1'b0;

    datapath datapath0(
        clock,
        reset
        );

    initial begin
        #300    reset = 1'b1;
        #6000
        $stop;
    end
    always #50 clock = ~clock;

endmodule
```

3.2.1 算术逻辑运算指令

如表 3-6 所示，在此涉及的指令主要为R-指令，包括了普通运算指令和带标志位的运算指令。

表 3-6 算术逻辑运算指令（含设置标志位）

指令	含义	opcode	功能
ADD	加	10001011000	R[Rd] = R[Rn] + R[Rm]
SUB	减	11001011000	R[Rd] = R[Rn] - R[Rm]
AND	与	10001010000	R[Rd] = R[Rn] & R[Rm]
ORR	或	10101010000	R[Rd] = R[Rn] | R[Rm]
EOR	异或	11001010000	R[Rd] = R[Rn] ^ R[Rm]
LSR	逻辑右移	11010011010	R[Rd] = R[Rn] >>> shamt
LSL	逻辑左移	11010011011	R[Rd] = R[Rn] << shamt
ADDS	带标志位加	10101011000	R[Rd],FLAGS = R[Rn] + R[Rm]
SUBS	带标志位减	11101011000	R[Rd],FLAGS = R[Rn] - R[Rm]
ANDS	带标志位与	11101010000	R[Rd],FLAGS = R[Rn] & R[Rm]

为了检验，我们编写了如下指令，测试的结果如图 3-6 所示。

```
ADD     X2, X1, X0
SUB     X5, X4, X3
SUB     X5, X3, X4
AND     X8, X7, X6
ORR     X11, X10, X9
EOR     X14, X13, X12
SUBS    X5, X4, X3
SUBS    X5, X3, X4
LSR     X5, X4, #3
LSL     X5, X3, #2
```

对应的机器码如下：

```
10001011000_00000_000000_00001_00010 //ADD
11001011000_00011_000000_00100_00101 //SUB
11001011000_00100_000000_00011_00101 //SUB
10001010000_00110_000000_00111_01000 //AND
10101010000_01001_000000_01010_01011 //ORR
11001010000_01100_000000_01101_01110 //EOR
11101011000_00011_000000_00100_00101 //SUBS
11101011000_00100_000000_00011_00101 //SUBS
11010011010_00011_000011_00100_00101 //LSR
11010011011_00100_000010_00011_00101 //LSL
```

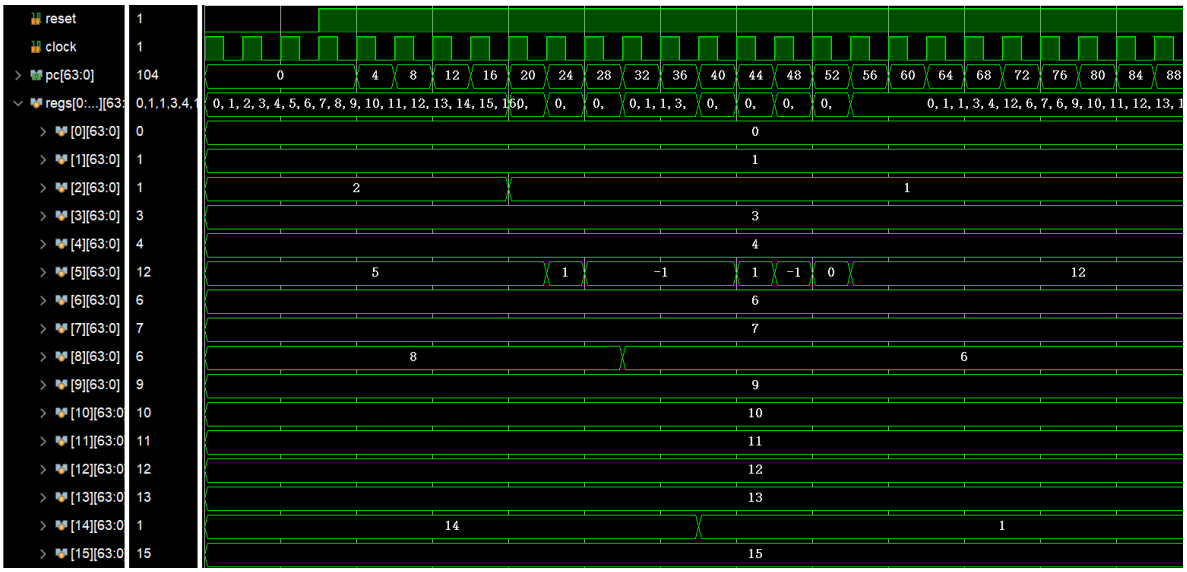


图 3-6 算术逻辑运算指令波形图

下面对程序所涉及到的算术逻辑运算指令进行逐一验证，图 3-6 展示了复位信

号、时钟信号、下一个 PC 的信号、寄存器堆中寄存器的值随着时间变化的情况。寄存器堆内的寄存器内容首先被初始化为了寄存器的编号，随后因各个运算指令的操作而变化所存储的内容。

第一条指令如下，涉及的波形如图 3-7 所示。可以看出这是一条加法指令，第一个操作数是 X1 寄存器的内容，为 1，第二个操作数是 X0 寄存器的内容，为 0，运算的结果应该为 1，并且将被送往 X2 寄存器。可以看出，指令执行的结果正确，时序正确。

```
ADD     X2, X1, X0  ; 10001011000_00000_000000_00001_00010
```

图 3-7 ADD X2,X1,X0 指令的执行情况

第二条指令如下，涉及的波形如图 3-8 所示。可以看出这是一条减法指令，第一个操作数是 X4 寄存器的内容，为 4，第二个操作数是 X3 寄存器的内容，为 3，运算的结果应该为 1，并且将被送往 X5 寄存器。可以看出，指令执行的结果正确，时序正确。

```
SUB     X5, X4, X3  ; 11001011000_00011_000000_00100_00101
```

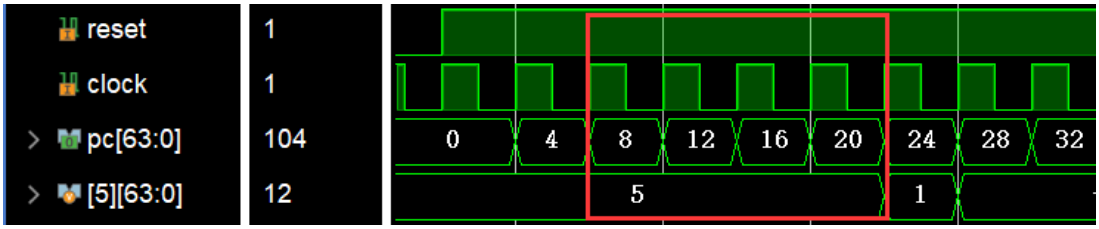


图 3-8 SUB X5,X4,X3 指令的执行情况

第三条指令如下，涉及的波形如图 3-9 所示。可以看出这也是一条减法指令，与上一条不同的是交换了两个操作数的位置，第一个操作数是 X3 寄存器的内容，为 3，第二个操作数是 X4 寄存器的内容，为 4，运算的结果应该为-1，并且将被送往 X5 寄存器。可以看出，指令执行的结果正确，时序正确。

```
SUB     X5, X3, X4  ; 11001011000_00100_000000_00011_00101
```

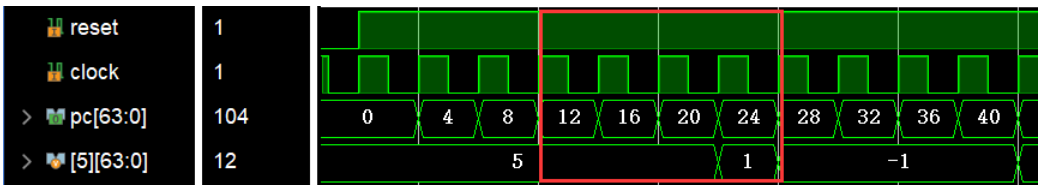


图 3-9 SUB X5, X3, X4 指令的执行情况

第四条指令如下，涉及的波形如图 3-10 所示。可以看出这是一条与指令，第一个操作数是 X7 寄存器的内容，为 0111，第二个操作数是 X6 寄存器的内容，为 0110，运算的结果应该为 0110，即为 6，并且将被送往 X8 寄存器。可以看出，指令执行的结果正确，时序正确。

```
AND     X8, X7, X6  ; 10001010000_00110_000000_00111_01000
```

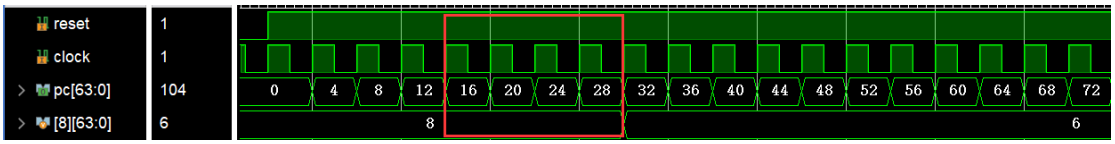


图 3-10 AND X8, X7, X6 指令的执行情况

第五条指令如下，涉及的波形如图 3-11 所示。可以看出这是一条或指令，第一个操作数是 X10 寄存器的内容，为 1010，第二个操作数是 X9 寄存器的内容，为 1001，运算的结果应该为 1011，即为 11，并且将被送往 X11 寄存器。可以看出，X11 的寄存器的内容没有变化，表明结果的确为 11，因此，指令执行的结果正确，时序正确。

```
ORR     X11, X10, X9 ; 10101010000_01001_000000_01010_01011
```

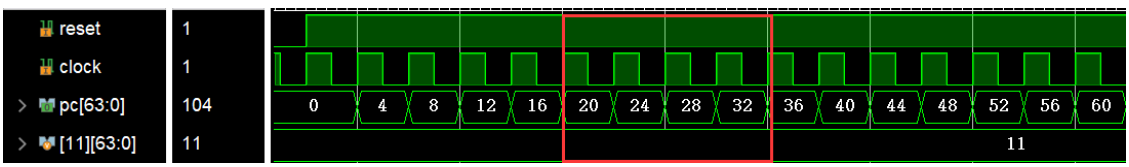


图 3-11 ORR X11, X10, X9 指令的执行情况

第六条指令如下，涉及的波形如所示。可以看出这是一条异或指令，第一个操作数是 X13 寄存器的内容，为 1101，第二个操作数是 X12 寄存器的内容，为 1100，

运算的结果应该为 0001，即为 1，并且将被送往 X14 寄存器。可以看出，指令执行的结果正确，时序正确。

```
EOR      X14, X13, X12 ; 11001010000_01100_000000_01101_01110
```

图 3-12 EOR X14, X13, X12 指令的执行情况

第七条指令如下，涉及的波形如图 3-13 所示。可以看出这是一条带标志位的减法指令，第一个操作数是 X4 寄存器的内容，为 4，第二个操作数是 X3 寄存器的内容，为 3，运算的结果应该为 1，并且将被送往 X5 寄存器。可以看出，指令执行的结果正确，时序正确。

```
SUBS     X5, X4, X3   ; 11101011000_00011_000000_00100_00101
```

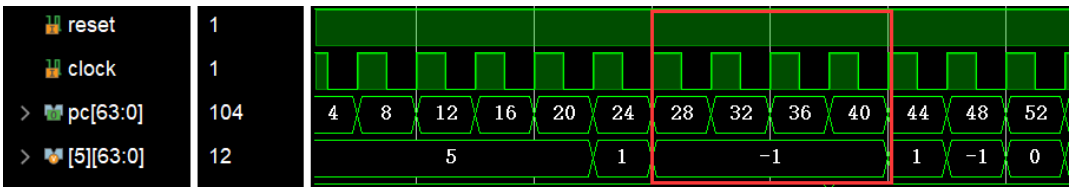


图 3-13 SUBS X5, X4, X3 指令的执行情况

第八条指令如下，涉及的波形如图 3-14 所示。可以看出这是一条带标志位的减法指令，第一个操作数是 X3 寄存器的内容，为 3，第二个操作数是 X4 寄存器的内容，为 4，运算的结果应该为-1，并且将被送往 X5 寄存器。可以看出，指令执行的结果正确，时序正确。

```
SUBS     X5, X3, X4   ; 11101011000_00100_000000_00011_00101
```

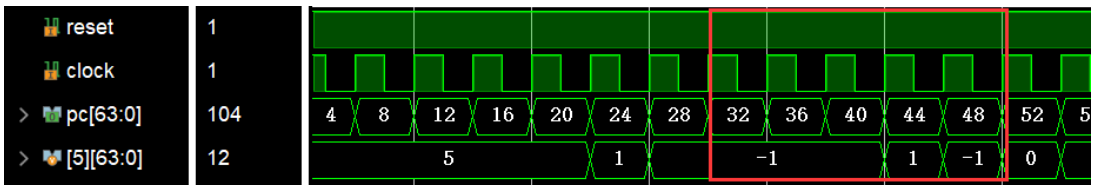


图 3-14 SUBS X5, X3, X4 指令的执行情况

第九条指令如下，涉及的波形如图 3-15 所示。可以看出这是一条逻辑右移指令，第一个操作数是 X4 寄存器的内容，为 0100，第二个操作数是立即数，为 3，运算的结果应该为 0，并且将被送往 X5 寄存器。可以看出，指令执行的结果正确，时序正确。

```
LSR      X5, X4, #3    ; 11010011010_00011_000011_00100_00101
```

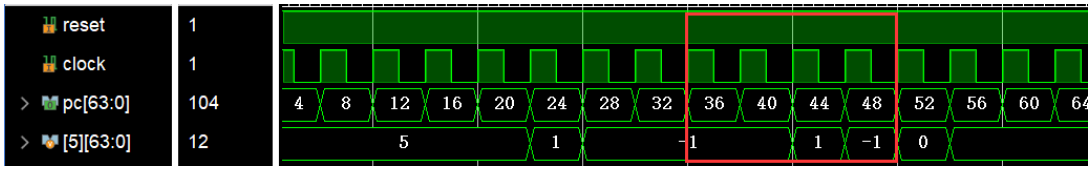


图 3-15 LSR X5, X4, #3 指令的执行情况

第十条指令如下，涉及的波形如图 3-10 所示。可以看出这是一条逻辑左移指令，第一个操作数是 X3 寄存器的内容，为 0011，第二个操作数是立即数，为 2，运算的结果应该为 1100，并且将被送往 X5 寄存器。可以看出，结果为 12，符合。可见指令执行的结果正确，时序正确。

```
LSL      X5, X3, #2    ; 11010011011_00100_000010_00011_00101
```

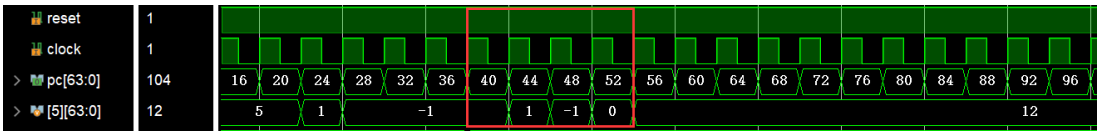


图 3-16 LSL X5, X3, #2 指令的执行情况

综合上述测试的结果，可以看出，所实现的算术逻辑运算指令能够得到正确的结果，且时序正确，由于篇幅受限，若要测试内部模块的信号，可以直接运行源码，查看相应的波形图。

3.2.2 带立即数的算术逻辑运算指令

如表 3-7 所示，在此涉及的指令主要为 I-指令，包括了带立即数的算术运算和逻辑运算指令。

表 3-7 带立即数的算术逻辑运算指令

指令	含义	opcode	功能
ADDI	带立即数的加	1001000100	R[Rd] = R[Rn] + ALUImm
SUBI	带立即数的减	1001001000	R[Rd] = R[Rn] - ALUImm
ANDI	带立即数的与	1011001000	R[Rd] = R[Rn] & ALUImm
ORRI	带立即数的或	1101000100	R[Rd] = R[Rn] | ALUImm
EORI	带立即数的异或	1101001000	R[Rd] = R[Rn] ^ ALUImm

为了检验，我们编写了如下指令，测试的结果如所示。

```
ADDI    X5, X0, #7
ANDI    X6, X1, #7
ORRI    X7, X2, #7
SUBI    X8. X3, #7
EORI    X9, X4, #7
```

对应的机器码如下：

```
1001000100_000000000111_00000_00101 // ADDI
1001001000_000000000111_00001_00110 // ANDI
1011001000_000000000111_00010_00111 // ORRI
1101000100_000000000111_00011_01000 // SUBI
1101001000_000000000111_00100_01001 // EORI
```

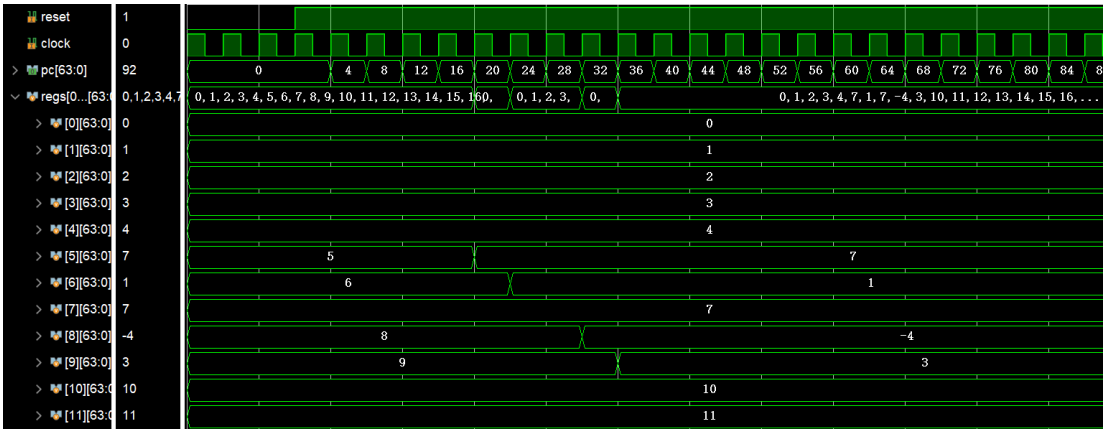


图 3-17 带立即数的算术逻辑运算指令波形图

下面对程序所涉及到的带立即数的算术逻辑运算指令进行逐一验证，图 3-17 展示了复位信号、时钟信号、下一个 PC 的信号、寄存器堆中寄存器的值随着时间变化的情况。寄存器堆内的寄存器内容首先被初始化为了寄存器的编号，随后因各个运算指令的操作而变化所存储的内容。

第一条指令如下，涉及的波形如图 3-18 所示。可以看出这是一条带立即数的加法指令，第一个操作数是 X0 寄存器的内容，为 0，第二个操作数是立即数，为 7，运算的结果应该为 7，并且将被送往 X5 寄存器。可见指令执行的结果正确，时序正确。

```
ADDI    X5, X0, #7   ; 1001000100_000000000111_00000_00101
```

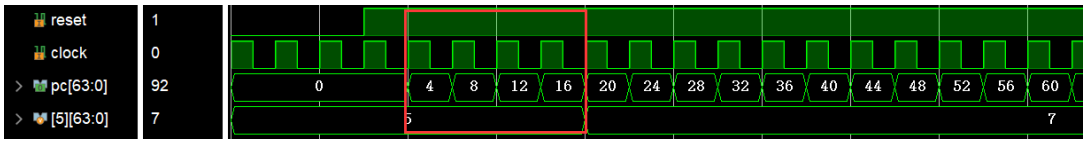


图 3-18 ADDI X5,X0,#7 指令的执行情况

第二条指令如下，涉及的波形如图 3-19 所示。可以看出这是一条带立即数的与运算指令，第一个操作数是 X1 寄存器的内容，为 0001，第二个操作数是立即数，为 7，即 0111，运算的结果应该为 1，并且将被送往 X6 寄存器。可以看出，指令执行的结果正确，时序正确。

```
ANDI    X6, X1, #7   ; 1001001000_000000000111_00001_00110
```

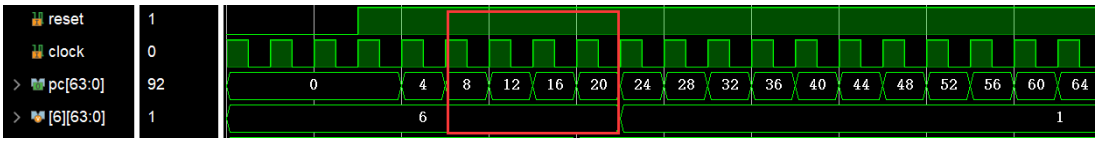


图 3-19 ANDI X6,X1,#7 指令的执行情况

第三条指令如下，涉及的波形如图 3-20 所示。可以看出这是一条带立即数的或运算指令，第一个操作数是 X2 寄存器的内容，为 0010，第二个操作数是立即数，为 7，即 0111，运算的结果应该为 0111，并且将被送往 X7 寄存器。可以看出，X7 寄存器的值不变，即写入的结果与原来的初始化值相同，结果为 7，符合。可见指令执行的结果正确，时序正确。

```
ORRI    X7, X2, #7   ; 1011001000_000000000111_00010_00111
```

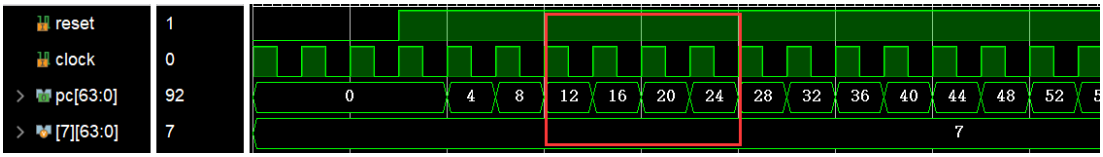


图 3-20 ORRI X7,X2,#7 指令的执行情况

第四条指令如下，涉及的波形如图 3-21 所示。可以看出这是一条带立即数的减法指令，第一个操作数是 X3 寄存器的内容，为 3，第二个操作数是立即数，为 7，运算的结果应该为-4，并且将被送往 X8 寄存器。可以看出，指令执行的结果正确，时序正确。

```
SUBI    X8, X3, #7   ; 1101000100_000000000111_00011_01000
```

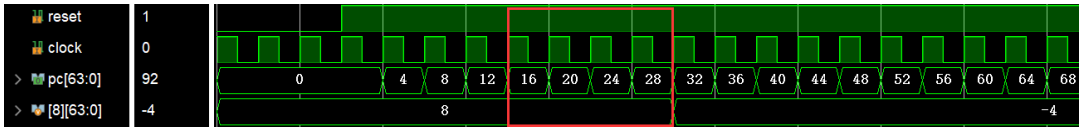


图 3-21 SUBI X8,X3,#7 指令的执行情况

第五条指令如下，涉及的波形如图 3-22 所示。可以看出这是一条带立即数的异或指令，第一个操作数是 X4 寄存器的内容，为 0100，第二个操作数是立即数，为 7，即 0111，运算的结果应该为 0011，为 3，并且将被送往 X9 寄存器。可以看出，结果为 3，符合。可见指令执行的结果正确，时序正确。

```
EORI    X9, X4, #7   ; 1101001000_000000000111_00100_01001
```

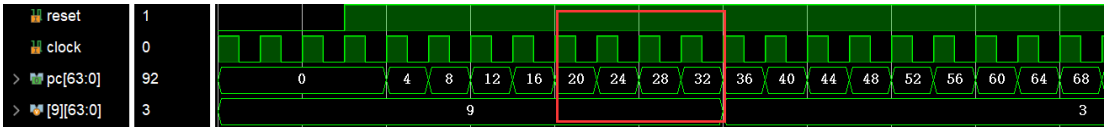


图 3-22 EORI X9,X4,#7 指令的执行情况

综合上述测试的结果，可以看出，所实现的带立即数的算术逻辑运算指令能够得到正确的结果，且时序正确。

3.2.3 带立即数和标志位的算术逻辑运算指令

如表 3-8 所示，在此涉及的指令主要为 I-指令，包括了带立即数和标志位的算

术运算和逻辑运算指令。

表 3-8 带立即数和标志位的算术逻辑运算指令

指令	含义	opcode	功能
ADDI	带立即数和标志位的加	1011000100	R[Rd],FLAGS = R[Rn] + ALUImm
SUBI	带立即数和标志位的减	1111000100	R[Rd],FLAGS = R[Rn] - ALUImm
ANDI	带立即数和标志位的与	1111001000	R[Rd],FLAGS = R[Rn] & ALUImm

为了检验，我们编写了如下指令，测试的结果如所示。

```
ADDIS   X1, X0, #1
SUBIS   X3, X2, #2
ANDIS   X5, X4, #3
```

对应的机器码如下：

```
1011000100_000000000001_00000_00001 // ADDIS
1111000100_000000000010_00010_00011 // SUBIS
1111001000_000000000011_00100_00101 // ANDIS
```

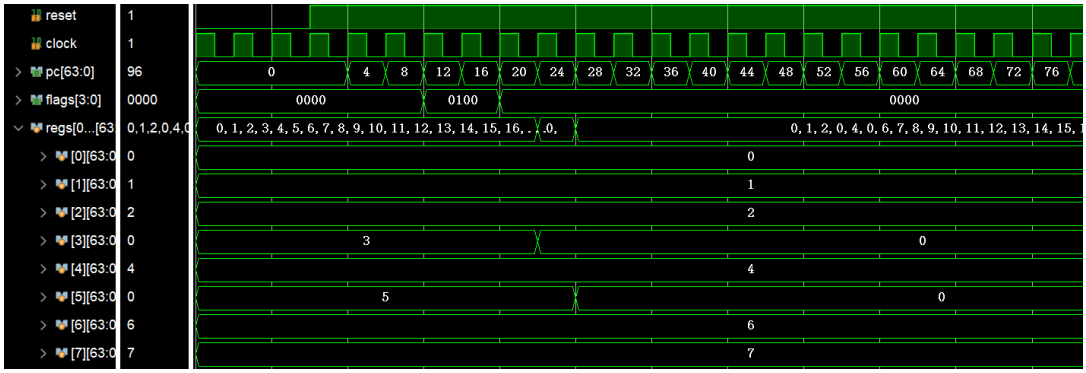


图 3-23 带立即数和标志位的算术逻辑运算指令波形图

下面对程序所涉及到的带立即数和标志位的算术逻辑运算指令进行逐一验证，图 3-23 展示了复位信号、时钟信号、下一个 PC 的信号、标志位 NZVC、寄存器堆中寄存器的值随着时间变化的情况。寄存器堆内的寄存器内容首先被初始化为了寄存器的编号，随后因各个运算指令的操作而变化所存储的内容。

第一条指令如下，涉及的波形如图 3-24 所示。可以看出这是一条带立即数和标志位的加法指令，第一个操作数是 X0 寄存器的内容，为 0，第二个操作数是立即数，为 1，运算的结果应该为 1，并且将被送往 X1 寄存器。从结果可以看出，X1 寄存器的值并未发生变化，可见指令执行的结果正确，时序正确。

```
ADDIS   X1, X0, #1   ; 1011000100_000000000001_00000_00001
```

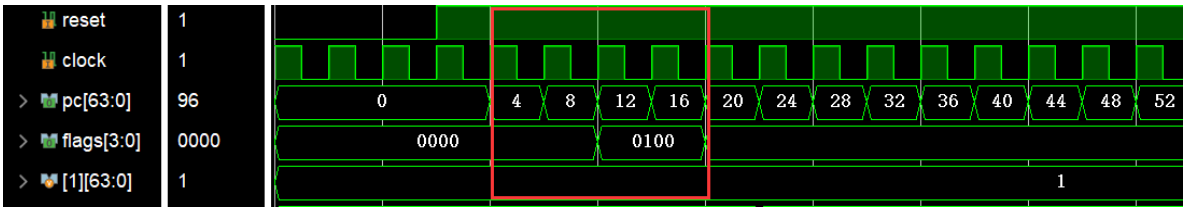


图 3-24 ADDIS X1,X0,#1 指令的执行情况

第二条指令如下，涉及的波形如图 3-25 所示。可以看出这是一条带立即数和标志位的减法指令，第一个操作数是 X2 寄存器的内容，为 2，第二个操作数是立即数，为 2，运算的结果应该为 0，并且将被送往 X3 寄存器，同时设置标志位 Z=1，则 FLAGS=0100。可见指令执行的结果正确，时序正确。

```
SUBIS   X3, X2, #2   ; 1111000100_000000000010_00010_00011
```

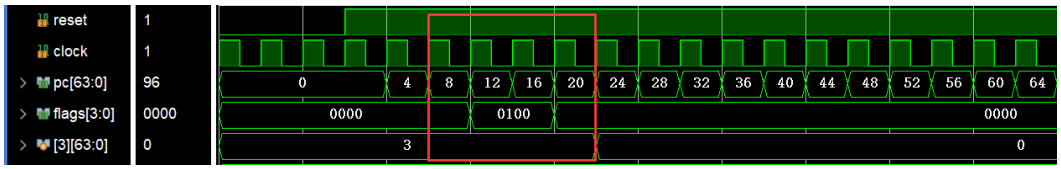


图 3-25 SUBIS X3,X2,#2 指令的执行情况

第三条指令如下，涉及的波形如图 3-26 所示。可以看出这是一条带立即数和标志位的与运算指令，第一个操作数是 X4 寄存器的内容，为 0100，第二个操作数是立即数，为 3，即 0011，运算的结果应该为 0，并且将被送往 X5 寄存器，同时设置标志位 Z=1，则 FLAGS=0100。可见指令执行的结果正确，时序正确。

```
ANDIS   X5, X4, #3   ; 1111001000_000000000011_00100_00101
```

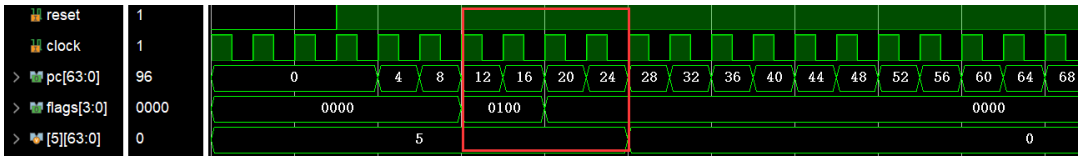


图 3-26 ANDIS X5,X4,#3 指令的执行情况

综合上述测试的结果，可以看出，所实现的带立即数和标志位的算术逻辑运算指令能够得到正确的结果，且时序正确。

3.2.4 存数取数指令

如所示，在此涉及的指令主要为 D-指令，包括了存数和取数指令。

表 3-9 存数取数指令

指令	含义	opcode	功能
LDUR	取数	11111000010	R[Rt] = M[R[Rn]+ DTAddr]
STUR	存数	1111100000	M[R[Rn]+ DTAddr] = R[Rt]

为了检验，我们编写了如下指令，测试的结果如所示。

```
LDUR    X2, [X0+8]
STUR    X3, [X1+1]
```

对应的机器码如下：

```
11111000010_000001000_00_00000_00010 // LDUR
11111000000_000000001_00_00001_00011 // STUR
```

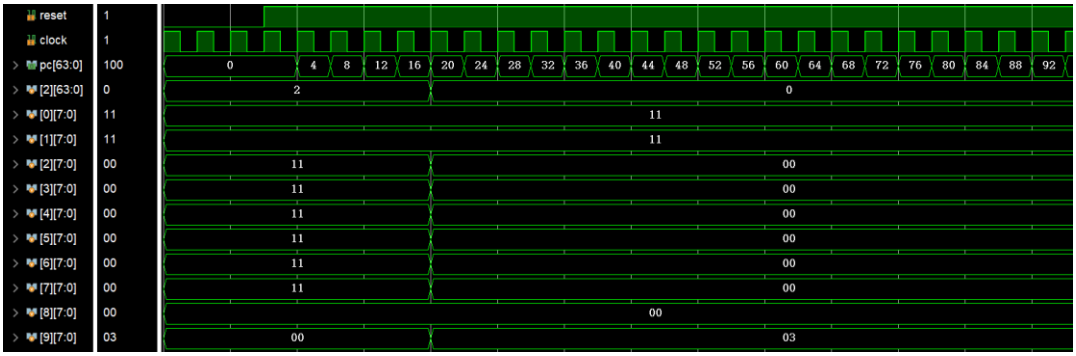


图 3-27 存数取数指令波形图

下面对程序所涉及到的存数取数指令进行逐一验证，展示了复位信号、时钟信号、下一个 PC 的信号、寄存器堆中寄存器的值、数据存储器中存储各个单元的值随着时间变化的情况。寄存器堆内的寄存器内容首先被初始化为了寄存器的编号，数据存储器中的各个单元前 8 个字节被初始化为 11，之后的单元被初始化为 00。随后因各个运算指令的操作而变化所存储的内容。

第一条指令如下，涉及的波形如图 3-28 所示。可以看出这是一条取数指令，即从[X0+8]的数据存储器位置取出数据，而 X0 被初始化为 0，即从第 8 个存储单元起，按照大端的方式取出数据，送往 X2 寄存器，从结果可以看出，X2 由初始值 2 变为 0，可见指令执行的结果正确，时序正确。

```
LDUR    X2, [X0+8]    ; 11111000010_000001000_00_00000_00010
```

图 3-28 LDUR X2, [X0+8]指令的执行情况

第二条指令如下，涉及的波形如图 3-29 所示。可以看出这是一条存数指令，即从 X3 寄存器取值，送往[X1+1]的数据存储器位置，而 X1 被初始化为 1，即送往首地址为 2 的存储单元，并按照大端的方式存放数据，从结果可以看出，以 2 为首的存储单元被存入了 64 位的数据，数值为 3，可见指令执行的结果正确，时序正确。

```
STUR    X3, [X1+1]    ; 11111000000_000000001_00_00001_00011
```

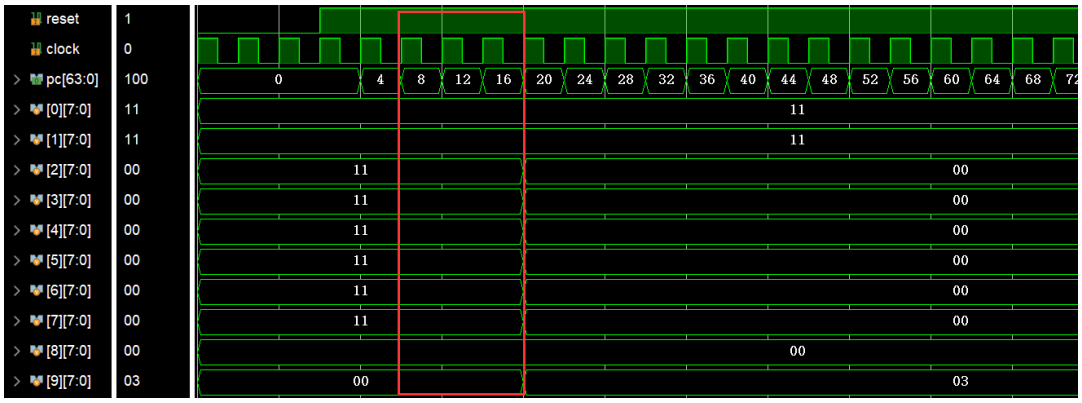


图 3-29 STUR X3, [X1+1]指令的执行情况

综合上述测试的结果，可以看出，所实现的存数取数指令能够得到正确的结果，且时序正确。

3.3 流水线的优化

至此，我们已经实现了支持各种类型指令的基于 ARMv8 指令集架构的 CPU，然而，依然存在一些问题，假若在执行运算等指令的时候，指令之间会存在很多的相关性，从而很容易引起冒险（Hazard）。为了解决这个问题，我们需要对流水线进行优化。对此，我们将从冒险的产生、冒险的解决、解决的效果等方面进行介绍。

3.3.1 冒险的产生

由于我们采用了五级流水线的设计，每一次写寄存器的操作都需要在写回阶段才能实现，故若有指令在数据被写回寄存器之前读取寄存器，那么最终读取的值是不准确的，即真相关引发的 RAW 冒险，具体而言，若这两条指令差距在三个时钟周期内，那么就会导致读取结果的错误。

3.3.2 冒险的解决

为了解决在本节所提及的问题，我们可以设置旁路模块，检测译码阶段所要访问的寄存器是否与当前执行阶段、访存阶段要写入的寄存器相同，同时检测该阶段的 RegWrite 是否活跃。

如果两条指令相差正好三个时钟周期，我们可以直接修改寄存器堆的硬件设计解决，即在时钟周期的前半部分写，后半部分读，因此读操作将读取到最新写入的内容。

对于相差小于三个时钟周期的两条指令，我们可以通过上述方案解决，在实现上，如果流水线中的指令以 XZR 作为目的寄存器，那么就要避免把可能的非零结果旁路。指令 STUR 通过 Rt 字段[4:0]指明第二寄存器操作数，对此，我们在流水线 ID 阶段通过一个二选一选择器和控制信号进行选择，并直接使用该多路选择器的输出代替实际的 Rm 字段，这样就统一了各种指令的执行情况。还需要考虑更复杂的情况是，若译码阶段所要访问的寄存器与当前执行阶段和访存阶段要写入的寄存器都相同，此时最新的数据应该在执行阶段，故需要直接取执行阶段的结果，在实现上需要引入更多的判断。

```
module forwarding(

    input   wire    [`RegAddrBus]     Rn,
    input   wire    [`RegAddrBus]     Rm,
    input   wire    [`RegAddrBus]     ex_Rd,
    input   wire    [`RegAddrBus]     mem_Rd,
    input   wire                      ex_RegWrite,
    input   wire                      mem_RegWrite,

    output  reg     [`ForwardBus]     Fa,
    output  reg     [`ForwardBus]     Fb

    );
```

```
    always @(*) begin
        Fa <= 2'b00;
        Fb <= 2'b00;
        if (ex_RegWrite && (ex_Rd != 5'd31) && (ex_Rd == Rn)) begin
            Fa <= 2'b10;
        end
        if (ex_RegWrite && (ex_Rd != 5'd31) && (ex_Rd == Rm)) begin
            Fb <= 2'b10;
        end
        if (mem_RegWrite && (mem_Rd != 5'd31) && (mem_Rd == Rn)
            && !(ex_RegWrite && (ex_Rd != 5'd31) && (ex_Rd == Rn))
        ) begin
            Fa <= 2'b01;
        end
        if (mem_RegWrite && (mem_Rd != 5'd31) && (mem_Rd == Rm)
            && !(ex_RegWrite && (ex_Rd != 5'd31) && (ex_Rd == Rm))
        ) begin
            Fb <= 2'b01;
        end
    end

endmodule
```

旁路模块将输入译码阶段需要访问的寄存器、执行和访存阶段的写信号和目的寄存器，经过处理输出相应的控制信号，以控制输入至运算器中的数据。控制信号如表 3-10 所示。

表 3-10 旁路多路选择器的控制信号

多路选择器控制	源	解释
Fa=00	ID/EX	第一个 ALU 操作数从寄存器文件中获得
Fa=10	EX/MEM	第一个 ALU 操作数由上一个 ALU 运算结果旁路获得
Fa=01	MEM/WB	第一个 ALU 操作数从数据存储器或前面的 ALU 结果中旁路获得
Fb=00	ID/EX	第二个 ALU 操作数从寄存器文件中获得
Fb=10	EX/MEM	第二个 ALU 操作数由上一个 ALU 运算结果旁路获得
Fb=01	MEM/WB	第二个 ALU 操作数从数据存储器或前面的 ALU 结果中旁路获得

3.3.3 解决的效果

我们通过引入旁路模块，检测各个阶段的寄存器和控制信号，解决了真相关引发的冒险。在此，通过一段指令展示最终的效果，相关代码如下，这段代码中寄存器 X2 被多次使用，并不断被累加，相邻的指令之间 X2 既作为输入也作为输出，有读后写的情况，即存在真相关。

```
ADD X2, X1, X0
ADD X2, X2, X1
ADD X2, X2, X1
ADD X2, X2, X1
```

执行的结果如图 3-30 所示，图中分别展示了时钟信号、旁路控制信号和寄存器堆中的值，可以看出，在寄存器堆中，X2 寄存器先按照寄存器编号被初始化为 2，随后在第一条指令被执行后的第五个周期，寄存器被置位 1，随后不断累加，查看旁路单元的信号，可以看出，检测信号为 Fa=10，即检测到了第一个操作数的冲突，并旁路执行阶段的结果，最终使得结果正确。

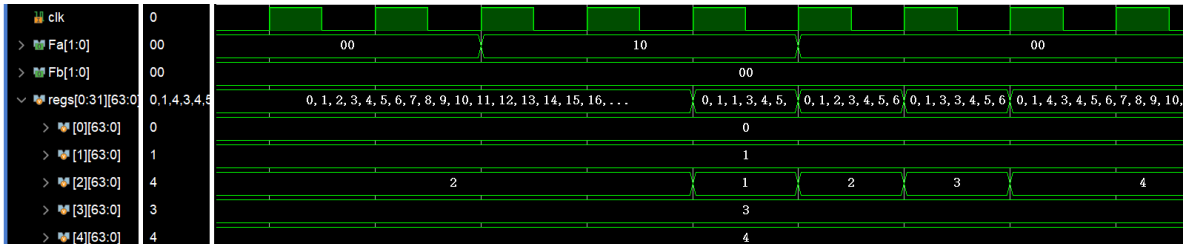


图 3-30 数据旁路的解决效果

4 开发板的烧录与调试

经过前面几节的介绍和实现，我们已经得到了支持三十多条指令的 ARMv8 架构的 CPU，在本节，我们将通过编写演示程序和其他模块以通过开发板展示。首先，本节将介绍为了开发板演示而增加的模块，随后，本节将展示演示程序和演示效果。

4.1 新增加的模块

在 2.3 节中曾经介绍过经过调研总结的开发板使用注意事项，基于这些注意事项，本节将对相应的模块进行开发设计。此外，为了通过数码管展示寄存器和 PC 的内容，我们需要对相关的模块进行修改，引出相应的内容。

4.1.1 数码管译码模块

本次开发设计需要分别显示第 0 号、第 2 号、第 3 号、第 4 号寄存器以及 PC 低 16 位的内容，因为要显示不同的数值，数码管需要采用动态显示的策略，在代码编写中需要将输入的数值进行译码，产生点亮数码管数码段的信号和位选的信号，利用人眼的视觉暂留现象即可出现数码管计数的效果。相关代码如下，在此，为了控制数码管位选的扫描频率，对输入的时钟信号进行了 2^{15} 分频，同时产生位选的信号。

```
module seg(

    output   reg      [15:0]     duan,  // 数码管的公共段选信号
    output   reg      [7:0]      an,      // 作为 8 个数码管的位选信号

    input    wire                  clk,
    input    wire                  rst,
    input    wire      [3:0]     in3, in2, in1, in0,
    input    wire      [3:0]     in7, in6, in5, in4

    );
        // EGo1 数码管是共阴极的，需要连接高电平，对应位置被点亮
        parameter    _0 = ~8'hc0;
        parameter    _1 = ~8'hf9;
        parameter    _2 = ~8'ha4;
```

```
    parameter    _3 = ~8'hb0;
    parameter    _4 = ~8'h99;
    parameter    _5 = ~8'h92;
    parameter    _6 = ~8'h82;
    parameter    _7 = ~8'hf8;
    parameter    _8 = ~8'h80;
    parameter    _9 = ~8'h90;
    parameter    _a = ~8'h88;
    parameter    _b = ~8'h83;
    parameter    _c = ~8'hc6;
    parameter    _d = ~8'ha1;
    parameter    _e = ~8'h86;
    parameter    _f = ~8'h8e;
    parameter    _err = ~8'hcf;

parameter    N = 18;
reg      [N-1 : 0]   regN;
reg      [3:0]       hex_in;

always @ (posedge clk or posedge rst)    begin
    if (rst == `RstEnable)      begin
        regN      <=   0;
    end else      begin
        regN      <=   regN + 1;
    end
end
// regN 实现对 100MHz 的系统时钟的分频
always @ (*)      begin
    case (regN[N-1: N-3])
        3'b000:   begin
            an   <=   8'b00000001;
            hex_in   <=   in0;
        end
        3'b001:   begin
            an   <=   8'b00000010;
            hex_in   <=   in1;
        end
        3'b010:   begin
            an   <=   8'b00000100;
            hex_in   <=   in2;
        end
        3'b011:   begin
            an   <=   8'b00001000;
            hex_in   <=   in3;
        end
```

```
            3'b100:   begin
                an   <=   8'b00010000;
                hex_in   <=   in4;
            end
            3'b101:   begin
                an   <=   8'b00100000;
                hex_in   <=   in5;
            end
            3'b110:   begin
                an   <=   8'b01000000;
                hex_in   <=   in6;
            end
            3'b111:   begin
                an   <=   8'b10000000;
                hex_in   <=   in7;
            end
            default:      begin
                an   <=   8'b11111111;
                hex_in   <=   in3;
            end
        endcase
    end

    always @ (*)      begin
        if ((an & 8'b11110000) == 0)
            case (hex_in)
                4'h0:     duan[7:0] <=   _0;
                4'h1:     duan[7:0] <=   _1;
                4'h2:     duan[7:0] <=   _2;
                4'h3:     duan[7:0] <=   _3;
                4'h4:     duan[7:0] <=   _4;
                4'h5:     duan[7:0] <=   _5;
                4'h6:     duan[7:0] <=   _6;
                4'h7:     duan[7:0] <=   _7;
                4'h8:     duan[7:0] <=   _8;
                4'h9:     duan[7:0] <=   _9;
                4'ha:     duan[7:0] <=   _a;
                4'hb:     duan[7:0] <=   _b;
                4'hc:     duan[7:0] <=   _c;
                4'hd:     duan[7:0] <=   _d;
                4'he:     duan[7:0] <=   _e;
                4'hf:     duan[7:0] <=   _f;
                default:duan[7:0] <=   _err;
            endcase
        else
```

```
            case (hex_in)
                4'h0:     duan[15:8] <=   _0;
                4'h1:     duan[15:8] <=   _1;
                4'h2:     duan[15:8] <=   _2;
                4'h3:     duan[15:8] <=   _3;
                4'h4:     duan[15:8] <=   _4;
                4'h5:     duan[15:8] <=   _5;
                4'h6:     duan[15:8] <=   _6;
                4'h7:     duan[15:8] <=   _7;
                4'h8:     duan[15:8] <=   _8;
                4'h9:     duan[15:8] <=   _9;
                4'ha:     duan[15:8] <=   _a;
                4'hb:     duan[15:8] <=   _b;
                4'hc:     duan[15:8] <=   _c;
                4'hd:     duan[15:8] <=   _d;
                4'he:     duan[15:8] <=   _e;
                4'hf:     duan[15:8] <=   _f;
                default:duan[15:8] <=   _err;
            endcase
    end

endmodule
```

4.1.2 分频器

如果直接采用开发板提供的 100M 晶振，数码管无法正常显示，会出现重叠的现象，对此需要通过分频控制程序执行的速度。相关代码如下，设置了 2 个 14 位的变量，通过累加进行分频计数，最终能够对信号进行 10^8 分频，最终能够使得程序以 1 秒 1 条指令的速度执行。

```
odule counter(

        input clk,
        input rst,
        output clk_bps

        );
        reg [13:0]cnt_first,cnt_second;
        always @( posedge clk or posedge rst )
            if( rst == `RstEnable )
                cnt_first <= 14'd0;
            else if( cnt_first == 14'd10000 )
                cnt_first <= 14'd0;
            else
```

```
                cnt_first <= cnt_first + 1'b1;
        always @( posedge clk or posedge rst )
            if( rst == `RstEnable )
                cnt_second <= 14'd0;
            else if( cnt_second == 14'd10000 )
                cnt_second <= 14'd0;
            else if( cnt_first == 14'd10000 )
                cnt_second <= cnt_second + 1'b1;
        assign clk_bps = cnt_second == 14'd10000 ? 1'b1 : 1'b0;
endmodule
```

4.2 演示程序设计

基于设计的指令，设计了如下的程序，其中包含了基本的算术逻辑运算指令、带立即数的算术逻辑运算指令、带标志位的算术逻辑运算指令、带立即数和标志位的算术逻辑运算指令以及跳转指令，在本程序中还包含了数据相关，从指令之间的距离关系可以看出，容易引发数据冒险。

```
start:
AND     X0, X0, X31
ADDI    X0, X0, #1
SUBS    X2, X1, X0
ADDI    X0, X0, #1
AND      X3, X0, X2
ADDIS   X0, X0, #1
LSR     X4, #3
B        start
```

该指令对应的机器码如下：

```
10001010000_11111_000000_00000_00000
1001000100_000000000001_00000_00000
11101011000_00001_000000_00000_00010
1001000100_000000000001_00000_00000
10001010000_00010_000000_00000_00011
1011000100_000000000001_00000_00000
11010011010_00000_000011_00100_00100
000101_11111111111111111111111001
```

从程序中可以看出，程序在执行的过程中将改变寄存器的值，最终 X0 的值将在 0、1、2、3 之间循环变化，而 X2、X3、X4 的值分别为 1、0、0。

4.3 演示效果

我们编写的文件保存并通过综合实现然后生成 Bitstream 文件，烧录至开发板中，如图 4-1 所示，开发板上的数码管分别展示了寄存器堆中第 0 号、第 2 号、第 3 号、第 4 号寄存器以及 PC 低 16 位的内容。

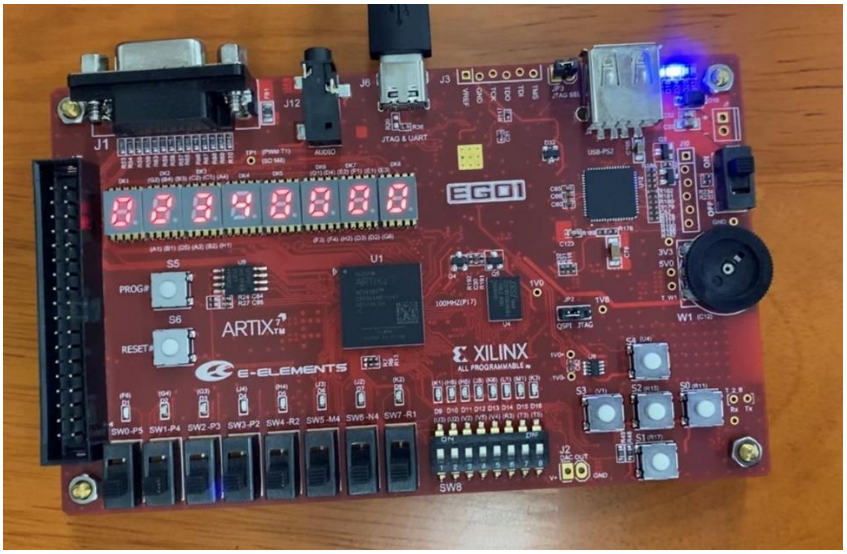


图 4-1 开发板演示效果

囿于篇幅，详细的过程照片可以参见附录三：开发板测试中数码管显示结果，在此将程序执行过程中数码管的显示情况通过表 4-1 展示。从开发板数码管的展示情况可以看出，寄存器的被初始化问对应的编号，PC 从 0 开始计数。经过分频，数码管以一个正常的速度显示每条指令执行的结果。由于指令是 32 位的，所以 PC 每个时钟周期都加 4。经过 5 个时钟周期，算术逻辑运算的指令得结果最终能够写入到对应的寄存器中，且跳转正常。源程序中存在数据相关，从结果能够正常显示可以看出，此次设计避免了数据冒险的发生。

表 4-1 程序执行过程中数码管的显示情况

X0	X2	X3	X4	PC[15:12]	PC[11:8]	PC[7:4]	PC[3:0]
0	2	3	4	0	0	0	0
0	2	3	4	0	0	0	4
0	2	3	4	0	0	0	8
0	2	3	4	0	0	0	c
0	2	3	4	0	0	1	0
1	2	3	4	0	0	1	4
1	1	3	4	0	0	1	8
2	1	3	4	0	0	1	c
2	1	0	4	0	0	2	0
3	1	0	4	0	0	2	4
3	1	0	0	0	0	2	8
3	1	0	0	0	0	0	0
3	1	0	0	0	0	0	4
3	1	0	0	0	0	0	8
3	1	0	0	0	0	0	c
0	1	0	0	0	0	1	0

5 总结与心得

5.1 设计总结

通过两人合作，本次课程设计主要完成了如下几点工作：

- 实现了基于 ARMv8 指令集架构五级流水线 CPU 设计，并支持三十余条指令，其中包括多种算术逻辑运算指令、访存指令、跳转指令等；
- 实现了通过数据前推解决真相关可能造成的冲突；
- 实现了开发板的烧录和调试，能够通过开发板的数码管展示寄存器堆中部分寄存器和程序计数器低 16 位的内容。

其中，本人参与的主要工作有：

- 编写了大部分基本硬件模块，实现了 21 条指令，其中包括 7 条基本算术逻辑运算指令、5 条含立即数的算术逻辑运算指令、3 条设置标志位的算术运算指令、3 条设置标志位的含立即数的算术运算指令、3 条存数取数和其他类型的指令；
- 通过数据前推解决数据冒险，实现了流水线优化；
- 合并两人的代码；
- 编写演示程序，调试烧录至开发板。

5.2 未来的工作

本次课程设计虽然实现了预期的目标，但还有一些提升的空间：

- 支持更多的指令，能够实现复杂的程序；
- 引入分支预测机制，解决控制相关导致的冒险；
- 支持程序的乱序执行，能够实现精确异常；
- 加入 Cache 或多级 Cache，实现各种替换算法；
- 编写支持本指令集架构的编译器，使本项目更加完整、友好。

5.3 设计心得

这绝对是一次难忘的经历，从未有过的合作设计开发、软硬件结合的调试……一点一滴中都给大学生话留下了深刻的一笔。通过此次《系统硬件综合设计》，我们学到了很多理论课程中没有的内容，在此次实践中，我们不仅能够加深课内知识的理解，还能够习得很多宝贵的经验。

学会合作。在以往的学习经历来看，不论是理论学习还是上机实验，我们很少有过合作开发的经历，就算有也大多是完成各自的子项目，并没有完全面向同一个项目。就我个人而言，更喜欢一个人开发，因为担心如果协调不好还要花费更多的时间，在此次课程设计中，我们通过合作实现了 CPU 的设计，在分工上，我们努力将很多的任务拆解为并行实现，避免有人没事情做的情况，从结果可以看到，通过合作能够在一定程度上提高开发的效率，不仅体现在任务分配上，而且通过相互监督，能够很好地杜绝动力不足的现象。这为以后的项目开发提供了宝贵的实践经验。

学会分析。我们在实践开发的过程中遇到了无数莫名其妙的情况，比如寄存器的值突然变成了 x 或 z、开发板上数码管静止不动、指令跳转不符合预期等等，一开始出现这些问题，我们往往表现得十分手足无措，最后就是干着急和绝望，但通过检查波形图、原理图，从一根根数据线、一条条数据通路中摸索，最终能够解决相关的问题。能够体会到，作为一名合格的程序员应该要具备会分析和调试的能力，否则遇到问题很难解决，导致最终无法出现想要的效果。

学会规划。虽然此次课程设计历时较短，但是包含了大量的工作，不论是前期的理论学习，还是后期的开发设计与调试，一步步都需要进行合理的规划，尤其不能把太多的时间花费在理论学习上，因为学习是永无止境的，如果不基于实际的问题，单纯地学习很多内容，一方面很难让知识落地，另一方面也会打消学习的积极型。再者，每一步开发都需要理清开发的思路和步骤，如在开发板调试中，一次 Bitstream 文件的生成与烧录需要花费至少五分钟的时间，在这样的过程中，每一次测试都是一种煎熬，尤其是当出现不符合预期的结果时，心情更是崩溃，然而在之前的开发中，一开始忽略了仿真测试，而直接烧录，结果因为一个低级错误尝试很久也没能看到预期的效果。由此也能窥见规划的重要性，不然可能花费更多的时间。

在此次课程设计的过程中，收获不仅限于此，还有很多内化于心、外化于行的宝贵经验，这些经验，也同样指引我们在未来的道路上越走越远。

5.4 致谢

经过短短几周的课程设计，从理论学习到开发设计，收获满满。首先在此感谢李老师、阙老师、丁老师贯穿大二大三的理论教学和实验指导，没有这些先修知识和实践经历我们很难能够在短时间内开发 5 阶段流水线的 CPU；其次感谢一同开发的徐同学，没有相互之间的合作、鼓励、监督，我们很难能够坚持下去，同时还要感谢一班的王同学，没有他提供的建议以及在开发板调试上提供的帮助，我们可能会一直停留在瓶颈处而花费更多的时间。

参考文献

[1] 雷思磊.自己动手写 CPU[M].北京：电子工业出版社,2014

[2] （美）戴维·A.帕特森，约翰·L.亨尼斯.计算机组成与设计　硬件/软件接口　ARM 版[M].北京：机械工业出版社,2018

[3] 依元素科技有限公司.EGO1 用户手册[M/OL].深圳：依元素科技有限公司,2017

[4] （美)亨尼斯等.计算机体系结构　量化研究方法　英文版　第 5 版[M].北京：机械工业出版社,2012

[5] 张晨曦，王志英，张春元，王伟，沈立.计算机系统结构　第 2 版[M].北京：高等教育出版社,2014

[6] Peter Knaggs. ARM Assembly Language Programming[M/OL],2016

[7] ARM. ARMv8 Instruction Set Overview[M/OL],2011

附录一：各个主要模块的引脚定义

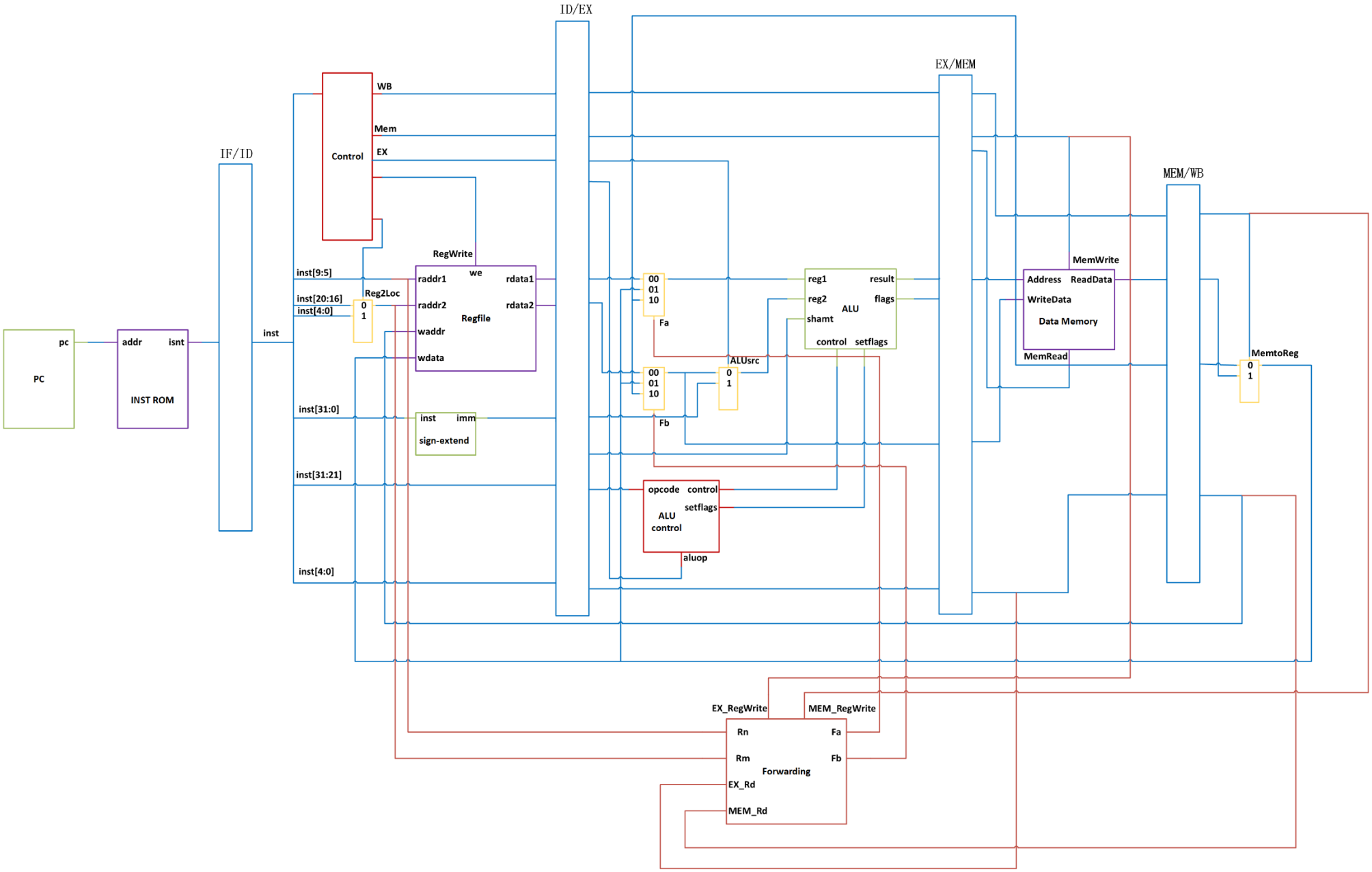
阶段	模块	输入/输出	接口名称	宽度	作用
取指阶段	PC	输入	rst	1	复位信号
		输入	clk	1	时钟信号
		输出	pc	64	要读取的指令地址
	INST ROM	输入	rst	1	复位信号
		输入	addr	64	要读取的指令地址
		输出	inst	32	读出的指令
	IF/ID	输入	rst	1	复位信号
		输入	clk	1	时钟信号
		输入	if_pc	64	取指阶段取得的指令对应的地址
		输入	if_inst	32	取指阶段取得的指令
		输出	id_pc	64	译码阶段取得的指令对应的地址
		输出	id_inst	32	译码阶段取得的指令
译码阶段	Regfile	输入	rst	1	复位信号，高电平有效
		输入	clk	1	时钟信号
		输入	raddr1	5	第一个读寄存器端口要读取的寄存器的地址
		输出	rdata1	64	第一个读寄存器端口输出的寄存器值
		输入	raddr2	5	第二个读寄存器端口要读取的寄存器的地址
		输出	rdata2	64	第二个读寄存器端口输出的寄存器值
		输入	we	1	写使能信号
		输入	waddr	5	写回阶段要写入的寄存器地址
		输入	wdata	64	写回阶段要写入的数据
	register_mux	输入	Ain	5	读寄存器 2 的寄存器号来自 Rt 字段 inst[4:0]
		输入	Bin	5	读寄存器 2 的寄存器号来自 Rm 字段 inst[20:16]
		输入	En	1	Reg2Loc 信号
		输出	out	5	选择结果
	sign_extend	输入	inst	32	读出的指令
		输出	imm	64	符号扩展后的指令字段
	control	输入	inst	32	产生的指令
		输出	Reg2Loc	1	读寄存器 2 的寄存器号来自 Rt 字段
		输出	MemRead	1	存储器读
		输出	MemtoReg	1	将存储器的结果写入寄存器
		输出	MemWrite	1	存储器写

		输出	ALUSrc	1	ALU 的第二个操作数为指令低 16 位的符号扩展
		输出	RegWrite	1	寄存器写
		输出	ALUOp	2	ALU 控制单元的控制信号
	ID/EX	输入	rst	1	复位信号，高电平有效
		输入	clk	1	时钟信号
		输入	id_opcode	11	译码阶段的指令的操作码字段，inst[31:21]
		输入	id_waddr	5	译码阶段的指令要写入的目的寄存器地址，inst[4:0]
		输入	id_reg1	64	译码阶段的指令要进行的运算的源操作数 1
		输入	id_reg2	64	译码阶段的指令要进行的运算的源操作数 2
		输入	id_shamt	6	译码阶段的指令要进行的运算的位移量
		输入	id_imm	6	译码阶段的指令要进行的运算的立即数
		输入	id_MemRead	1	译码阶段的存储器读
		输入	id_MemtoReg	1	译码阶段的将存储器的结果写入寄存器
		输入	id_MemWrite	1	译码阶段的存储器写
		输入	id_ALUSrc	1	译码阶段的 ALU 的第二个操作数为指令低 16 位的符号扩展
		输入	id_RegWrite	1	译码阶段的寄存器写
		输入	id_ALUOp	2	译码阶段的 ALU 控制单元的控制信号
		输入	id_Rn	5	译码阶段的源寄存器 1
		输入	id_Rm	5	译码阶段的源寄存器 2
		输出	ex_opcode	11	执行阶段的指令的操作码字段，inst[31:21]
		输出	ex_waddr	5	执行阶段的指令要写入的目的寄存器地址，inst[4:0]
		输出	ex_reg1	64	执行阶段的指令要进行的运算的源操作数 1
		输出	ex_reg2	64	执行阶段的指令要进行的运算的源操作数 2
		输出	ex_shamt	6	执行阶段的指令要进行的运算的位移量
		输出	ex_imm	6	执行阶段的指令要进行的运算的立即数
		输出	ex_MemRead	1	执行阶段的存储器读
		输出	ex_MemtoReg	1	执行阶段的将存储器的结果写入寄存器
		输出	ex_MemWrite	1	执行阶段的存储器写
		输出	ex_ALUSrc	1	执行阶段的 ALU 的第二个操作数为指令低 16 位的符号扩展
		输出	ex_RegWrite	1	执行阶段的寄存器写
		输出	ex_ALUOp	2	执行阶段的 ALU 控制单元的控制信号
		输出	ex_Rn	5	执行阶段的源寄存器 1
		输出	ex_Rm	5	执行阶段的源寄存器 2
执行阶段	ALU	输入	reg1	64	参与运算的源操作数 1
		输入	reg2	64	参与运算的源操作数 2
		输入	control	4	ALU 控制信号
		输入	shamt	6	移位量

		输入	setflags	1	是否设置标志位
		输出	result	64	ALU 运算结果
		输出	flags	4	NZVC 标志位
	Triple Mux0	输入	Ain	64	来自寄存器堆中的操作数
		输入	Bin	64	从数据存储器或者前面的 ALU 结果中旁路获得的操作数
		输入	Cin	64	由上一个 ALU 运算结果旁路获得的操作数
		输入	En	2	ForwardA 信号
		输出	out	64	选择结果
	Triple Mux1	输入	Ain	64	来自寄存器堆中的操作数
		输入	Bin	64	从数据存储器或者前面的 ALU 结果中旁路获得的操作数
		输入	Cin	64	由上一个 ALU 运算结果旁路获得的操作数
		输入	En	2	ForwardB 信号
		输出	out	64	选择结果
	Mux	输入	Ain	64	立即数
		输入	Bin	64	三路选择器获得的操作数
		输入	En	1	ALUsrc 信号
		输出	out	5	选择结果
	ALU control	输入	opcode	11	操作码字段，inst[31:21]
		输入	aluop	2	ALU control 控制信号
		输出	setflags	1	是否设置标志位
		输出	control	4	ALU 控制信号
	EX/MEM	输入	rst	1	复位信号，高电平有效
		输入	clk	1	时钟信号
		输入	ex_flags	1	执行阶段的 NZCV 标志位
		输入	ex_result	64	执行阶段的 ALU 运算结果
		输入	ex_reg2	64	执行阶段的参与运算的源操作数 2
		输入	ex_waddr	5	执行阶段的指令要写入的目的寄存器地址，inst[4:0]
		输入	ex_MemRead	1	执行阶段的存储器读
		输入	ex_MemtoReg	1	执行阶段的将存储器的结果写入寄存器
		输入	ex_MemWrite	1	执行阶段的存储器写
		输入	ex_RegWrite	1	执行阶段的寄存器写
		输出	mem_flags	1	访存阶段的 NZCV 标志位
		输出	mem_result	64	访存阶段的 ALU 运算结果
		输出	mem_reg2	64	访存阶段的参与运算的源操作数 2
		输出	mem_waddr	5	访存阶段的指令要写入的目的寄存器地址，inst[4:0]
		输出	mem_MemRead	1	访存阶段的存储器读
		输出	mem_MemtoReg	1	访存阶段的将存储器的结果写入寄存器

		输出	mem_MemWrite	1	访存阶段的存储器写
		输出	mem_RegWrite	1	访存阶段的寄存器写
访存阶段	DATA ROM	输入	rst	1	复位信号，高电平有效
		输入	clk	1	时钟信号
		输入	addr	64	存储器的读写地址
		输入	wdata	64	要写入的数据
		输入	rden	1	读取使能
		输入	wren	1	写入使能
		输出	rdata	64	读取输出的数据
	MEM/WB	输入	rst	1	复位信号，高电平有效
		输入	clk	1	时钟信号
		输入	mem_rdata	64	访存阶段数据存储器读取输出的数据
		输入	mem_result	64	访存阶段的 ALU 运算结果
		输入	mem_waddr	5	访存阶段的指令要写入的目的寄存器地址，inst[4:0]
		输入	mem_MemtoReg	1	访存阶段的将存储器的结果写入寄存器
		输入	mem_RegWrite	1	访存阶段的寄存器写
		输出	wb_rdata	64	写回阶段数据存储器读取输出的数据
		输出	wb_result	64	写回阶段的 ALU 运算结果
		输出	wb_waddr	5	写回阶段的指令要写入的目的寄存器地址，inst[4:0]
		输出	wb_MemtoReg	1	写回阶段的将存储器的结果写入寄存器
		输出	wb_RegWrite	1	写回阶段的寄存器写
写回阶段	Mux	输入	Ain	64	存储器中取得的数
		输入	Bin	64	ALU 计算产生的数
		输入	En	1	MemtoReg 信号
		输出	out	5	选择结果
其他模块	forwarding	输入	Rn	5	源寄存器 1
		输入	Rm	5	源寄存器 2
		输入	ex_Rd	5	执行阶段的目的寄存器
		输入	mem_Rd	5	访存阶段的目的寄存器
		输入	ex_RegWrite	1	执行阶段的寄存器写信号
		输入	mem_RegWrite	1	访存阶段的寄存器写信号
		输出	Fa	2	控制信号 a
		输出	Fb	2	控制信号 b

附录二：主要的数据通路


ID/EX
EX/MEM
MEM/WB
IF/ID
WB
Mem
EX
Control
RegWrite
we
raddr1
raddr2
waddr
wdata
rdata1
rdata2
Regfile
inst[9:5]
inst[20:16]
inst[4:0]
Reg2Loc
0
1
inst[31:0]
inst
imm
sign-extend
inst[31:21]
inst[4:0]
pc
PC
addr
isnt
INST ROM
inst
00
01
10
Fa
Fb
ALUsrc
reg1
reg2
shamt
ALU
control
result
flags
setflags
MemWrite
Address
ReadData
WriteData
Data Memory
MemRead
MemtoReg
opcode
control
setflags
ALU
control
aluop
EX_RegWrite
MEM_RegWrite
Rn
Rm
Fa
Fb
Forwarding
EX_Rd
MEM_Rd

附录三：开发板测试中数码管显示结果

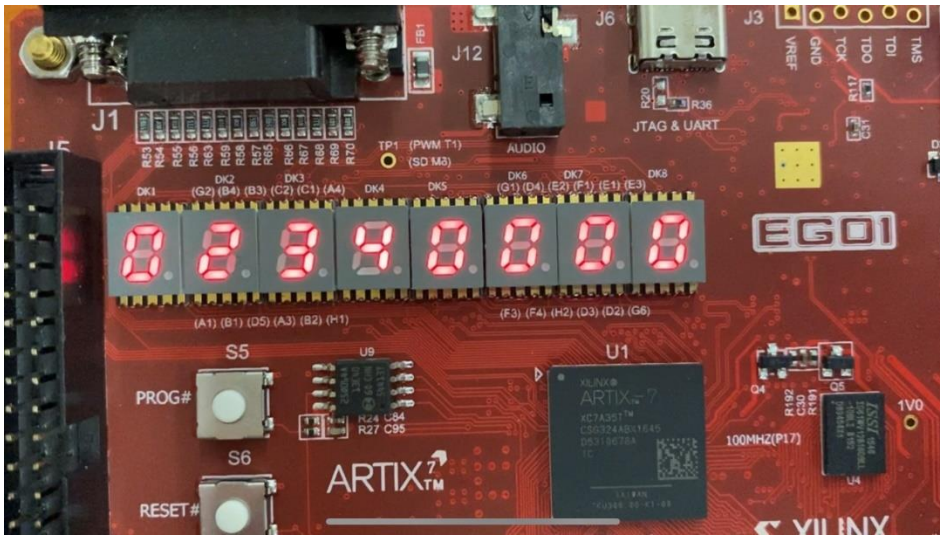


图 0-1 显示内容：0234 0000

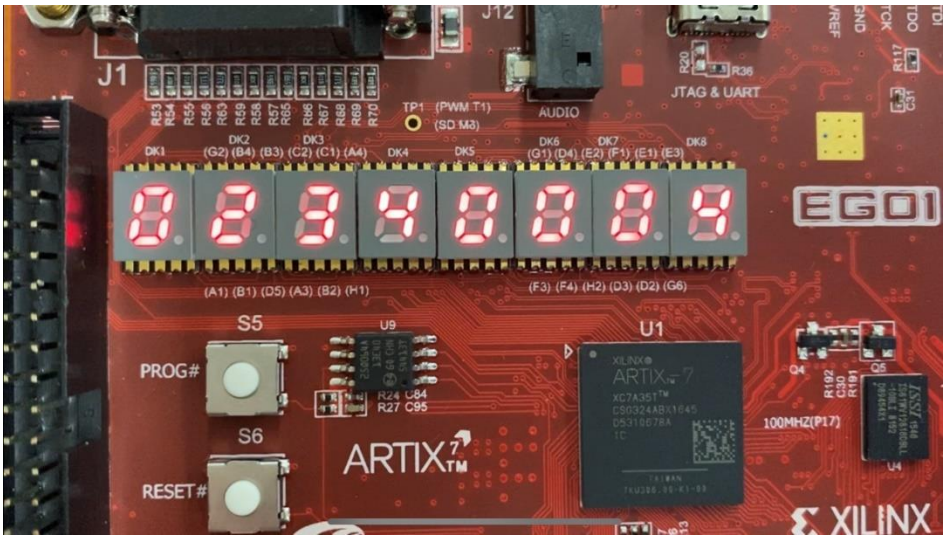


图 0-2 显示内容：0234 0004

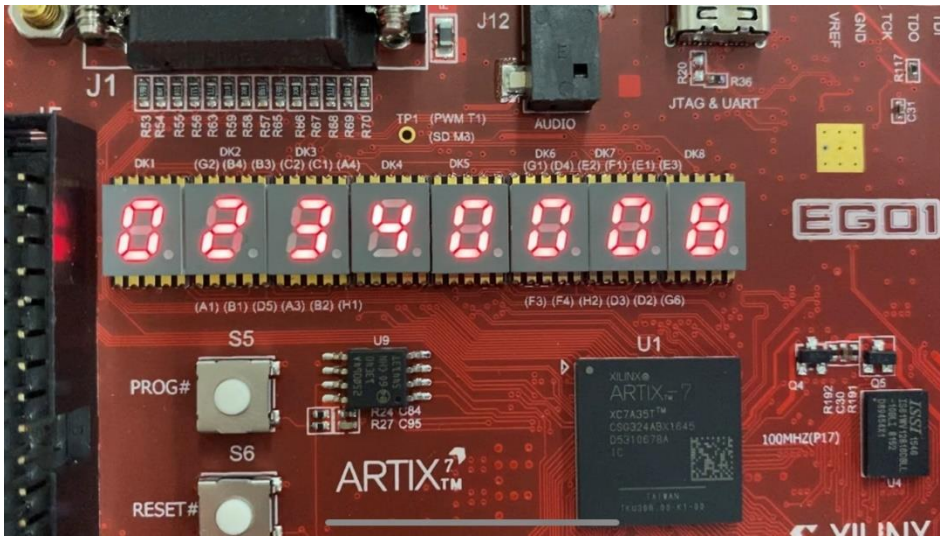


图 0-3 显示内容：0234 0008

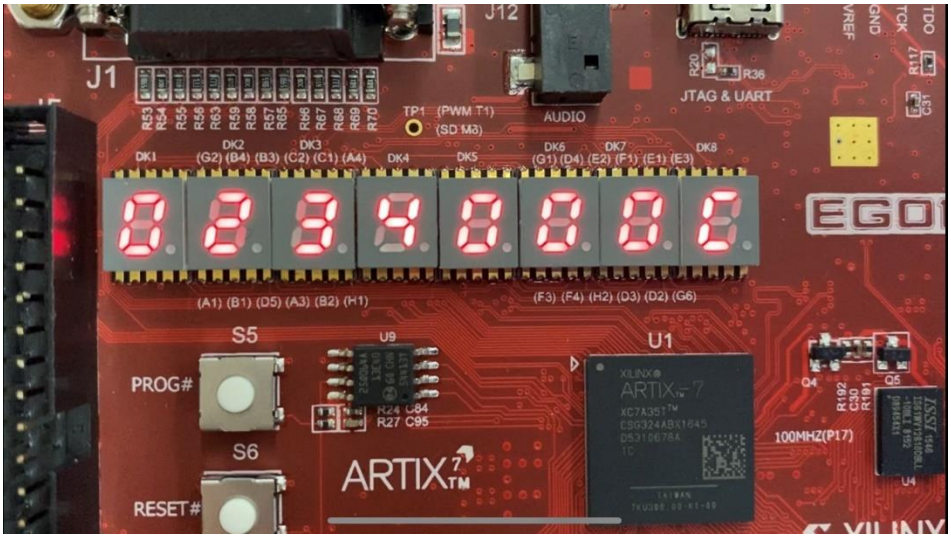


图 0-4 显示内容：0234 000c

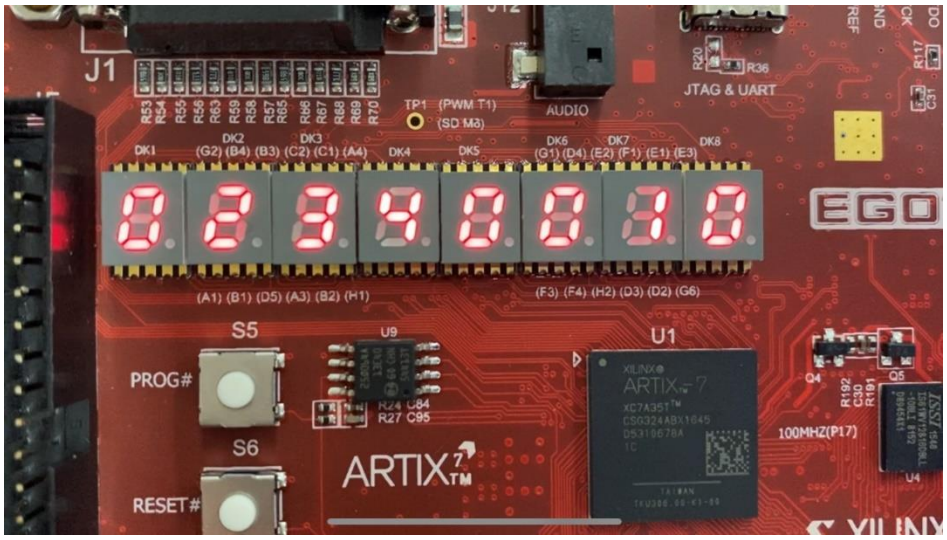


图 0-5 显示内容：0234 0010

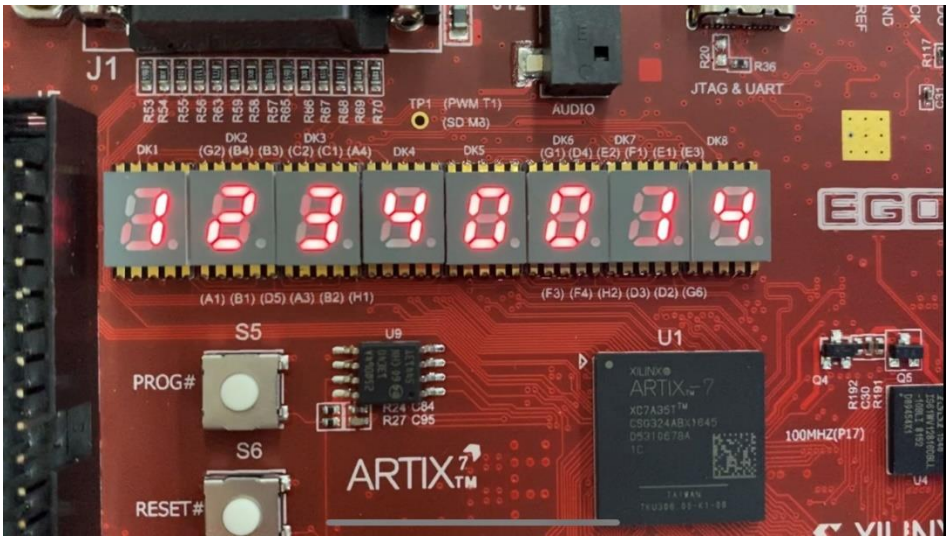


图 0-6 显示内容：1234 0014

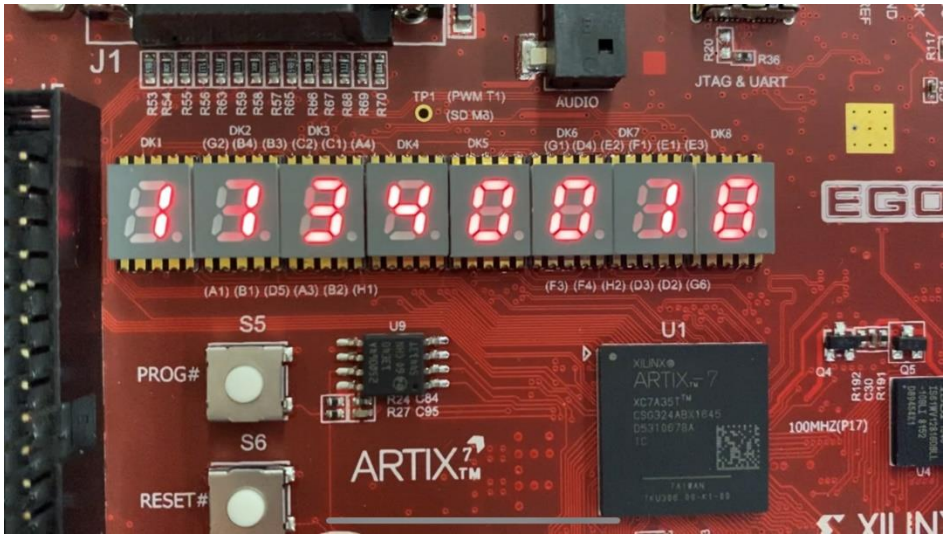


图 0-7 显示内容：1134 0018

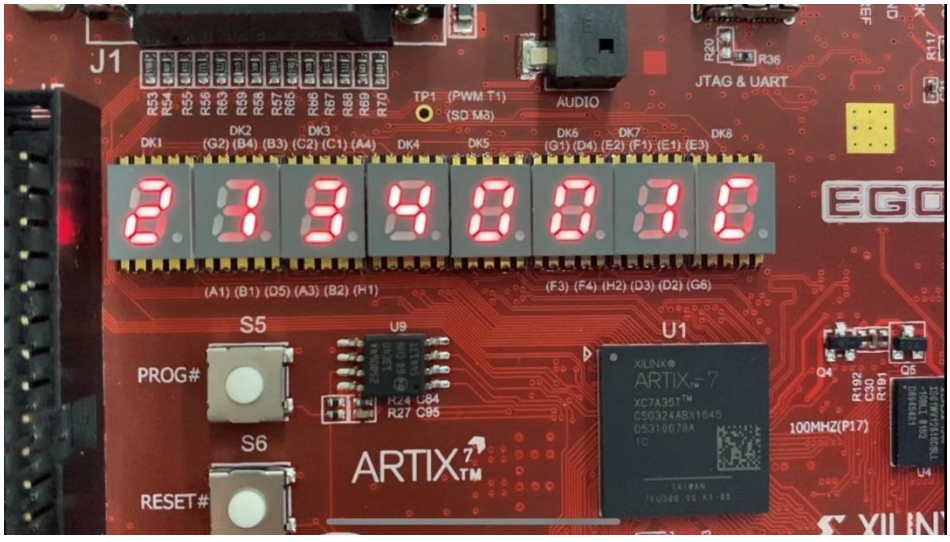


图 0-8 显示内容：2134 001c

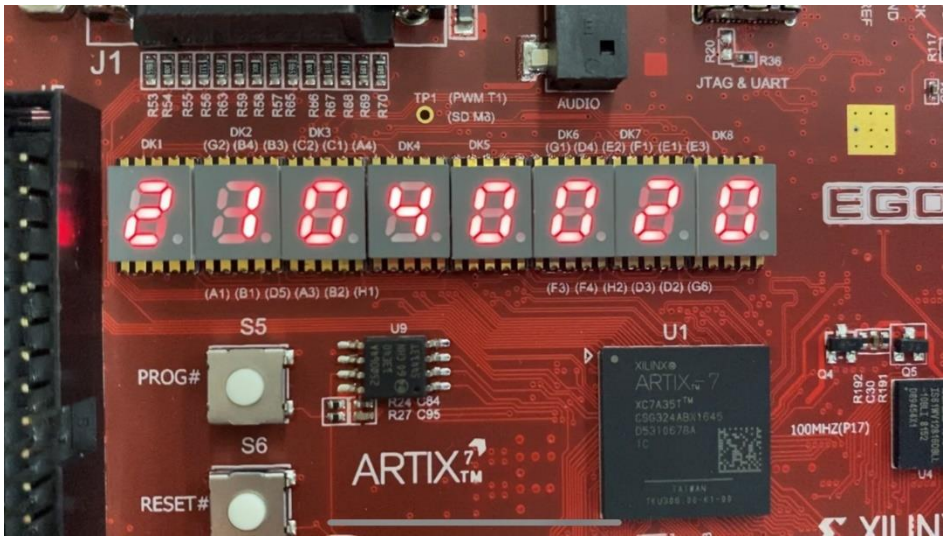


图 0-9 显示内容：2104 0020

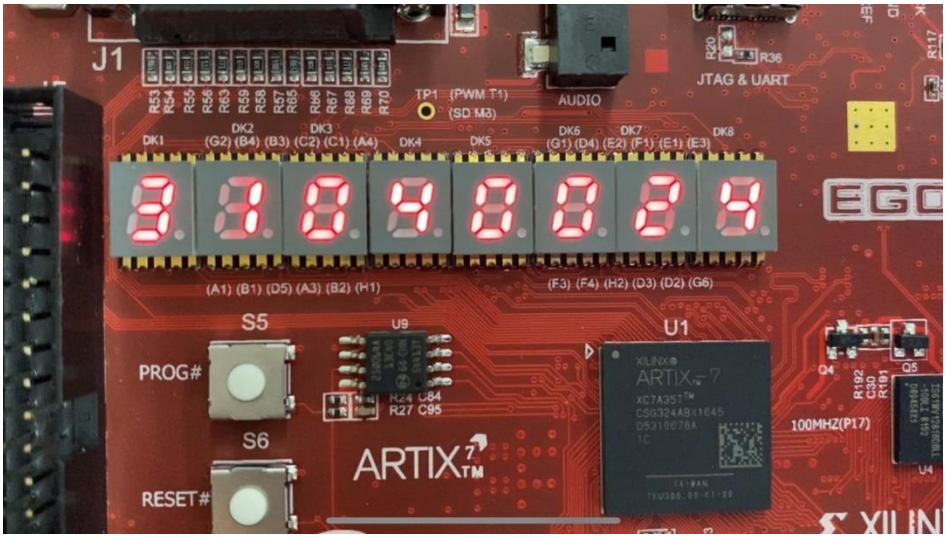


图 0-10 显示内容：3104 0024

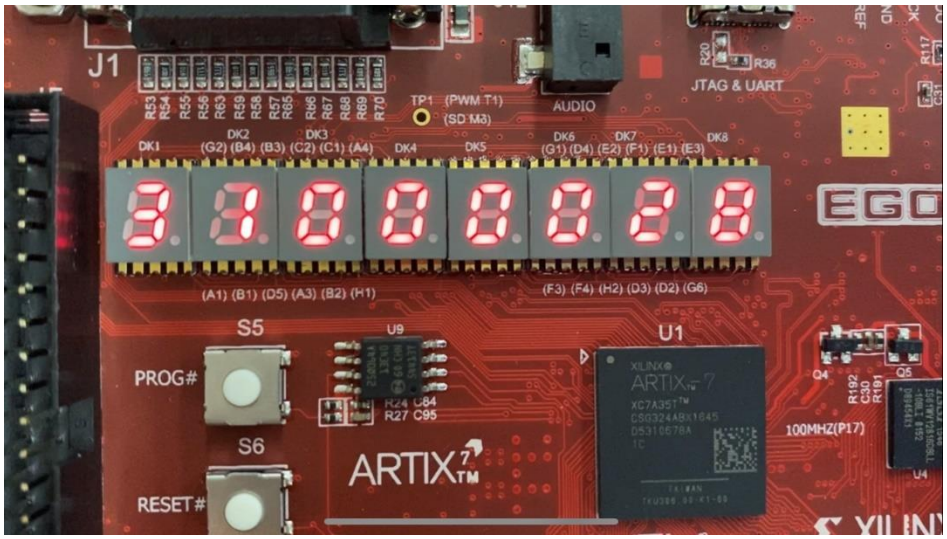


图 0-11 显示内容：3100 0028

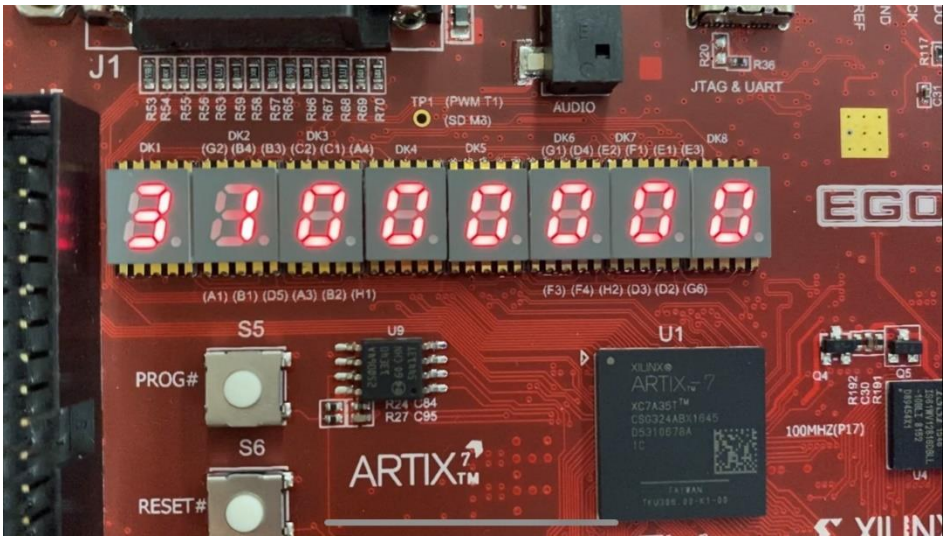


图 0-12 显示内容：3100 0000

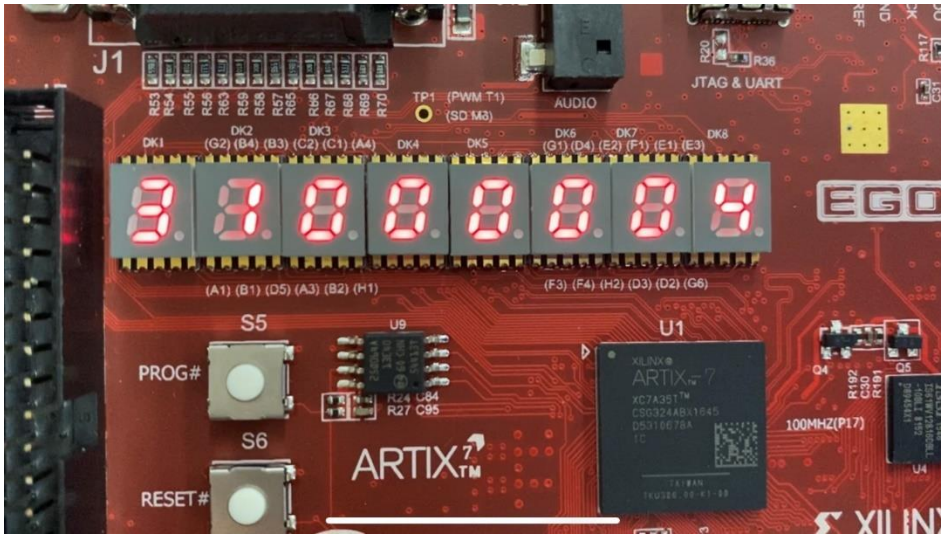


图 0-13 显示内容：3100 0004

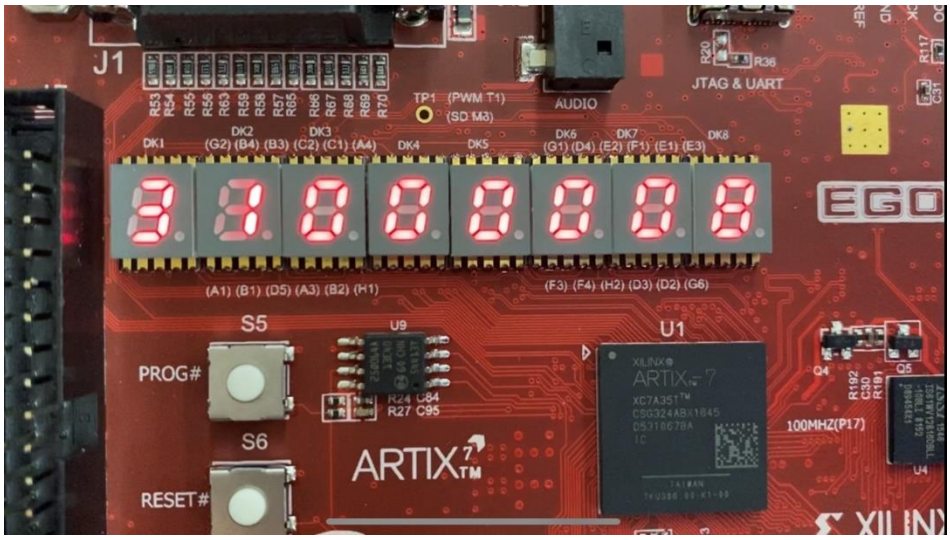


图 0-14 显示内容：3100 0008

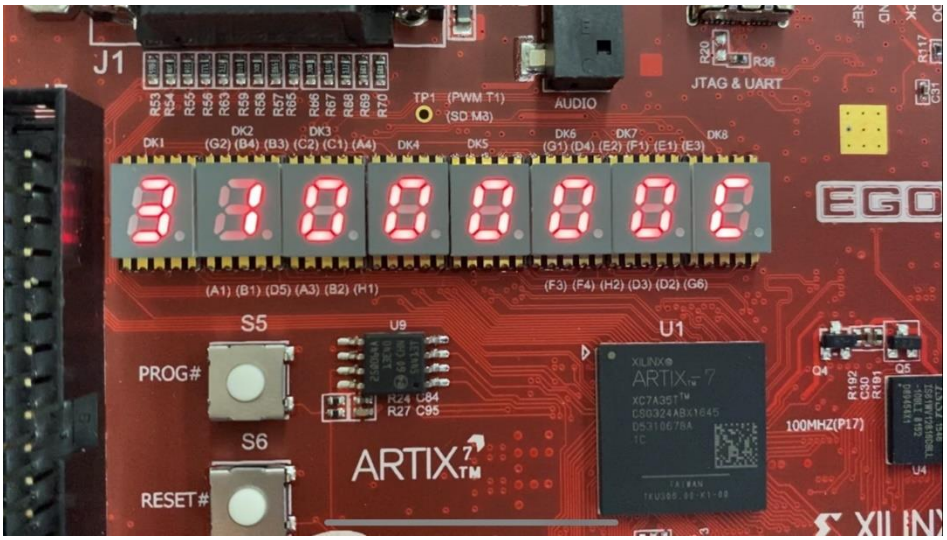


图 0-15 显示内容：3100 000c

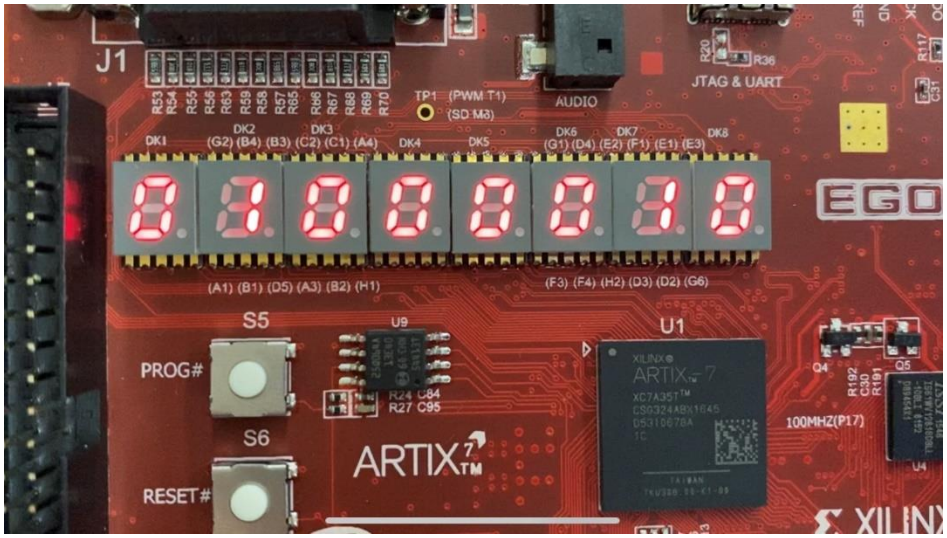


图 0-16 显示内容：0100 0010