摘要

当前无论国外还是国内，在 SoC 设计领域已展开激烈的竞争。作为现代计算机专业的大学生，应该强调在面向应用的基础上，更加注重自己面向系统的分析与设计能力的提高，于是东南大学计算机学院在五年前，开设了“计算机系统综合课程设计”实践环节，以此提高本科生软硬件综合的实践能力。

本论文针对“国家十一五规划教材《计算机系统综合设计》”的示范原型系统 MiniSys SoC 在乘除运算方面的缺失，为其增加专用的高性能乘除法运算单元。对原来 MiniSys CPU 中不利于乘除法实现的部分进行了修改，设计实现了适合于 MiniSys CPU 的多周期乘除法运算单元，并且进行了仿真测试。

此外，为了让 MiniSys SoC 系统有更丰富的扩展资源，设计实现了三个新的 IO 接口部件。第一个是 IO 接口部件是为了解决 FPGA 芯片中存储容量太小的问题，设计完成一个专用的 SDRAM 控制器。它能够完成对 SDRAM 的初始化、读、写、刷新、预充的操作，实现了对 SDRAM 的控制。第二个是 I^2C 总线控制器，可以使系统的可扩展性更强。本文设计完成一个适合该系统的 32 位数据收发的 I^2C 总线控制器。最后是新增的 LCD 控制器，可以增加系统的输出能力。

本文所设计并实现的所有部件，已经整合到原有的 MiniSys SoC 系统中，并加以优化，使该系统有了更加完善的功能。

关键词：乘法运算单元 除法运算单元　SDRAM 控制器　I^2C 总线控制器 LCD 控制器

Abstract

At present, there are fierce competitions in SOC design field in the world .As modern computer science students, we should pay more attention on advancing our system analyze and design ability. To this purpose, the computer science academe of Southeast University provided a practical course, called “Computer System Design”, to improve the practice ability of the undergraduates.

The prototype MiniSys SoC system didn’t have any multiply or divide computing hardware. Programmers should design multiply function to implement multiply operation. This will be low-performance. On this paper, I have designed a special high-performance multiplication and division unit.

In addition, three kinds of IO interface units were designed to wild the application of MiniSys Soc. One of them is the SDRAM Controller. Different to SRAM,SDRAM usually need a controller to work. I will describe the design detail. Another is I^2C-bus Controller.I^2C-bus supports any IC fabrication process. Two wires, serial data and serial clock, carry information between the devices. The last is LCD Controller.

In this thesis, the design of every unit in detail and some optimizing for CPU has done were described.

KEY WORD : Multiply , Division ,SDRAM Controller,I^2C -bus Controller ,LCD Controller

目录

第 1 章 绪论

1.1 引言

微电子技术与计算机技术的发展历史是一个不断创新的过程，这种创新包括原始创新、技术创新和应用创新等。每一项创新都能开拓出一个新的领域，带来新的巨大的市场，对我们的生产、生活方式产生重大的影响。

自集成电路发明以后，集成电路芯片的发展基本上遵循的 Intel 公司创始人之一的 Gordon Moore 1965 年预言的摩尔定律，即 IC 上可容纳的晶体管数目，约每隔 18 个月便会增加一倍，性能也将提升一倍[1]。自 20 世纪下半叶以来，微电子在技术得到了迅猛的发展，集成电路设计和工艺技术水平有了极大的提高，从而使得将原有 IC 组成的电子系统集成在一个单片硅片上成为可能，构成了所谓的片上系统（System on a Chip,即 SoC），系统芯片不再是一种功能单一的单元电路，而是将信号采集、处理、和输入输出等完整的系统功能集成在一起，成为一个专用功能的电子系统芯片，是一种高度集成化、固件化的系统集成技术。

如何界定 SoC，认识并未统一。但可以归纳如下：

1. SoC 应由可设计重用的 IP 核组成，IP 核是具有复杂系统功能的能够独立出售的 VLSI 块；
2. IP 核应采用深亚微米以上工艺技术；
3. SoC 中可以有多个 MPU、DSP、MCU 或其复合的 IP 核。

SoC 从整个系统的角度出发，把处理机制、模型算法、芯片构成、各层次电路，直至器件的设计紧密的结合起来，在单个（或少数几个）芯片上完成整个系统的功能。SoC 的设计以 IP 核为基础，以分层次的硬件描述语言为系统功能和结构的主要核技术手段，借助于以计算机为平台的 EDA 工具进行。研究表明，与 IC 组成的系统相比，由于 SoC 设计能够结合并全盘考虑整个系统的各种情况，因而可以在同样的工艺技术条件下，实现更高性能的系统指标[2]。SoC 技术，也大大促进了软硬件协同设计及计算机系统设计自动化的发展，它是集成电路发展的必然，也是 IC 产业未来的发展。

当前芯片设计业正面临着一系列的挑战，系统芯片 SoC 已经成为 IC 设计业界的焦点，SoC 性能越来越强，规模越来越大。SoC 芯片的规模一般远大于普通的 ASIC，同时由于深亚微米工艺带来的设计困难等，使得 SoC 设计的复杂度大大提高。在 SoC 设计中，仿真与验证是 SoC 设计流程中最复杂、最耗时的环节，约占整个芯片开发周期的 50%～80%，采用先进的设计与仿真验证方法成为 SoC 设计成功的关键。SoC 技术的发展趋势是基于 SoC 开发平台，基于平台的设计是一种可以达到最大程度系统重用的面向集成的设计方法，分离 IP 核开发与系统集成成果，不断重整价值链，在关注面积、延迟、功耗的基础上，向成品率、可靠性、EMI 噪声、成本、易用性等转移，使系统级集成能力快速发展。

作为现代计算机专业的大学生，应该强调在面向应用的基础上，更加注重自己面向系统的分析与设计能力的提高，这就要求当代高校计算机教育应该从最初的培养纯应用软件的开发，逐渐向系统软件和硬件相结合的开发方向发展。同时，当前 IT 业也更加突出的表现出对具有综合创新能力的人才的急切需求。认识到

这一点，国外的著名高校，如加州大学伯克利分校等，纷纷开展了自己的本科生软硬件综合能力实践，成为了目前计算机专业的发展趋势之一。我校计算机科学与工程学院也在国内率先借鉴这一思路，开设了“计算机系统综合课程设计”实践环节，使学生在嵌入式专用芯片设计、计算机接口电路的设计、嵌入式系统的软硬件协同设计和计算机系统软件设计等方面得到很好的训练，培养出高素质的，具有创新思维与综合能力，符合当前时代需要的人才。

1.2 SOC 设计的现状

当前无论在国外还是国内，在 SoC 设计领域已展开激烈的竞争。SOC 按指令集来划分主要分 x86 系列（如 SiS550）、ARM 系列（如 OMAP）、MIPS 系列（如 Au1500）和类指令系列（如 M3 Core）等几类，每一类都各有千秋。国内研制开发者主要基于后两者，如中科院计算所中科 SoC（基于龙芯核，兼容 MIPS III指令集）、北大众志（定义少许特殊指令）、方舟 2 号（自定义指令集）、国芯 C3 Core（继承 M3 Core）等。开发拥有自主知识产权的处理器核、核心 IP 和总线架构，同时又保证兼容性（集成第三方 IP），将使我国 SoC 发展具有更强的竞争力，从而带动国内 IC 产业往深度、广度方向发展。

1.3 MiniSys SoC 概述及论文的工作

我院的“计算机系统综合课程设计”经过五年的摸索和实践，已经趋于成熟，根据多年教学实践总结的讲义而编写的教材已经被定为国家十一五规划教材。经过五年的课程教学积累，众多学长们已经为该 MiniSys SoC 系统实现了 CPU 设计和 LED、键盘、串口、定时/计数器、看门狗等外围设备，使 MiniSys SoC 系统初具雏形，并具有了一定的可应用性，但是这还远不能满足实现应用的需求。

为了更加完善 MiniSys SoC 系统结构，增加该系统的实用性，并能够最终成为一个完整的嵌入式开发平台，需要对该系统做必要的补充，因此，本文的主要工作包括：

1. 优化现有的 CPU 流水结构，并增加对新增部件的指令的支持；
2. 设计并实现适合 MiniSys CPU 的乘/除法运算单元及相关指令；
3. 设计并实现 SDRAM 控制器；
4. 设计并实现 I^2C 控制器；
5. 设计并实现 LCD 控制器。

以上的工作，不但能直接的增加系统中 CPU 的性能和外设的种类，还能为今后进一步优化 CPU 架构（如增加 Cache 和虚拟内存，扩大可用存储空间等）奠定基础，增加外围接口的可扩展性。

1.4 本论文内容安排

本论文共分 6 章，以下是其他各章介绍，了解这些可以帮助读者取舍内容。

- 第 2 章从整个 MiniSys SoC 系统的角度，介绍了该系统的整体架构以及将在本文中新增部件的设计说明。
- 第 3 章介绍适合 MiniSys SoC 系统的乘除法运算单元的设计和实现。该章描述了有符号、无符号乘除法运算单元的实现，并将其整合成一个运算部件。
- 第 4 章介绍 SDRAM 控制器、I^2C 总线控制器、LCD 控制器的设计和实现。
- 第 5 章把前两章设计的部件整合到 MiniSys SoC 系统中，并进行仿真测试。
- 第 6 章对本次毕业设计工作的总结以及接下来要做的工作的介绍。

第 2 章 MiniSys SoC 的架构

2.1 MiniSys SoC 的总体架构

MiniSys 系统包含一个以 32 位 RISC 型处理器为核心，自带多个外围部件的 SoC 芯片和相关的系统软件，系统软件包括以便于上层软件编程而提供的系统功能调用接口为主体的 BIOS 和一个简单的 MiniSys 汇编语言汇编器。MiniSys SoC 的硬件功能结构如图 2.1 所示。

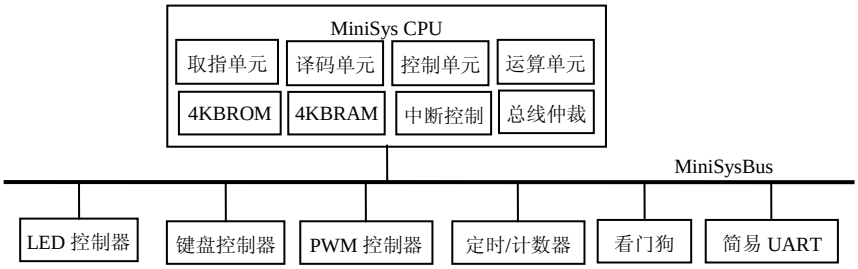

图 2.1 MiniSys 硬件功能结构图

MiniSys CPU 有 32 个 32 位寄存器，32 位数据线（对 I/O 只有 16 位数据线）和 16 位地址线。外围部件包括一个 4 位 7 段 LED 控制器，一个 4×4 键盘控制器，两个 16 位定时/计数器、一个 16 位 PWM 控制器、一个看门口控制器和一个简易 UART 串行通信控制器。

MiniSys SoC 采用了 32 位 MIPS CPU 的 31 条常用指令，CPU 大致采用 MIPS 的体系结构，只是在部分细节上作了一些调整。32 个 32 位寄存器除了 5 个寄存器被固定功能外，其余的都可以做通用寄存器。

系统的存储结构采用哈佛结构，在 MiniSys CPU 中包含片内的 4KB ROM 和 4KB RAM，它们都采用字节编址，但以 32 位（4 字节）为一个存储单元，即他们和 CPU 之间的数据交换都以 32 位为单位进行。

MiniSys SoC 的 I/O 空间编址采用与存储器统一编址方式，既将整个地址空间分为两个部分，一部分作为访问 RAM 的存储空间，另一部分作为访问 I/O 部件的 I/O 空间，因此，对 I/O 部件的访问采用与存储器访问相同的指令格式。

系统内部提供两个中断源的控制电路，两个中断源为 INT0 和 INT1，其中 INT0 的优先级高于 INT1。

系统提供用于堆栈操作的 SP 寄存器，但没有提供压栈和退栈指令，因此对于堆栈的操作需要用软件实现。同样，由于系统没有提供乘除指令和浮点运算单元，因此有关这些方面的功能也需要编译器利用库文件的形式提供软件仿真。

2.2 流水线 CPU 的结构

2.2.1 MiniSys CPU 总体结构图

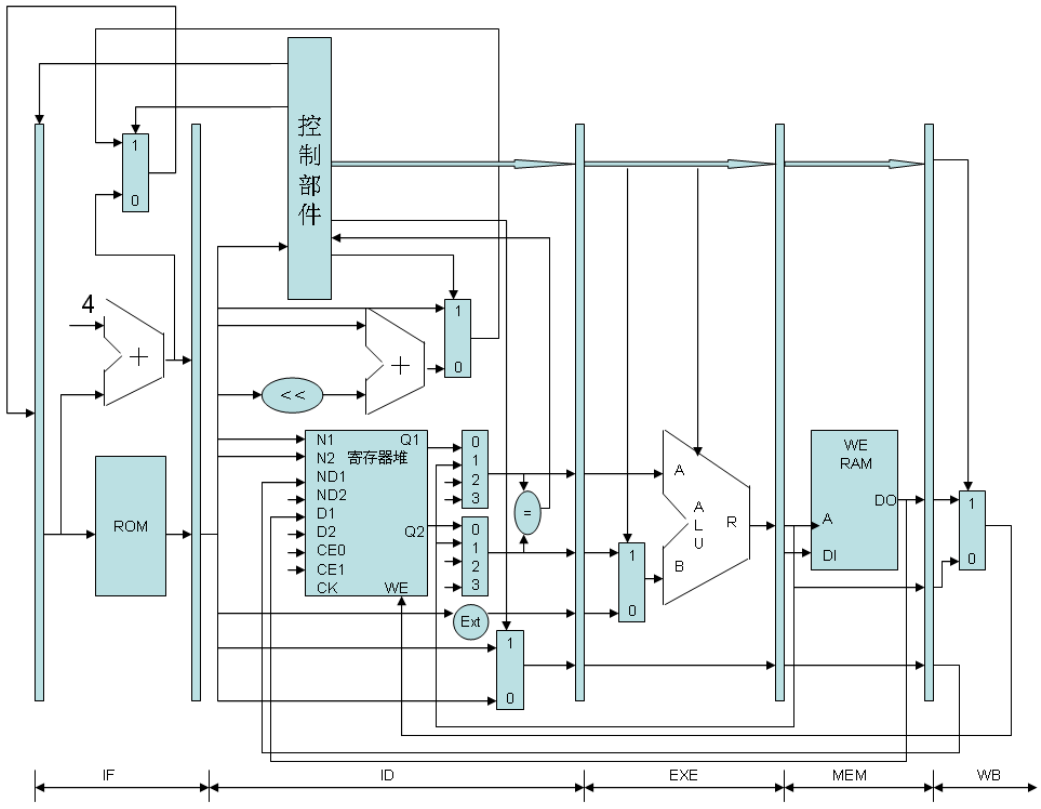

图 2.2 流水线 CPU 逻辑电路图

流水线 MiniSys CPU 指令执行分成 5 个阶段：

- IF 阶段
  1) 计算下一条指令的指令地址 PC；
  2) 从存储器读出当前需要处理的指令。
- ID 阶段
  1) 指令译码；
  2) 处理数据相关；
  3) 判断分支指令的分支条件；
  4) 把操作数从寄存器堆中读出。
- EXE 阶段
  1) 使用 ALU 完成算术逻辑运算；
  2) 把运算结果保存到寄存器中，供 MEM 执行阶段使用；
  3) 给出各种控制信号。
- MEM 阶段
  1) 给出各种存取存储器的控制信号，并完成存储器的存取任务；
  2) 把从存储器中读入的数据保存到寄存器中，供 WB 执行阶段使用；
  3) 给出各种协处理器 CP0 的控制信号，并完成对协处理器 CP0 存取工

作。

- WB 阶段
  1) 将指令执行结果写入到寄存器堆中；
  2) 将乘法运算和除法运算的执行结果定写入到寄存器 HI 和寄存器 LO 中。

2.2.2 相关问题处理

流水线 CPU 中最重要和最复杂的工作就是相关问题的处理。相关问题包括：结构相关、数据相关和控制相关。

1) 结构相关：当 CPU 的硬件结构不能够同时支持几条指令的执行时，这几条指令将发生结构相关．例如只有一个存储器模块时，取指令与 load/store 指令访问存储器不能同时进行。
2) 数据相关：当一条指令的执行依赖于前面某一条指令执行的结果时，这两条指令将发生数据相关。
3) 控制相关：当指令执行到分支或者其他引起程序计数器 PC 值发生变化的指令时，这些跳转指令将和后面的若干条指令发生控制相关。

MiniSys CPU 所面临的相关问题

结构相关：

MiniSys CPU 中出现结构相关的情况只有一种，即有 MUL 指令时，可能同时对寄存器堆两个单元进行写操作。以前 CPU 寄存器堆一个周期只能写一个单元。现在的寄存器堆设计成了一个周期能同时写两单元。这样解决了结构相关。

数据相关：

- 当前指令 I_n 执行到 ID 阶段需要从寄存器堆中取得数据时，如果前第三条指令 I_{n-3} 不是乘法或者除法指令，则已经运行到了 WB 执行阶段。此时 I_{n-3} 的运行结果在该周期中间写入到寄存器堆中，所以，指令 I_n 就可以直接从寄存器堆中读出需要的数据。这样，就不存在数据相关问题。如图 2.3 所示。

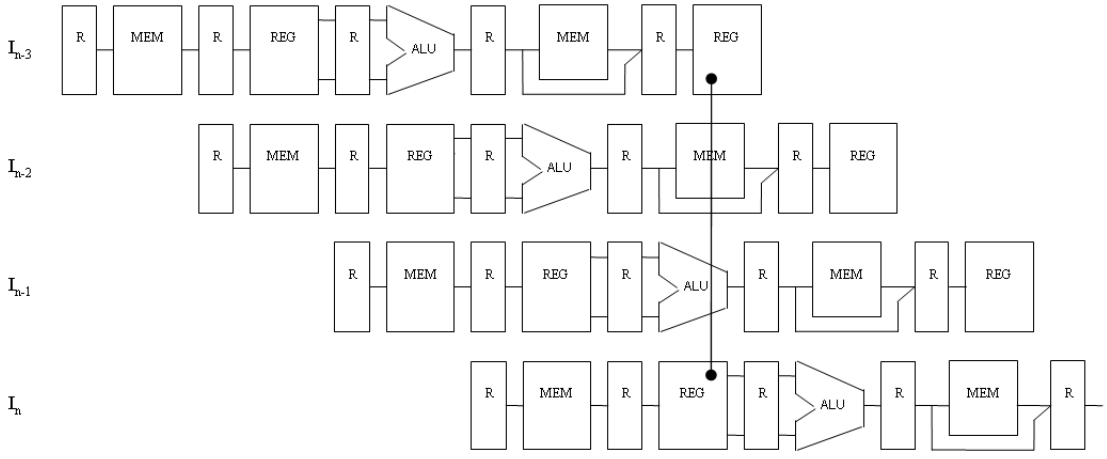

图 2.3 当前指令与前面第三条指令的数据关系

◆ 当前指令 I_n 执行到 ID 阶段需要从寄存器堆中取得数据时，如果前第二条指令 I_{n-3} 不是乘法或者除法指令，则已经运行到了 MEM 执行阶段。此时，无论指令 I_{n-2} 的运行结果来源如何，都已经得到，只是还没有来得及进入到 WB 执行阶段来写入到寄存器堆中。如果指令 I_n 需要使用指令 I_{n-2} 的运行结果，就需要把指令 I_{n-2} 的运行结果提前引入到指令 I_n 的 ID 阶段输入数据寄存器的数据输入端。如图 2.4 所示。

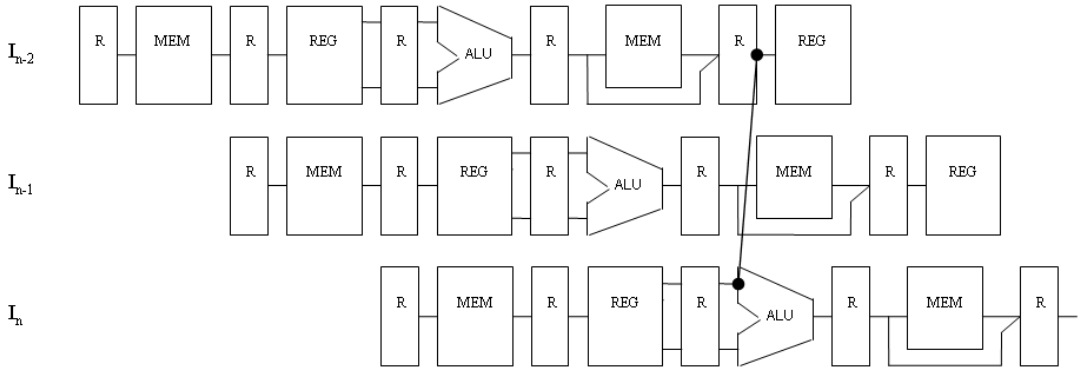

图 2.4 当前指令与前面第二条指令的数据相关

◆ 当前指令 I_n 执行到 ID 阶段需要从寄存器堆中取得数据时，前面第一条指令 I_{n-1} 已经运行到了 EXE 执行阶段：

1) 如果指令 I_{n-1} 不是 Load 或者乘法或者除法指令，并且运行结果数据来自于 ID 执行阶段或者 EXE 执行阶段，则此时已经得到了运行结果。如果指令 I_n 需要使用指令 I_{n-1} 的运行结果，就需要把指令 I_{n-1} 的运行结果提前引入到指令 I_n 的 ID 阶段输入数据寄存器的数据输入端。如图 2.5 所示。
2) 如果指令 I_{n-1} 不是 Load 或者乘法或者除法指令，并且运行结果数据来自于 MEM 执行阶段，则此时还没有得到运算结果。如果指令 I_n 需要使用指令 I_{n-1} 的运行结果，则需要阻塞等待，直至指令 I_{n-1} 的运行结果完成。

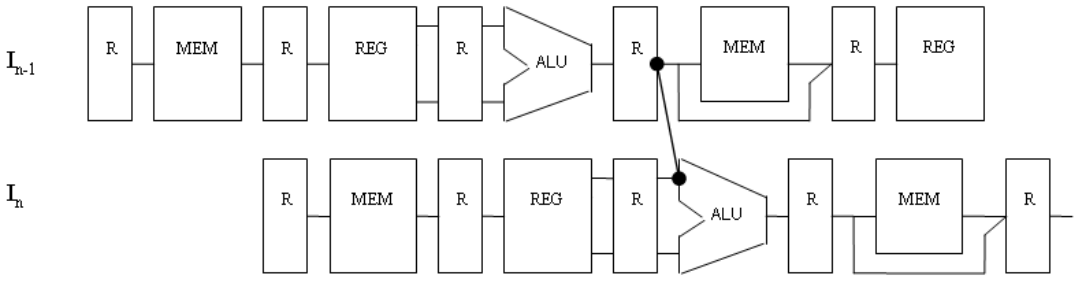

图 2.5 当前批令与前面第一条指令的数据相关

◆ 当前指令 I_n 执行到 ID 阶段需要从寄存器堆中取得数据时，指令需要的数据来自于前面的 Load 或者乘法指令或者除法指令，并且，乘法指令或者除法指令还没有完成执行，那么，指令 I_n 就需要阻塞等待，真到乘法或者除法指令完成执行并得到运算结果。

控制相关：

当指令流执行到分支指令或者跳转指令的时候，将产生控制相关。此时，并不能够简单地执行分支指令或者跳转指令后面的指令，而是只有首先判断了分支指令中的条件是否成立之后，才能够决定下面将从何处开始执行指令。

本文采用无延迟方式[1]解决控制相关，分支条件将在 ID 执行阶段来判断。然后，使用判断的结果来决定下一条指令从何处执行。

2.3 I/O 接口模块的结构

2.3.1 I/O 端口地址与 I/O 地址空间设计

在计算机系统中，外设不会直接挂在系统总线或者 CPU 上，这是因为外设在信号种类、时序等方面往往会千差万别，甚至电平特性都会与 CPU 或系统总线不同，因此，外设需要相应的接口电路来完成与系统总线之间的各种转换。在各种接口电路中，会设计一些供 CPU 直接存取访问的寄存器或特定电路，它们称作为 I/O 端口。这些端口有的负责接收 CPU 输出的 I/O 命令（命令寄存器）和数据，有的存放向 CPU 输送的数据和状态（状态寄存器）。对 I/O 接口的访问实际上是对接口电路上的 I/O 端口进行访问。为了区分不同的 I/O 端口，通常对这些端口像存储单元一样进行编号，形成 I/O 端口地址。

在计算机领域，对于 I/O 地址空间的设计通常采用两种方案，一种是 I/O 与存储器统一编址，一种是 I/O 独立编址。[3]现在来分析一下 MiniSys SoC 系统的具体情况，MiniSys SoC 指令系统中没有专门的 I/O 指令，因此只能使用 LW 和 SW 两条指令来进行 RAM 访问和 I/O 访问，这就决定了 MiniSys SoC 只能采用 I/O 统一编址方式。

MiniSys SoC 系统使用 16 位地址线，尽管各类与地址有关的指令计算出的地址是 32 位，但因为系统只有 4KB ROM 和 4KB RAM，所以实际上只用低 16 位地址线，而 16 位地址线能寻址的空间为 64KB（2^{16}）。在这 64KB 的地址空间中，将最后的 256 字节分配给 I/O 端口，即 FF00H~FFFFH 的地址空间。如此以来，64KB 的地址空间被划分成如图 2.6 所示的 3 个分区，其中阴影部分没有使用。

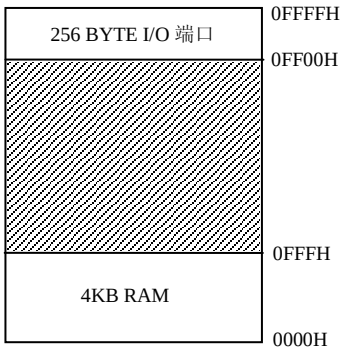

图 2.6 MinSys RAM 与 I/O 地址空间

由于地址空间被分区，因此对于 LW 和 SW 指令就可能产生两种访问，一种

是存储器访问，一种是 I/O 访问。

2.3.2 MiniSysBus 与总线控制模块设计

在设计 MiniSysBus 的时候采用集中译码方式，地址译码单元将 16 位地址线（32 位地址的低 16 位）中的高 12 位译码成枚举的 6 个片选信号 XXXCtrl，用于选择 6 个不同的接口电路。这高 12 根地址线组成的地址作为接口电路的基地址，地址低端（低 4 位）作为端口号。接口基地址加上端口号，最终形成 I/O 端口地址。对于存储器的访问，由于只有 4KB 字节，所以将 16 位地址线的低 12 位作为访存地址线。要注意片选信号的作用，接口电路只有当片选信号有效的时候才能和系统总线交换数据。表 2.1 所示列出了 MiniSys SoC 系统现有的各接口电路的基地址。

根据表 2.1 和地址线低 4 位做端口号的约定，所设计的接口部件最多只能有 16 个字节端口或者 8 个字端口，由于 I/O 数据线是 16 位的，所以使用字端口。

表 2.1 现有接口基地址分配表

部件名称	片选信号	接口基地址
4 位 7 段 LED 数码管	LEDCtrl	0FF00H
4×4 键盘控制器	KEYCtrl	0FF10H
定时/计数器	CTCCtrl	0FF20H
PWM 控制器	PEMCtrl	0FF30H
UART 串行通信控制器	UARTCtrl	0FF40H
看门狗控制器	WDTCtrl	0FF50H

在总线时序方面，由于时钟上升沿取指，而 Lw 和 Sw 指令得到访问的地址（在执行单元计算出来）还需要一段时间的延迟，因此，对总线上外设的读写只能在时钟下降沿完成。

由于本文中将为 MiniSys SoC 系统设计三个新的 I/O 接口，所以需要为这些新 I/O 接口分配片选信号及 I/O 基地址。新的接口基地址分配见表 2.2。在这要说明的是，将看门狗控制器基地址由 0FF50H 换成 0FF70H，是因为看门狗可以使系统复位，通常系统设计都会将它放在一个特殊位置，比如最后一个基地址，来引起用户注意。

表 2.2 新的接口基地址分配表

部件名称	片选信号	接口基地址
4 位 7 段 LED 数码管	LEDCtrl	0FF00H
4×4 键盘控制器	KEYCtrl	0FF10H
定时/计数器	CTCCtrl	0FF20H
PWM 控制器	PEMCtrl	0FF30H
UART 串行通信控制器	UARTCtrl	0FF40H
I^2C 总线控制器	IICCtrl	0FF50H
LCD 控制器	LCDCtrl	0FF60H
看门狗控制器	WDTCtrl	0FF70H

第 3 章 乘/除法运算单元的设计与实现

3.1 乘法运算单元的设计与实现

3.1.1 两种常用的乘法算法

1. Booth 算法[3]

Booth 算法主要用于补码的相乘。现将 Booth 归纳如下：

1) 符号位参加运算；
2) 在乘数最低位后面增加一位附加位（B_{-1}），初值为 0；
3) 部分积初值为 0；
4) 每次以前一次部分积为基础，根据表 3.1 判断相邻两位之差（低位减高位）决定+$[A]_{补}$、+$[-A]_{补}$或不加；
5) 按补码规则右移一位；
6) 重复第 4、第 5 步 n 次（n 为数值的位数）；
7) 最后一次重复第 4 步，但不必移位。

表 3.1 补码一位乘法规则

B_i（高位）	B_{i-1}（低位）	操　作
0	0	部分积右移一位
0	1	部分积+$[A]_{补}$，右移一位
1	0	部分积+$[-A]_{补}$，右移一位
1	1	部分积右移一位

Booth 算法的补码一位乘法又叫 Radix-1+Booth 算法。Radix-n+Booth 算法是目前微处理器乘法运算单元中应用比较广的一种实现方法。

2. 累加算法[4]

乘法是在加法的基础上发展出来的一种计算方法，首先观察一下十进制的乘法的过程:

$$
\begin{array}{rl}
\text{被乘数} & 1000_{10} \\
\text{乘数} \times & \underline{1001_{10}} \\
 & 1000 \\
 & 0000 \\
 & 0000 \\
 & \underline{1000} \\
\text{结果} & 1001000_{10}
\end{array}
$$

在这个例子中，把 1、0 作为十进制数据进行计算，结果只有两种选择，每一次乘法操作如下：

1) 如果乘数位是 1，将被乘数的拷贝放在恰当的位置。
2) 如果乘数位是 0，将恰当的位置放 0。

以上只是最原始的累加过程，在设计乘法运算单元时可根据实际需要做改

进。

3.1.2 乘法运算单元设计与实现

由于乘法运算单元需要同时支持无符号数和有符号数相乘，如果对于这两种乘法操作分别做一个乘法运算单元来设计，在运算速度上有优势，但会占用比较多的逻辑单元，增加实现成本。同时，对于 MiniSys SoC 系统，工作频率在 30MHz 左右，对乘法运算单元速度上要求并不是很高。所以采用无符号数和有符号数乘法共用一个乘法运算单元比较合适。

目前国内已经设计很多的基于 Booth 算法的乘法运算单元都只能支持有符号数的乘法，没有考虑对无符号数乘法的支持 [5]。如果要用有符号数乘法来支持 32 位无符号数乘法，那么必须增加一位符号位，数据宽度达到 33 位。一方面增加实现硬件部件，另一方面破坏数据对齐原则，加大控制电路复杂度。为了满足 MiniSys SoC 系统的要求，决定乘法运算单元设计采用改进型的累加算法，用原码来同时支持有符号数和无符号数的乘法。

1.算法改进部分

如果是原始累加算法，一个周期判断一次乘数位，这样两个 32 位的数相乘就要 32 个周期计数，再加上前后两个符号调整周期，一共 34 个周期完成一次乘法操作，显然效率是很低的。结合 FPGA 中高并行性的特点，我们对原始累加算法做了必要的改进，改进型累加算法可以在四个周期完成乘法操作，流程图 3.1 所示。

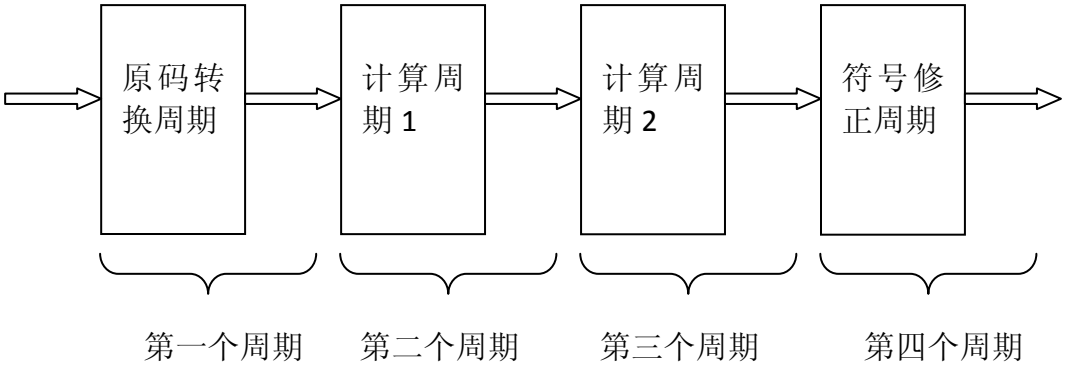

图 3.1 乘法运算单元进行乘法运算的流程图

采用 32 个 32 位的数据选择器同时对乘数的 32 位进行判断，且这种判断是一种提前的数据判断，因为它会在第二个周期之前就将数据准备好，而不用占用运算周期。进入第二周期，32 个数据会分成 6 组分别进入六个加法阵列同时计算出累加结果。第三周期，再用一个加法阵列完成最终计算。第四周期，有符号数乘法要对结果进行符号修正。

2.乘法运算单元的实现

如图 3.1 所示，乘法运算单元完成一次乘法操作为成四个周期：

1) 保存乘数、被乘数。对于无符号乘，直接保存；有符号乘是补码表示的，所以要将输入的补码转化为相应的原码。同时，乘法运算单元忙信号有

效，告诉后面的乘法指令等待。核心代码如下：

```
MULTWRITE<=1’b0;MULTBUSY<=1’b0;
if(ALUOP==5'b00110)//无符号乘法
 begin
   A<=ALUA;B<=ALUB;
   MULTBUSY<=1'b1;
   sign<=1'b0;
   state<=4'b0001;
 end
 else if(ALUOP==5'b01110)//有符号乘法
 begin
 A<=ALUA[31]?{~ALUA[30:0]+1}:ALUA;
 B<=ALUB[31]?{~ALUB[30:0]+1}:ALUB;
 minus<=ALUA[31]^ALUB[31];
 A[31]<=1'b0;B[31]<=1'b0;
 MULTBUSY<=1'b1;
 sign<=1'b1;
 state<=4'b0001;
 end
```

2) 计算周期 1。把乘数分成 6 段，每段与被乘数分别相乘，并保存下中间结果。核心代码如下：

```
result0<=temp[0]+temp[1]+temp[2]+temp[3]+temp[4]+temp[5];
result1<=temp[6]+temp[7]+temp[8]+temp[9]+temp[10]+temp[11];
result2<=temp[12]+temp[13]+temp[14]+temp[15]+temp[16]+temp[17];
result3<=temp[18]+temp[19]+temp[20]+temp[21]+temp[22]+temp[23];
result4<=temp[24]+temp[25]+temp[26]+temp[27]+temp[28]+temp[29];
result5<=temp[30]+temp[31];
```

用 6 个寄存器将中间结果保存下来。此为一个加法阵列器，比普通的累加速度上快 6 倍，如果还要提高速度，可考虑采用 Wallace 树的结构[5]（它也是 Booth 乘法算器最核心的结构）。

3) 计算周期 2。累加 6 个中间结果。核心代码如下：

```
{ALUHI,ALULO}<=result0+{result1,6'b0}+{result2,12'b0}+{result3,18'b0}+{result4,24'b0}+{result5,30'b
0};
```

4) 结果转化。对于第 3 周期的结果，如果是有符号数相乘，需要进行符号位的调整；否则不用调整。同时置乘法写有效。核心代码如下:

```
MULTWRITE<=1'b1; //写有效
if(sign)//如果是有符号乘法,进行符号修正
begin
      {ALUHI,ALULO}<=minus?(~{ALUHI[30:0],ALULO}+1):{ALUHI,ALULO};
      ALUHI[31]<=minus;
End
```

此外，乘法运算单元中提前的数据选择器核心代码如下：

```
assign temp[0]=B[0]?{5'b0,A}:37'b0;//乘数第 0 位产生的结果
assign temp[1]=B[1]?{4'b0,A,1'b0}:37'b0; //乘数第 2 位产生的结果
assign temp[2]=B[2]?{3'b0,A,2'b0}:37'b0; //乘数第 3 位产生的结果
assign temp[3]=B[3]?{2'b0,A,3'b0}:37'b0; //乘数第 4 位产生的结果
assign temp[4]=B[4]?{1'b0,A,4'b0}:37'b0; //乘数第 5 位产生的结果
assign temp[5]=B[5]?{A,5'b0}:37'b0; //乘数第 6 位产生的结果
.....
```

3.1.3 仿真测试

经过 Quartus Ⅱ 7.2(32-Bit)编译，该乘法运算单元共占用 2274 个 LE 实现，其主工作时钟 CLK 最高频率可达 37MHz。

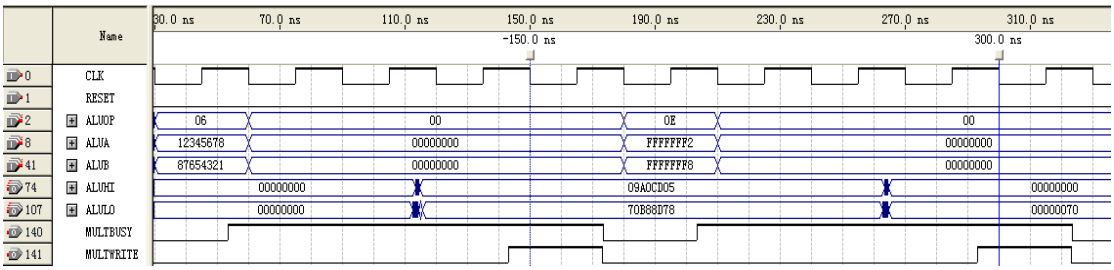

图 3.2 乘法运算单元仿真测试波形

乘法运算单元仿真测试波形如图 3.2 所示（测试时钟频率为 33MHz），对乘法运算单元进行了两条指令的测试，第一条指令是 12345678H 和 87654321H 无符号乘，得到结果 ALUHI=09A0CD05H，ALULO=70B88D78H，结果正确。第二条指令是 FFFFFFF2H 和 FFFFFFF8H 的有符号乘，即十进制中-14 和-8 相乘，结果应该是 112。见波形 300ns 处，得到结果 ALUHI=00000000H，ALULO=00000070H，换算成十进制即为 112。

3.2 除法运算单元的设计与实现

3.2.1 两种减法逻辑算法

1. 恢复余数法[6]

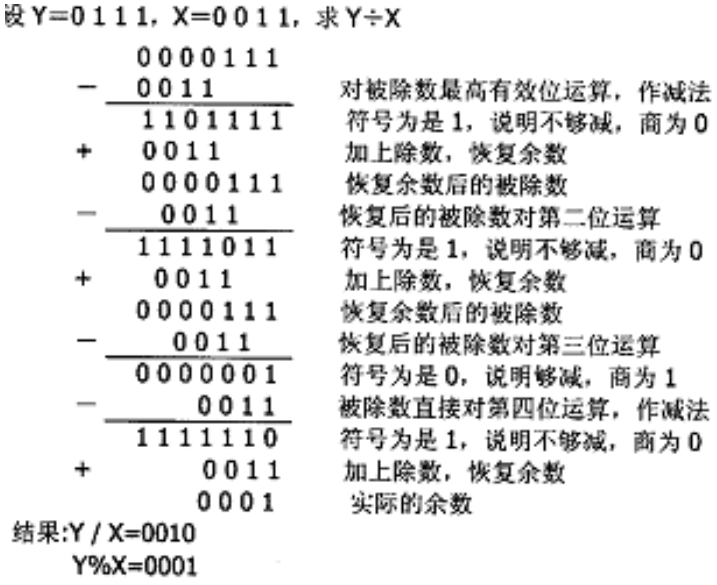

图 3.3 恢复余数法

如图 3.3 所示恢复余数法的过程。可以看出，运算过程就是不断的比较除数 X 和 $2R_i$（R_i 为上次的余数）的过程；若 $2R_i>X_i$ 则够减，商 1；若 $2R_i<X_i$ 则不够减，商 0。如何判断够减不够减，方法是先做减法（$2R_i$-X）。如果余数为正（符号位为 0），说明够减，商位 1；如果余数为负（符号位为 1），说明不够减，商位 0，这时应该将除数再加回去，恢复成原来的余数。可以看出，恢复余数法的

运算次数并不确定。

2. 加减交替法[6]

恢复余数法的缺点是，商 0 还是商 1，不能预先确定，帮所需运算步骤了不能预先确定，这会使实现电路复杂化，而加减交替法的运算步骤则是固定的。

原理如下：

用恢复余数法至 I 位时，余数 $R_i=2R_{i-1}-X$

若 $R_i<0$，则商 0，同时恢复余数，给余数为 R_i+X；然后再求下一位的商数，即求：$R_i=2(R_i+X)-X=2R_i+X$

可见，当 $R_i<0$ 时，可直接把 R_i 左移一位，再加 Y 得出，而不必恢复成余数 $2R_{i-1}$。

其运算过和可如图 3.4 所示。

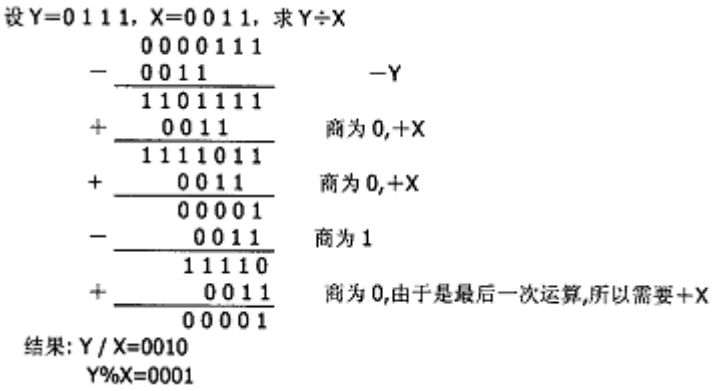

图 3.4 加减交替法运算过程

加减交替法的运算次数是确定的，需要注意的是：如果最后一次运算得到的商为 0，则必须要加上除数，以恢复成正确的值，这一步称作余数调整，是加减交替除法的必要步骤之一。

3.2.2 除法运算单元的设计和实现

除法运算单元的实现除了上一节介绍的减法逻辑，还有乘法逻辑。乘法逻辑运算速度快，但需要的硬件资源多，在芯片中所占面积太大，功率消耗很大，一般芯片设计不采用这种方法[6]。因此本文不做说明了。

对于两种减法逻辑，恢复余数法由于恢复余数法加减操作步数的不确定性，不利于硬件实现。加减交替法是比较理想的实现除法运算单元的方法[6]。

本文除法运算单元实现采用改进的加减交替法。用原码实现加无符号数的除法，两个 32 位数据相除要用 32 个周期，和上一节中实现的乘法类似，要做改进来提高运行速度。改进的算法用八个周期完成无符号数的计算。另外，需在前后分别加一个符号调整周期来支持有符号数的运算。所以完成一次除法操作需十个周期，其流程图如图 3.5 所示。

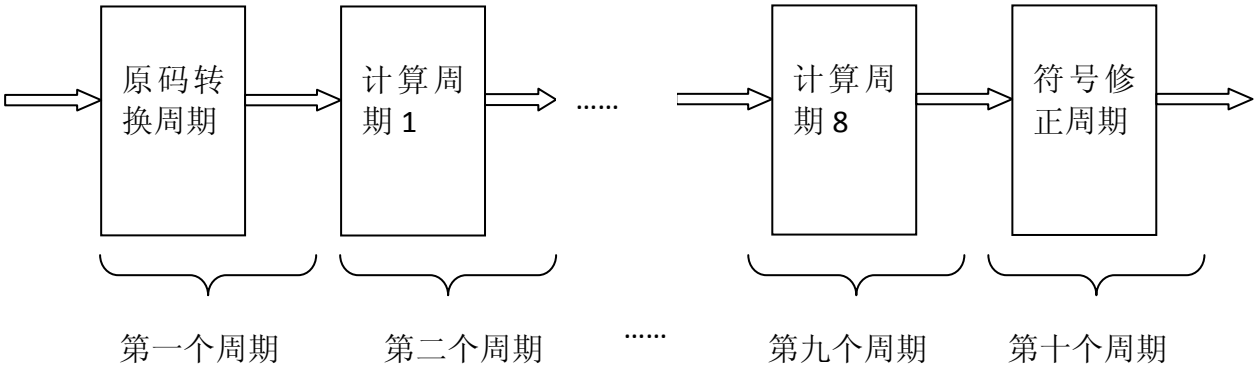

图 3.5 除法运算单元进行除法运算的流程图

只有有符号数除法是需要调商和余数的整符号位，调整规则见表 3.2。

表 3.2 有符号数除法符号位调整规则

被除数	除数	迭代前的预处理	迭代后的符号修正
正	正	无	无
正	负	除数取反	商取反
负	正	被除数取反	商和余数取反
负	负	被除数和除数取反	余数取反

采用加减交替法的 32 位的除法需要进行加法或减法累计一共 32 位操作，即对于被除数的每一位都要进行加法或减法操作其中之一。本除法运算单元在第 2 到 9 个周期共用 8 个周期完成这 32 个操作，即每个周期完成 4 位的判断操作。核心代码如下所示：

```
//第 2 个周期
4'b0100:
begin
   DIVBUSY<=1'b1;//置忙信号有效，使后面的乘除指令等待，直到本指令结束
   state<=4'b0101;
   temp1[3]={31'b0,A}-{B,31'b0};
   temp1[2]=temp1[3][62]?(temp1[3]+{1'b0,B,30'b0}):(temp1[3]-{1'b0,B,30'b0});
   temp1[1]=temp1[2][61]?(temp1[2]+{2'b0,B,29'b0}):(temp1[2]-{2'b0,B,29'b0});
   temp1[0]=temp1[1][60]?(temp1[1]+{3'b0,B,28'b0}):(temp1[1]-{3'b0,B,28'b0});
   ALULO[31:28]<=~{temp1[3][62],temp1[2][61],temp1[1][60],temp1[0][59]};
end

//第 3 个周期
4'b0101:
begin
   state<=3'b0110;
   temp1[3]=temp1[0][59]?(temp1[0]+{4'b0,B,27'b0}):(temp1[0]-{4'b0,B,27'b0});
   temp1[2]=temp1[3][58]?(temp1[3]+{5'b0,B,26'b0}):(temp1[3]-{5'b0,B,26'b0});
   temp1[1]=temp1[2][57]?(temp1[2]+{6'b0,B,25'b0}):(temp1[2]-{6'b0,B,25'b0});
   temp1[0]=temp1[1][56]?(temp1[1]+{7'b0,B,24'b0}):(temp1[1]-{7'b0,B,24'b0});
   ALULO[27:24]<=~{temp1[3][58],temp1[2][57],temp1[1][56],temp1[0][55]};
end

.....
//第 4~8 个周期代码与前面类似，在此省略。详细代码可见附录三

//第 9 个周期
4'b1011:
begin
   state<=4'b1100;
   temp1[3]=temp1[0][35]?(temp1[0]+{28'b0,B,3'b0}):(temp1[0]-{28'b0,B,3'b0});
   temp1[2]=temp1[3][34]?(temp1[3]+{29'b0,B,2'b0}):(temp1[3]-{29'b0,B,2'b0});
   temp1[1]=temp1[2][33]?(temp1[2]+{30'b0,B,1'b0}):(temp1[2]-{30'b0,B,1'b0});
```

```
   temp1[0]=temp1[1][32]?(temp1[1]+{31'b0,B}):(temp1[1]-{31'b0,B});
   ALULO[3:0]<=~{temp1[3][62],temp1[2][61],temp1[1][60],temp1[0][59]};
end
```

需要指出的是，原始的加减交替算法是要带符号位进行运算的，那么 32 位无符号应该用 33 位（多一位符号位来表示）。这显然对于 32 位 MIPS 是不合理的，而且还会影响有符号除法的运算。于是，本文又将算法做了小的优化：在无符号除时，如果除数小于 80000000H，即 32 位数据的最高位是 0，则可以看成是 31 位无符号数和 1 位符号位参与运算，结果是正确，无需修改；但是如果除数大于或等于 80000000H，就不能按加减交替算法运算了。此时有一种更简单的算法。因为如果除数大于或等于 80000000H，商只可能是 0 或 1 两种——被除数小于除数，商等于 0，余数等于被除数；否则，商等于 1，余数等于被除数减去除数。只需两个周期即可完成该类除法。其核心代码如下：

```
case(state)
4'b0000:
begin
   DIVWRITE<=1'b0;DIVBUSY<=1'b0;
   if(ALUOP==5'b00111)
   begin
       if(ALUB[31])//判断除数最高位是否是 1，
       begin
          A<=ALUA;B<=ALUB;
          DIVBUSY<=1'b1;state<=4'b1101;
       end
       else
       ......
   end
end
   ......
4'b1101://进行商和余数的计算
begin
   DIVWRITE<=1'b1;state<=4'b0;
   if(A<B)
   begin
       ALUHI=A;ALULO=32'b0;
   end else
   begin
      ALUHI=A-B;ALULO=32'b1;
   end
   ......
end
```

3.2.3 仿真测试

经过 Quartus Ⅱ 7.2(32-Bit)编译，该乘/除法运算器共占用 4837 个 LE 实现，其主工作时钟 CLK 最高频率可达 35MHz。

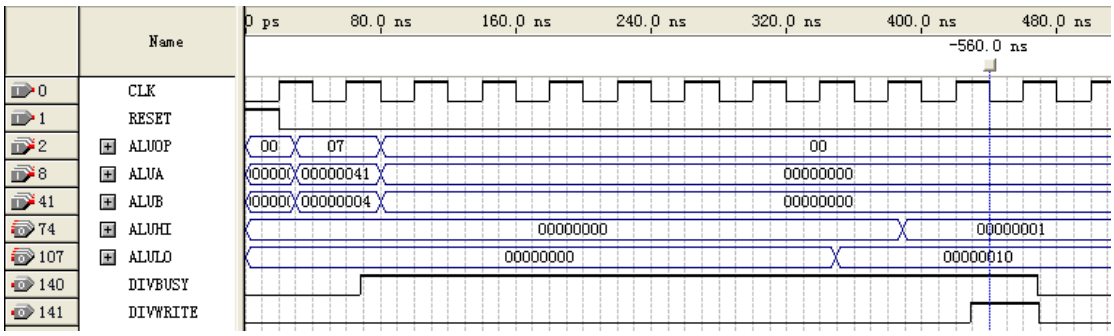

图 3.6 除法运算单元无符号除法仿真测试波形

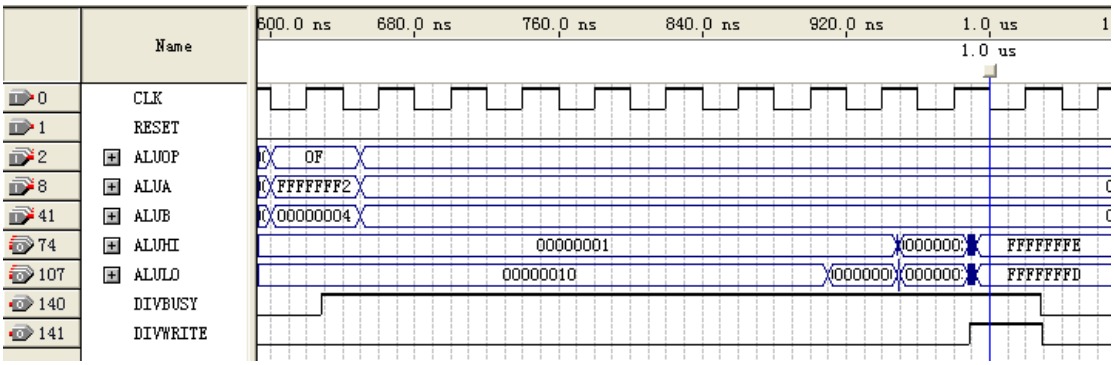

图 3.7 除法运算单元有符号除法仿真测试波形

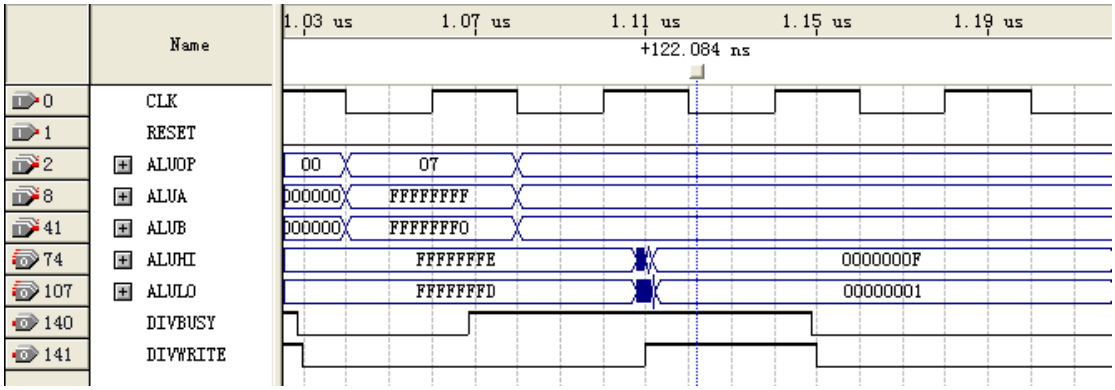

图 3.8 除法运算单元无符号除法改进部分仿真测试波形

图 3.6 是除法运算单元无符号数除法仿真测试波形（测试时钟频率 25MHz），进行无符号除法 41H÷4H，所得结果商是 10H，余数是 1H，计算正确。

图 3.7 是除法运算单元有符号数除法仿真测试波形（测试时钟频率 25MHz），进行有符号除法 FFFFFFF2H÷4H，即十进制中的-15÷4 所得结果商是 FFFFFFFDH，余数是 FFFFFFFEH，计算正确。

图 3.8 是除法运算单元无符号数除法改进部分的仿真测试波形，可以看出，只用了两个周期就完成了两个数的无符数除法 FFFFFFFFH÷FFFFFFFDH，所得结果商是 1H,余数是 FH，计算正确。

将已经设计完成的乘法运算单元和除法运算单元合成一个乘/除法运算单元，经 Quartus 编译后，占用 5693 个 LE，最高工作频率 35MHz。在实现面积和速度上都能很好满足 MiniSys SoC 系统的要求。

第 4 章 I/O 接口的设计与实现

4.1 SDRAM 控制器的设计与实现

4.1.1 SDRAM 简介

在很多通信芯片及系统的开发中，常要用到存储容量大、读写速度快的存储器。各种随机存储器中，SDRAM 以其低价格、小体积、高速度、大容量的优势，成为比较理想的存储器。但与 SRAM 比较，SDRAM 控制逻辑复杂，接口方式与普通存储器差异很大[7]。为此，要专门设计专用的 SDRAM 控制器，来方便用户的使用。

SDRAM 器件的管脚分为控制信号、地址和数据三类。通常一个 SDRAM 中包含几个 BANK,每个 BANK 的存储单元是按行和列寻址的。[8]SDRAM 的工作原理有以下几个特点：

1. SDRAM 在上电后 100~200us 后，必须由一个初始化过程来配置 SDRAM 的工作模式器。初始化过程由启动指令流完成：首先由一个 Precharge all bank 指令完成对所有 BANK 的预充，然后是两个或多个 Auto Refresh 指令，最后在模式配置指令下完成 SDRAM 内部模式设置寄存器的配置。模式寄存器的值控制着 SDRAM 的工作方式，详细描述如表 4.1 所示。

表 4.1 SDRAM 内部模式控制寄存器

Address	**BA**	**A10/AP**	**A9**	**A8**	**A7**	**A6**	**A5**	**A4**	**A3**	**A2**	**A1**	**A0**
Function	RFU	RFU	W.B.L	TM		CAS Latency			BT	Burst Length		

Test Mode			**CAS Latency**				**Burst Type**		**Burst Length**				
A8	**A7**	**Type**	**A6**	**A5**	**A4**	**Latency**	**A3**	**Type**	**A2**	**A1**	**A0**	**BT=0**	**BT=1**
0	0	Mode Register Set	0	0	0	Reserved	0	Sequential	0	0	0	1	1
0	1	Reserved	0	0	1	-	1	Interleave	0	0	1	2	2
1	0	Reserved	0	1	0	2			0	1	0	4	4
1	1	Reserved	0	1	1	3			0	1	1	8	8
Write Burst Length			1	0	0	Reserved			1	0	0	Reserved	Reserved
A9	**Length**		1	0	1	Reserved			1	0	1	Reserved	Reserved
0	Burst		1	1	0	Reserved			1	1	0	Reserved	Reserved
1	Single Bit		1	1	1	Reserved			1	1	1	Full Page	Reserved

2. SDRAM 行列地址采用复用的方式传送，减少了地址总线的宽度。这样一来，SDRAM 在每次读定操作时，行列地址要锁存。具体地，由 ACTIVE 命令激活要读写 BANK，并锁存行地址，然后在读写指令有效时锁存列地址。
3. SDRAM 的操作是通过 CS#、RAS#、CAS#、WE#、AP 信号的组合指令字完成的。由于特殊的存储结构，SDRAM 操作指令字比较多，不像 DRAM 一样只有简单的读写。其主要的指令字见表 4.2。

表 4.2 SDRAM 控制器控制命令

命令	CS#	RAS#	CAS#	WE#	DQM	ADDR
空操作(NOP)	0	1	1	1	X	X
激活(ACTIVE)	0	0	1	1	X	组/行
带预充的读(READA)	0	1	0	1	0	组/列
带预充的写（WRITEA）	0	1	0	0	0	组/列
刷新(REFRESH)	0	0	0	1	X	X
预充(PRECHARGE)	0	0	1	0	X	代码
设置模式寄存器(LOADMODE)	0	0	0	0	X	配置数据

4. 需要定时对刷新。刷新间隔时间依不同的 SDRAM 而定。刷新方式可有分散式刷新和集中式刷新两种选择方式。

4.1.2 SDRAM 控制器的设计与实现

MiniSys SoC 的 SDRAM 采用两片三星 1M×16bit 的 SDRAM“K4S161622”并联构成 32 位的 SDRAM 存储器。

SDRAM 控制器总体设计框图如图 4.1 所示。

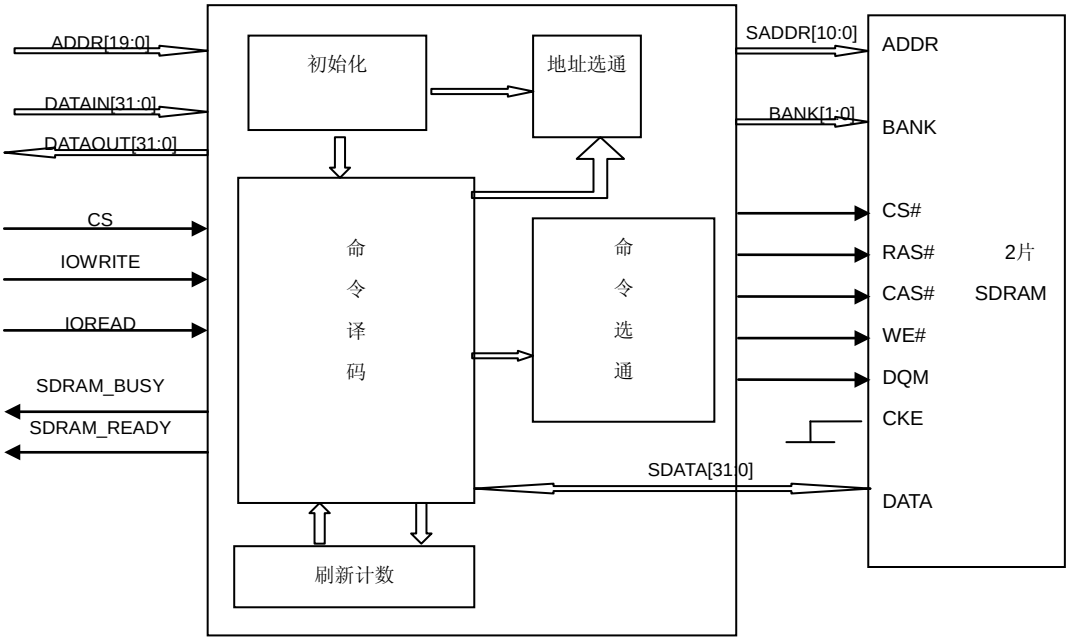

图 4.1 SDRAM 控制器总体设计框图

1. 初始化模块

完成 SDRAM 的初始化工作。为方便灵活的应用，本设计中将 SDRAM 模式寄存器设置为 0x0020（突发长度为 1，CAS Latency 为 2）。完成初始化后，该模

块向命令译码模块发出 InitHold 无效信号，把命令和地址选通的控制权交给命令译码模块。只有完成初始化过程，SDRAM 才可以正常工作。其核心代码如下：

```
case(state)//初始化状态机

wait100us://等待 100us
begin
   if(InitTime)
   begin
      PreEnable<=1'b1;
      InitCMD<=precharge;
      state<=initPre;
   end
end

initPre://预充指令，需要 2 个周期完成
begin
   if(PreDone)
   begin
      InitCMD<=refresh;
      PreEnable<=1'b0;
      state<=initAuto1;
   end
   else
   begin
      InitCMD<=nop;
      state<=initPre;
   end
end

initAuto1://第一次刷新，需要 6 个周期完成
begin
   if(RefDone)
   begin
      RefEnable<=1'b0;
      InitCMD<=refresh;
      state<=initAuto2;
   end
   else
   begin
      RefEnable<=1'b1;
      InitCMD<=nop;
      state<=initAuto1;
   end
end

initAuto2://第二次刷新
begin
   if(RefDone)
      begin
      RefEnable<=1'b0;
      InitCMD<=loadmode;
      state<=initLoad;
   end
   else
   begin
   RefEnable<=1'b1;
   InitCMD<=nop;
   state<=initAuto2;
   end
end

initLoad://设置模式寄存器
begin
   InitCMD<=nop;
   InitHold<=1'b0;
   state<=initEnd;
end

initEnd:
begin
   state<=initEnd;
end

default:    state<=initEnd;
endcase
```

2. 刷新模块

SDRAM 要求要 32ms 内对所有的 2048 个单元刷新一次，本设计采用分散刷新，也是说每 15.6us 至少要刷新一个单元。由于设定 SDRAM 工作时钟周期为 20ns，所以刷新计数达到 780，就需要对 SDRAM 发出刷新命令，刷新计数模块和命令译码模块之间能过 RefreshHold 和 RefreshDone 来进行通信。

当计数到 780 刷新计数模块向命令译码模块发出 RefreshHold 有效，后者据此向发出刷新命令，同时使 RefreshDone 信号无效，刷新计数清零，信止刷新模块计数。并且在刷新操作同时置 SDRAM_BUSY 有效，直至完成刷新操作将 SDRAM_BUSY 至无效，才可以接收新的命令。同时令 RefreshDone 信号有效，刷新模块开始重新计数。核心代码如下：

```
always@(posedge CLK or posedge RESET)
   begin
      if(RESET)//复位
      begin
//注意实际的计数周期是 750 个。这样//设置原因将在命令译码模块中说明。
```

```
        counter<=10'h2ee;
        RefreshHold<=0;
    end
    else if(RefreshDone)
    begin
        RefreshHold<=0;
                counter<=10'h2ee;
    end
    else if(counter==0)
        RefreshHold<=1;
    else
        counter<=counter-10'b1;
end
```

3. 命令译码模块

此模块是整个 SDRAM 控制器的核心模块，主要功能是锁存 MiniSys CPU 发出的地址、数据线上的数据，以及对 MiniSys CPU 发出的命令进行译码，并将译码后的命令发给 SDRAM 进行相应的操作，同时完成数据传送，具本过和如图 4.2 所示。在控制器中采用了状态机的设计方式，图 4.2 中的 idle、auto_refresh、active、write、read、auto_pre 均为状态机内部状态。

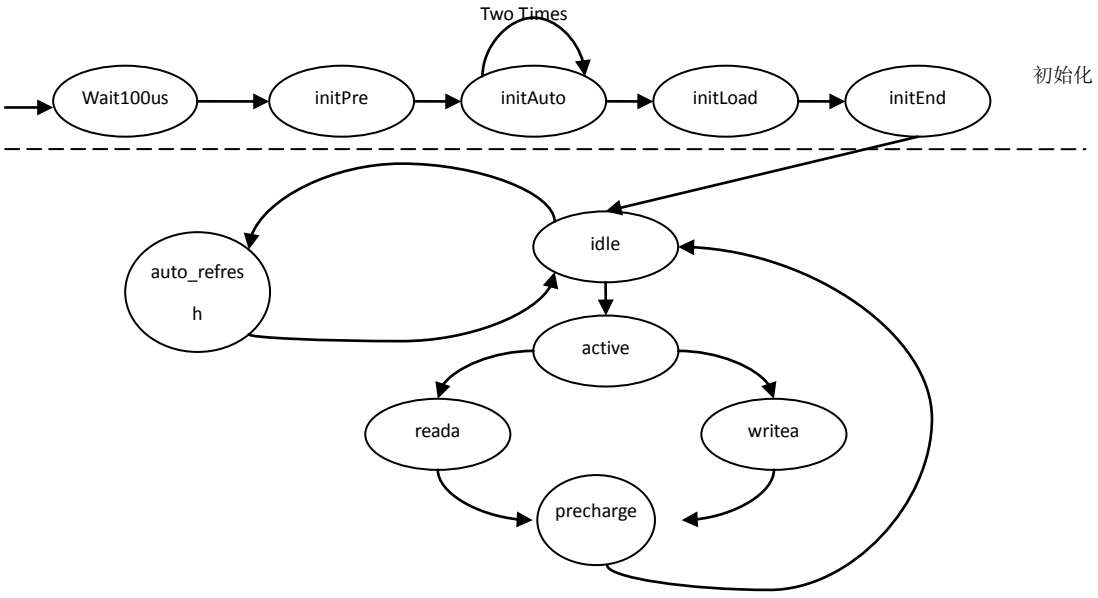

图 4.2 SDRAM 控制器状态机

初始化模块完成初始化后，向命令译码模块发出 InitHold 信号无效，命令译码模块检测到 InitHold 信号无效后进入 idle 状态，从而跟据 MiniSys CPU 发出的命令进行译码执行相应的操作。在图 4.2 中 idle 状态中首先判断 RefreshHold 信号是否有效，如果有效，令 SDRAM_BUSY 信号无效，阻止 MiniSys CPU 发出新的读写操作命令，控制器进入到刷新状态执行 SDRAM 刷新操作。刷新操作完成后状态机回到 idle 状态进行下一次操作判断。如果在读写过程中 RefreshHold 有效，此时状态机在执行读写操作，无法立即响应刷新请求。此时只能等待读写

操作完成后状态机回到 idle 状态后再响应刷新请求。为了使刷新命令不打断读写操作，因此计数模块计数到 750 就发出刷新请求信号。

如果 RefreshHold 信号无效即不需要刷新操作，则判断 IOWRITE 或者 IOREAD 信号是否有效，如果无效下一周期则继续回到 idle 状态进行重新判断。如果 IOWRITE 或者 IOREAD 有效，进入 active 状态，锁存 MiniSys CPU 发出的地址，发出激活命令，并判断是 IOWRITE 还是 IOREAD，然后分别相应进入到 write 或者 read 状态执行读写操作，同时令 SDRAM_BUSY 有效。在 read 状态中发送 reada 命令，向 SDRAM 地址线上发送列地址，然后等两个周期从 SDRAM 数据线上读回数据并送往 MiniSys CPU 完成读操作。在 write 状态中发送 writea 命令，向 SDRAM 地址线上发送列地址，完成定操作。读或写完成后都需要对所激活的行和组进行预充。状态机重新回到 idle 状态进行一下次操作判断。其核心代码如下：

```
    if(InitHold)//初始化
        state<=init;

      else
if(RefreshHold&&!RefreshDone&&state==idle)//是否有刷新操作
        begin
          SDRAM_BUSY<=1'b1;WCMD=refresh;
       RefEnable<=1'b1;state<=auto_refresh;
        End
    //主状态机
        else
        begin
          case(state)
          init:
          begin
            if(!InitHold)
            begin
              WCMD<=nop;state<=idle;
              SDRAM_BUSY<=1;SDRAM_READY<=0;
            end
          end
    //空闲状态
          idle:
          begin
            WCMD<=nop;
            SDRAM_READY<=0;
            SDRAM_BUSY<=0;
            RefreshDone<=0;
            if(op!=sign)
            begin
                    state<=start;
              op<=sign;
                    WaitEnable<=1'b1;
              state<=start;
              SDRAM_BUSY<=1;
            end
          end
    //开始状态
        start:
          begin
              if(WaitDone)
              begin
                    WaitEnable<=1'b0;

SADDR<=RW?ReadA[18:8]:WriteA[18:8];
                    ActEnable<=1'b1;
                    WCMD<=act;
                    state<=active;
              end
        end
//自动刷新状态，6 个周期完成
    auto_refresh:
    begin
      if(RefDone)
          begin
                RefreshDone<=1'b1;
                RefEnable<=1'b0;
                state<=idle;
          end


          else
                WCMD<=nop;
    End
//激活状态
    active:
    begin
      if(ActDone)
          begin
                if(RW)
                begin
                      DIR<=1'b0;
                      ReadEnable<=1'b1;
                      ActEnable<=1'b0;
          SADDR<={3'b100,ReadA[7:0]};
                      WCMD<=reada;
                      state<=read;
                end
                else
                begin
                      DIR<=1'b1;
                      ActEnable<=1’b0;
                      WCMD<=writea;
          SADDR<={3'b100,WriteA[7:0]};
                      state<=write;
                end
          end
          else
          begin
                WCMD<=nop;
                state<=active;
          end
      end
```

```
//读状态，3 周期完成
      read:
      begin
        if(ReadDone)
        begin
                SDRAM_READY<=1'b1;
                ReadEnable<=1'b0;
                PreEnable<=1'b1;
          state<=auto_pre;
        end
             else
                WCMD<=nop;
      End
//写状态，1 周期完成
      write:
      begin
        WCMD<=nop;
                PreEnable<=1'b1;
          state<=auto_pre;
        end
//自动预充状态，3 周期，无需发预充指令
      auto_pre:
      begin
             if(PreDone)
             begin
                PreEnable<=1'b0;
                SDRAM_BUSY<=1'b0;
                state<=idle;
             end
             SDRAM_READY<=1'b0;
        DIR<=0;
      end
      endcase
    end
```

4. SDRAM 命令选通模块

SDRAM 初始化时和正常工作时给 SDRAM 命令线上均发出相应的命令。初始化时，此模块命令线上的命令也由命令译码模块跟据初始化模块提供的 InitHold 信号及 CMD 信号提供相应的命令。

5. SDRAM 地址选通模块

SDRAM 初始化时需要通过 SDRAM 地址线(A0～A10)来配置模式寄存器。SDRAM 正常工作状态时，地址线是是读写数据的相应地址。

4.1.3 仿真测试

经过 Quartus Ⅱ 7.2(32-Bit)编译，SDRAM 控制器占用 210 个 LE，其有两个工作时钟。与 CPU 接口部分时钟 CLK_CPU 最高频率可达 100MHz。另一个是 SDRAM 工作时钟 CLK，最高可达 90MHz。需要注意的是，本设计中 SDRAM 正常工作频率应为 50MHz。下是仿真波形图（CLK=50MHz，CLK_CPU=20MHz），图 4.3 为 SDRAM 初始化时序图；图 4.4 为 SDRAM 写时序图；图 4.5 为 SDRAM 读时序图。

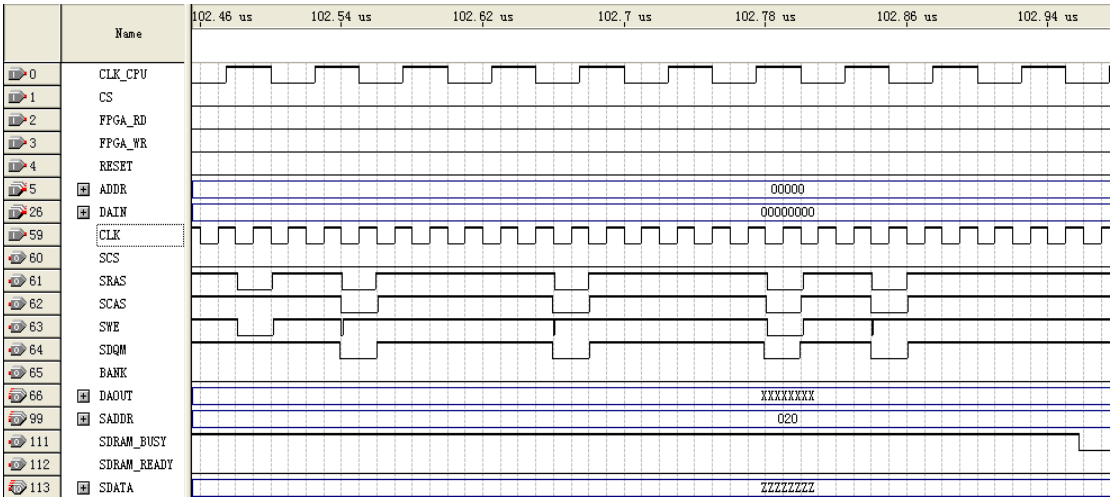

图 4.3 SDRAM 初始化时序图

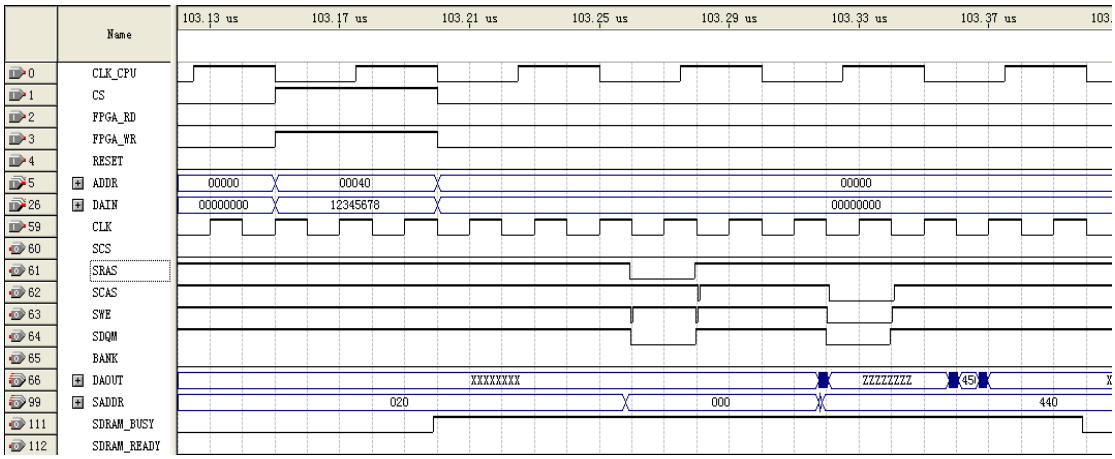

图 4.4 SDRAM 写时序图

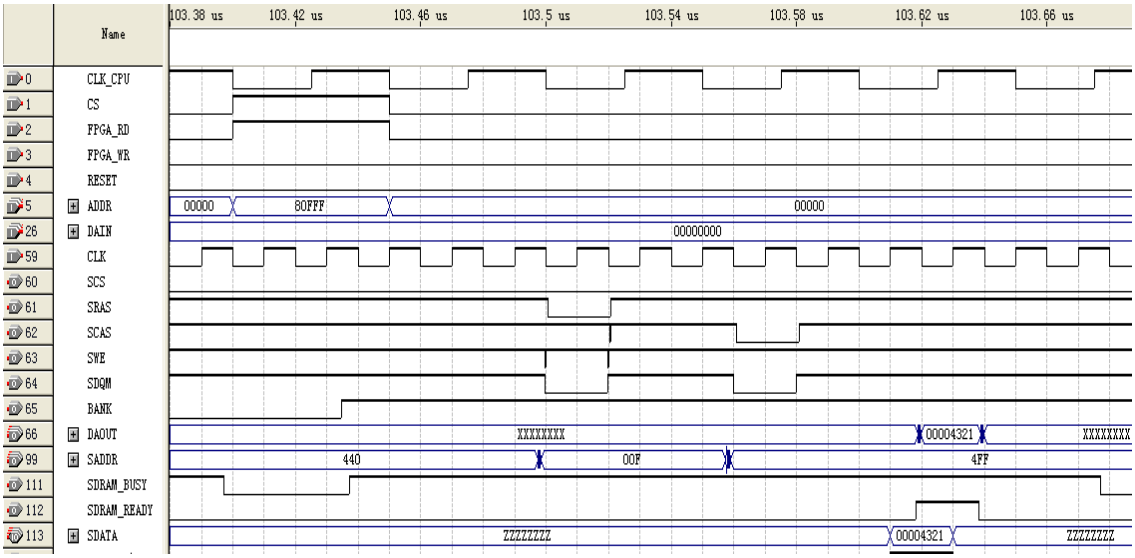

图 4.5 SDRAM 读时序图

图 4.3 表明了初始化过程。图 4.4 中进行的操作是将数据 12345678H 写入 SDRAM 地址 0x0040 中，整个过程符合 K4S161622 时序规则[9]。图 4.5 是读操作时，数据读取成功时 SDRAM 控制器会发给 CPU 一个周期的 SDRAM_READY 有效信号。由于 SDRAM 不具有数据锁存功能，这就要 CPU 在 SDRAM_READY 有效时将读出数据保存起来。

4.2 I^2C 总线控制器的设计与实现

4.2.1 I^2C 总线简介

I^2C（Inter-Integrated Ciruit）总线是一种由 PHILIPS 公司开发的两线式串行总线，用于连接微控制器用其外围设备。I^2C 总线产生于 80 年代，最初为音频和视频设备开发。

1. I^2C 总路线特点

I^2C 总线最主要的优点是其简单性和有效性。由于接口直接在组件之上，因此 I^2C 总线占用的空间非常小，减少了电路板的空间和芯片管脚的数量，降低了互联成本。总线的长度可高达 25 英尺，并且能够以 10Kbps 的最大传输速率支持 40 个组件。I^2C 总线的另一个优点是，它支持多主控(multimastering)， 其中任何能够进行发送和接收的设备都可以成为主总线。一个主控能够控制信号的传输和时钟频率。当然，在任何时间点上只能有一个主控。

2. 总线的构成用信号类型

I^2C 总线是由数据线 SDA 和时钟 SCL 构成的串行总线，可发送和接收数据。在 CPU 与被控 IC 之间、IC 与 IC 之间进行双向传送，最高传送速率 100kbps。各种被控制电路均并联在这条总线上，但就像电话机一样只有拨通各自的号码才能工作，所以每个电路和模块都有唯一的地址，在信息的传输过程中，I^2C 总线上并接的每一模块电路既是主控器（或被控器），又是发送器（或接收器），这取决于它所要完成的功能。CPU 发出的控制信号分为地址码和控制量两部分，地址码用来选址，即接通需要控制的电路，确定控制的种类；控制量决定该调整的类别（如对比度、亮度等）及需要调整的量。这样，各控制电路虽然挂在同一条总线上，却彼此独立，互不相关。

I^2C 总线在传送数据过程中共有三种类型信号， 它们分别是：开始信号、结束信号和应答信号。

- 开始信号：SCL 为高电平时，SDA 由高电平向低电平跳变，开始传送数据。
- 结束信号：SCL 为低电平时，SDA 由低电平向高电平跳变，结束传送数据。
- 应答信号：接收数据的 IC 在接收到 8bit 数据后，向发送数据的 IC 发出特定的低电平脉冲，表示已收到数据。CPU 向受控单元发出一个信号后，等待受控单元发出一个应答信号，CPU 接收到应答信号后，根据实际情况作出是否继续传递信号的判断。若未收到应答信号，由判断为受控单元出现故障。

3. 总路线基本操作

I^2C 规程运用主/从双向通讯。器件发送数据到总线上，则定义为发送器，器件接收数据则定义为接收器。主器件和从器件都可以工作于接收和发送状态。总线必须由主器件（通常为微控制器）控制，主器件产生串行时钟（SCL）控制总线的传输方向，并产生起始和停止条件。SDA 线上的数据状态仅在 SCL 为低电平的期间才能改变，SCL 为高电平的期间，SDA 状态的改变被用来表示起始和停止条件（参见图 4.6）。

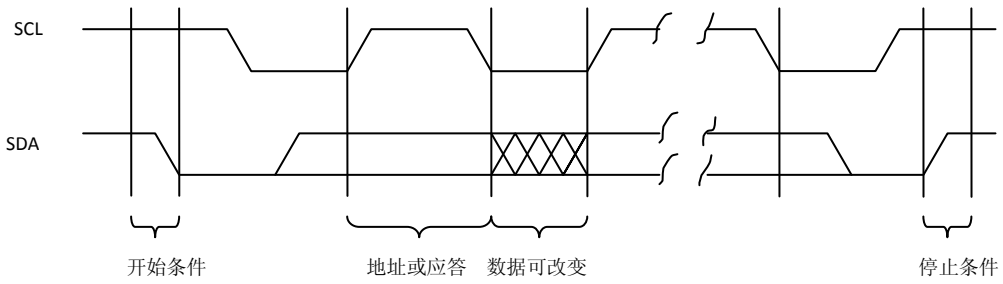

图 4.6 I^2C 总线传输规则

控制字节

在起始条件之后，必须是器件的控制字节，其中高四位为器件类型识别符，接着三位为片选，最后一位为读写位，当为 1 时为读操作，为 0 时为写操作。如图 4.7 所示。

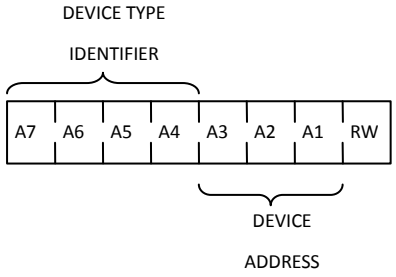

图 4.7 I^2C 总线控制字节格式

写操作

读操作有三种基本操作：当前地址读、随机读和顺序读。图 4.8 给出的是顺序读的时序图。应当注意的是：最后一个读操作的第 9 个时钟周期不是“不关心”。为了结束读操作，主机必须在第 9 个周期间发出停止条件或者在第 9 个时钟周期内保持 SDA 为高电平、然后发出停止条件。

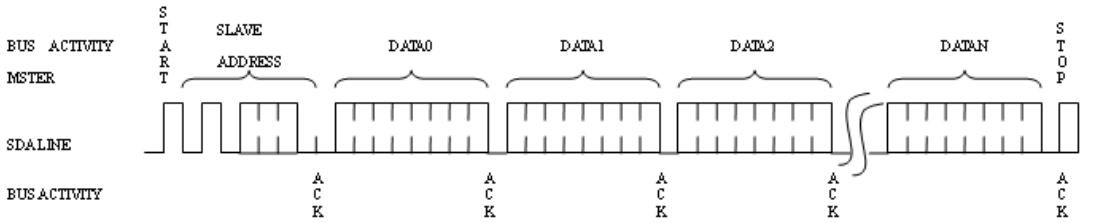

图 4.8 I^2C 总线顺序读时序图

读操作

关于页面写的地址、应答和数据传送的时序参见图 4.9。

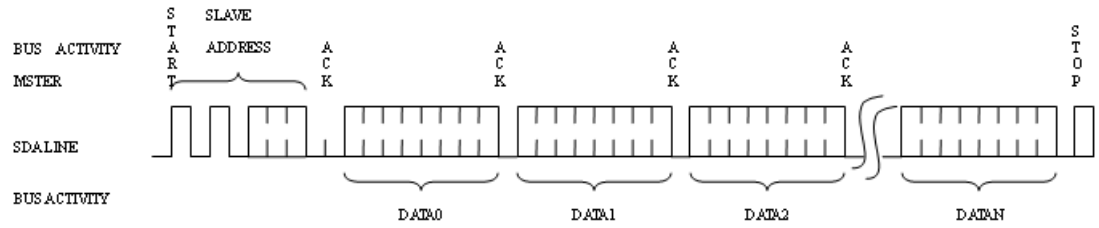

图 4.9 I^2C 总线页面写时序图

4.2.2 I^2C 总线控制器设计与实现

功能：实现 MiniSys CPU 与 I^2C 总线上的外设的双向通信。

输入输出安排：

- CLK(I)　　I^2C 工作时钟 50MHz
- CLK_CPU(I)　CPU 工作时钟
- RESET(I)　　复位信号，高电平有效
- CS(I)　　片选信号
- IOWRITE(I)　寄存器写信号
- IOREAD　(I)　　寄存器读信号
- ADDR3～0(I)　　寄存器地址
- DATAIN31～0(I)　　输入数据线
- DATAOUT31～0(O)　　输出数据线
- SCL_PIN(O)　　I^2C 总线输出时钟
- SDA_PIN(IO)　　I^2C 总线数据线

寄存器安排:

- ff50h　输出/输入数据锁存器
- ff54h　输出设备的地址锁存器(只用 8 位，高七位表示从设备地址，最低位表示读/写。1 表示向从设备读数据，0 表示向从设备写数据)(只写)
- ff54h　状态寄存器(只读，只用 4 位)

31～4	3	2	1	0
未用	是否出错	总线忙	READREDY	WRITEREADY

WRITEREADY:1 表示输出完成，读后自动清零；
READREADY:1 表示输入完成，读后自动清零；
总线忙：1 表示总线正被占用；0 表示总线空闲。
是否出错：1 表示传输中握手示成功；0 表示传输成功。

I^2C 总线控制器使用方法：

标准 I^2C 总线传输数据过程中是以 8 位为一个单位的，为了配合 32 位的 MiniSys CPU，这里设计的 I^2C 控制器每传输一次要传 4 个字节，即 32 位数据。而且为了 CPU 数据的安全性，是不允许 I^2C 总线上的其它设备主动读取 CPU 内部数据的，所以该 I^2C 控制器只能是 CPU 做主设备，其它总线上设备是从设备。

如果要向总线上写 32 位数据，在总线空闲的情况下，将要写数据写入 ff60h 寄存器中，再按相应格式把设备地址写入 ff64h 寄存器（见上面寄存器安排说明，此时最低位肯定是 0）。写完后不断读状态寄存器 ff64h，直到 WRITEREADY 为 1。若出错标志是 0，则表示写 32 位数据已写完；否则写入失败。

如果要从某从设备读取数据，在总线空闲的情况下，将从设备地址写入 ff64h 寄存器中，不断读状态寄存器 ff64h，直到 READREADY 为 1。若出错标志是 0，表示读失败成功，数据已存在 ff60h 中；否则，读取失败。

实现方案及关键代码说明：

1. 时钟分频模块

由于工作时钟是 50MHz,故采用 512 分频使传输时速度在 10kbit/s 以下。

2. 指令译码接口模块

存储 MiniSys CPU 传来的指令，并做出相应的译码给状态控制模块。

3. 状态控制模块

状态机模块，是整个控制器正常工作最重要的模块。SCL 和 SDA 也是由该模块控制的。状态机状态如图 4.10 所示。

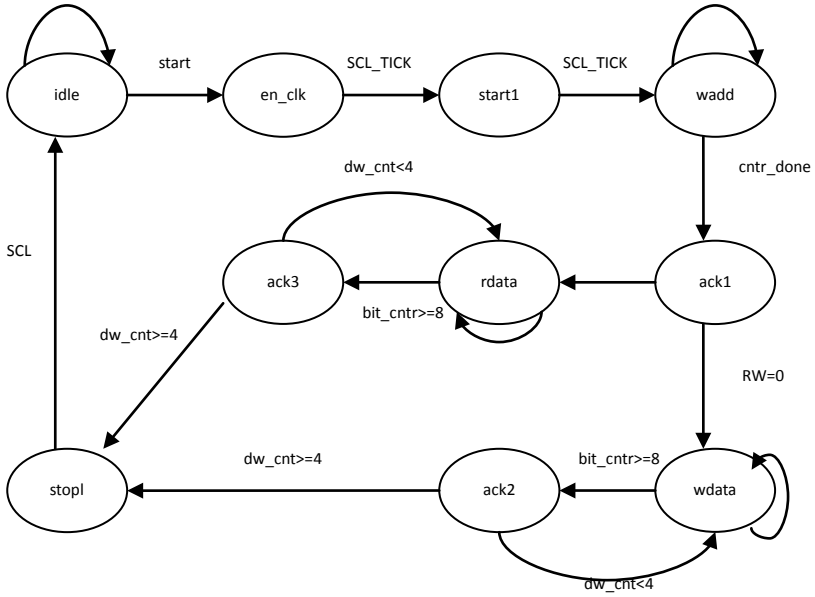

图 4.10 I^2C 总线控制器状态机

状态机核心实现代码如下：

```
case(state)
        idle : if (START)
           state <=en_clk;
        en_clk : if (SCL_TICK)
           state <=start1;
        start1 : if (SCL_TICK)
           state <=wadd;
        wadd : if (cntr_done && SCL_TICK)
           state <=ack1;
        ack1 : if (SCL_TICK && SCL)
           if(ADDRESS[0])
              state <=rdata;
           else
              state<=wdata;
        wdata : if (cntr_done && SCL_TICK)
           state <= ack2;
        ack2 : if (SCL_TICK && SCL)
                  if(dw_done)
                     state <= stopl;
                  else
                     state<=wdata;
               rdata : if (cntr_done && SCL_TICK)
                  state <= ack3;
               ack3 : if (SCL_TICK&&SCL)
                  if(dw_done)
                     state <= stopl;
                  else
                     state<=rdata;
               stopl : if (SCL_TICK && SCL)
                  state <= idle;
               default :
                  state <=    idle;
            endcase
```

SCL 信号产生代码，总路线通信的时钟：

```
//SCL 信号产生，SCL_TICK 是通信时 CLK 的 512 分频
always @(posedge CLK or posedge RESET)
  if (RESET)
     SCL <=1'b1;
  else if(scl_en && SCL_TICK)
     SCL <=~SCL;
```

SDA 信号产生代码，主要是写 32 位数据时要用：

```
//SDA 信号产生
always @(posedge CLK or posedge RESET)
     if (RESET)
          SDA<= 1'b1;
     else if ((state == start1)||(state == stopl))//读
          SDA <= 1'b0;
     else if(state == wadd)                    //写地址，8 位
```

```
            SDA <=ADDRESS[bit_cntr];
        else if(state==wdata)          //写数据，32 位
            SDA<=WDATA[{dw_cnt,bit_cntr}];
        else
            SDA <=1'b1;
```

SDA 信号是在定 32 位数据时产生，首先在 start1 状态将 SDA 变为 0，输出开始信号。然后在 wadd 状态写 7 位地址及 1 位读/写标志。然后是 wdata 状态写 32 位数据，每写 1 次 8 位数据就要进入 ack 状态握手一次，写 4 个 8 位就完成。在这巧妙的用两个计数器 dw_cnt 与 bit_cntr 组合，完成 32 位输出。大大简化了状态机的状态。32 位数据从最高位向最低位依次输出。

读 32 位数据代码：

```
always @(posedge CLK or posedge RESET)
        if (RESET)
            RDATA <=8'b0;
         else if ((state == rdata) && SCL)//SCL 高电平是读入数据
            RDATA[{dw_cnt,bit_cntr}] <=SDA_PIN;
```

读 32 位数据时也要分 4 个 8 位数据传输，同理也要握手 4 次。工作过程与写 32 位数据类似，只不过传输方向变了。

4. 传输缓冲模块

由于 SDA_PIN 与 SCL_PIN 都是双向总线，与单向输出线不同，它还要做缓冲处理，否则会引起总线和双向传输的电位竞争。

```
assign SCL_PIN=SCL?1'bz:1'b0;
assign SDA_PIN=SDA?1'bz:1'b0;
```

4.2.3 仿真测试

经过 Quartus Ⅱ 7.2(32-Bit)编译，该 I²C 控制器用 186 个 LE 实现，与 CPU 接口的时钟最高频率可达 90MHz，I²C 工作时钟 CLK 最高频率可达 188MHz，设计正常工作频为 50MHz。

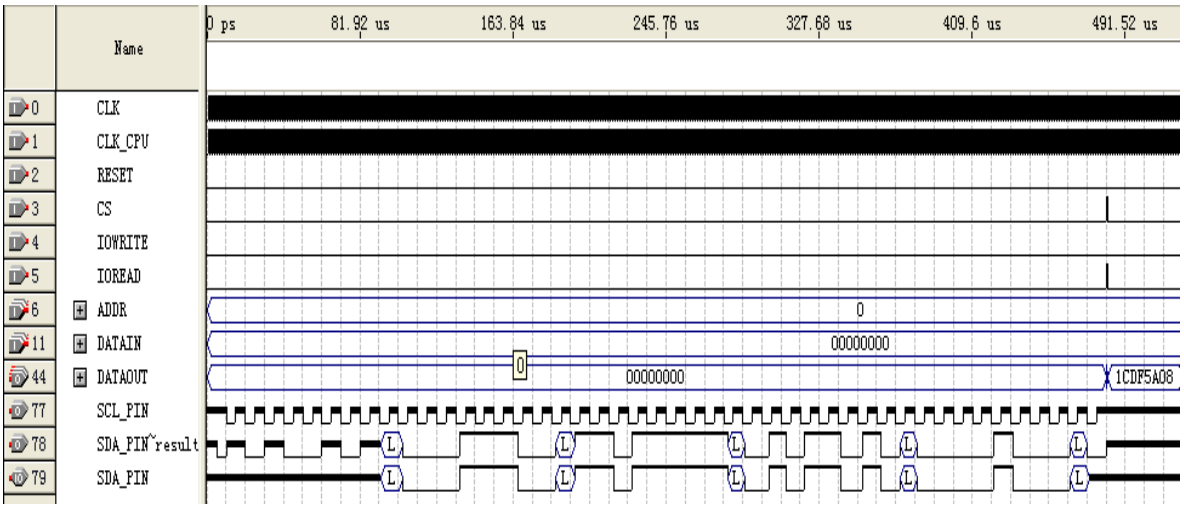

图 4.11 读数据时序图

如图 4.11 所示，I²C 总线控制器从地址为 0x52 的从设备中读取一个 32 位数据，根据读取波形可知读取的数据应为 1CDF5108H，读完后 CPU 向 I²C 总线发出一条取数据指令，信号 DATAOUT 输出是 1CDF5108H，接收正确。

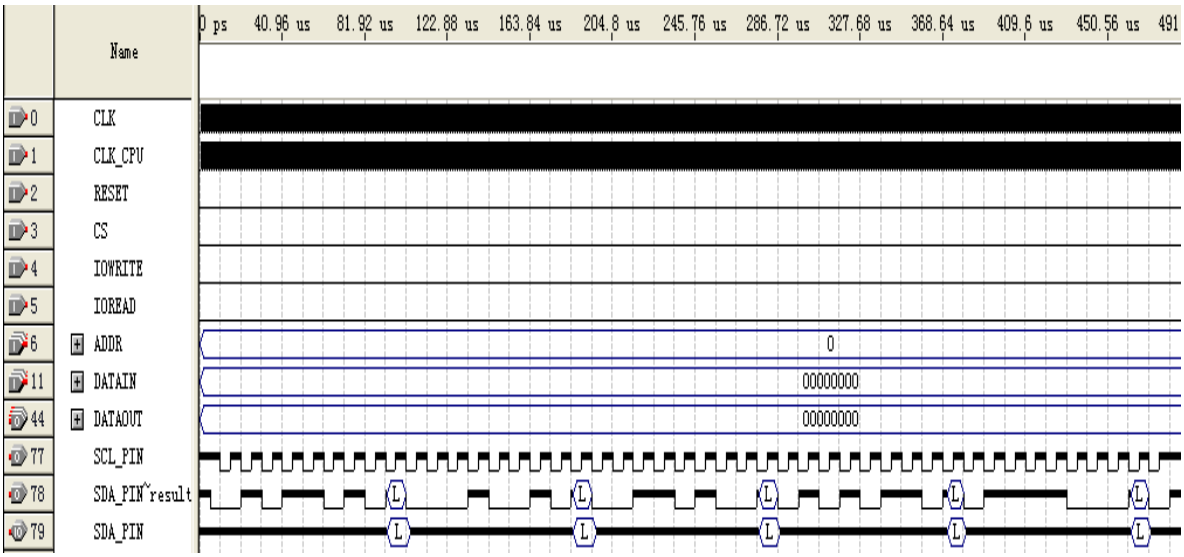

图 4.12 写数据时序图

图 4.12 是 I^2C 总线向地址为 0x2d 的从设备发送 12345678H，从 SDA_PIN 输出波形可以看出，输出正确。

4.3 LCD 控制器的设计与实现

4.3.1 LCD 简介

随着电子产业的速度发展，目前市场上的 LCD 品牌和种类都十分繁多，LCD 也越来越多的走进的生活、生产中。在不同的领域对于 LCD 特性的要求是不同的。如对于 LCD 电视，人们可能会注重追求其尺寸大小和画质效果，工业中会注重交互性和使用质量。而对于嵌入式系统来说，LCD 应在满足应用需求的情况下，尽量的压低成本。

应用中常见的 LCD 可分成两种：普通 LCD 和触摸式 LCD。前者只具有图像输出功能，而后者有接收用户输入的功能。所以在尺寸、分辨率、色彩等指标相同的情况下，触摸式 LCD 的价格会比普通 LCD 贵很多。由于 MiniSys 已有了键盘输入模块，所以出于控制成本考虑，使用普通 LCD 已经足够满足应用需求。

综合各方面考虑，我选定了目前普通使用的 WG12864 作为 MiniSys 系统的 LCD。

WG12864 本身带有 KS0107 或 KS0108 控制器，其引脚说明如表 4.3 所示：

表 4.3 WG12864 引脚说明[11]

引脚编号	符号	功能
1	Vss	接地
2	Vdd	供电（+5V）
3	Vo	对比度调整
4	D/I	数据/指令
5	R/W#	读数据/写数据
6	E	下降沿使能信号
7	DB0	数据线
8	DB1	数据线
9	DB2	数据线

10	DB3	数据线
11	DB4	数据线
12	DB5	数据线
13	DB6	数据线
14	DB7	数据线
15	CS1	左半屏片选信号
16	CS2	可半屏片选信号
17	RST#	复位信号（低电平有效）
18	Vee	负电压输出
19	A	LED 供电线（+4.2V）RA=0 欧
20	K	LED 供电线（0V）

WG12864 分辨是 128×64，即每行能显示 128 个点，共有 64 行。分成左右两个屏，每屏分别由 CS1、CS2 来控制。D/I#引脚 1 表示输入是数据，0 表示输入的是指令。R/W#引脚 1 表示读 LCD 寄存器，0 表示写 LCD 寄存器。DB7~DB0 是 8 位数据总路线。所有的这些信号都是在使能信号 E 下降沿时有效的。

写入的 LCD 控制指令功能编码如表 4.4 所示：

表 4.4 WG12864 控制指令编码格式[12]

指令	R/I#	R/W#	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	功能说明
显示开/关	L	L	L	L	H	H	H	H	H	H/L	控制显示器的开或关。内部的状态和 RAM 区数据不受影响。 H：开 ；L：关
设置地址	L	L	L	H	列地址(0~63)						设置列地址
设置页(X 地址)	L	L	H	L	H	H	H	页号 (0~7)			设置行地址
显示开始行	L	L	H	H	显示开始行 (0~63)						决定从哪行开始显示
读状态	L	H	BUSY	L	ON/OFF	L	L	L	L	L	读状态 BUSY:L:就绪 H:操作中； ON/OFF:L:开 H：关 RESET：L：正常 H：重启
写显示数据	H	L	写入数据								写入数据（DB0：7）；在写入后，列地址自动加 1
读显示数据	H	H	读出数据								从相应显示 RAM 读出数据

写入显示数据前，应先将行列地址和开始显示行设置成相应的数值，除非这些寄存器的值就是你就设的值。

4.3.2 LCD 控制器的设计与实现

功能：完成向 LCD 写入控制寄存器和显示 RAM 来实现对 LCD 的控制

输入输出安排：

- CLK(I) 时钟
- RESET(I) 重启
- IOWRITE(I) 写信号
- CS(I) 片选信号
- ADDR3～ADDR0(I) 寄存器地址线
- DATAIN8～DATAIN0(I) 输入数据线
- ENA(O) LCD 工作使能信号

- CS2～CS1(O)　　片选信号
- DI(O)　　数据/指令信号
- DATAOUT7～DATAOUT0(O)　输出数据线

内部寄存器设计：

◆ ff60h　　输出的指令

9	8	7～0
CS2(1 有效；0 无效)	CS1(1 有效；0 无效)	指令码

◆ ff64h　　输出的显示数据

9	8	7～0
CS2(1 有效；0 无效)	CS2(1 有效；0 无效)	显示数据

实现的核心代码：

```
module LCD(CLK,RESET,CS,IOWRITE,ADDR,DATAIN,IDOUT,DI,ENA,CS1,CS2);
    .....
    reg [9:0]InstOut,DataOut;
    reg s;

    assign IDOUT=s?DataOut[7:0]:InstOut[7:0]; //输出数据
    assign CS1=s?DataOut[8]:InstOut[8]; //片选 1,s 为 1 表示输出指令,0 输出数据
    assign CS2=s?DataOut[9]:InstOut[9]; //片选 2
    assign DI=s;

    reg write;
    reg temp;

    always@(posedge CLK or posedge RESET)
     if(RESET)
            //IOWRITE 上升沿,并且选片有效时,写入相应寄存器
     begin
       InstOut<=0;DataOut<=0;
        write<=1'b0;
        s<=1'b0;
     end
     else if(CS&&IOWRITE)
     begin
        if(ADDR==4'b0000)
        begin
            InstOut<=DATAIN;
            s<=1'b0;
        end        else if(ADDR==4'b0100)
        begin
            DataOut<=DATAIN;
            s<=1'b1;
       end
       write<=~write;
     end

    always@(negedge CLK or posedge RESET) //LCD 使能信号 ENA 的产生
    if(RESET)
    begin
       temp<=1'b0;
       ENA<=1'b1;
    end
    else if(temp!=write) //当有新的写入请求,会在接下来的 CLK 上升沿时刻产生 ENA 的下降沿
    begin
       temp<=write;
       ENA<=1'b0;
    end
    else
       ENA<=1'b1;
```

.....

在实现中,需要注意的时 ENA 下降沿产生的时机,因为这直接关系到写给 LCD 的数据。在 ENA 下降沿产生前，必须使 CS1、CS2、DI、RW，D7～D0 信号都已准备好。在这选择 CLK 下降沿触发产生 ENA 下降沿。上面设计用了 write 和 temp 信号相配合来实现是否有新操作的判断工作。

4.3.3 仿真测试

经过 Quartus Ⅱ 7.2 (32-Bit)编译，该 LCD 控制器用 28 个 LE 实现。
波形仿真如图 4.13 LCD 仿真波形图所示：

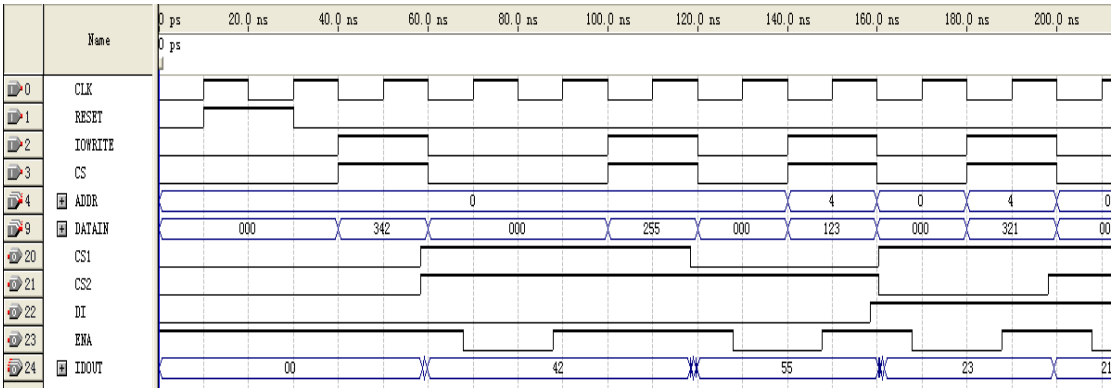

图 4.13 LCD 控制器仿真波形

第 5 章 MiniSys SoC 系统整合与仿真测试

5.1 MiniSys SoC 系统整合

乘除法运算单元与 CPU 整合

对于 MiniSys CPU 因为增加乘除法运算单元，在译码单元中需要增加必要的指令。其中包括 mult,multu,div,divu,mfhi,mflo,mthi,mtlo。这 8 条指令均是 R-type 指令，其格式设计如表 5.1 所示：

表 5.1 新增 CPU 指令格式

助记符	指令格式						示例	示例含义	操作及其解释
Bit #	**31..26**	**25..21**	**20..16**	**15..11**	**10..6**	**5..0**			
R-type	**op**	**rs**	**rt**	**rd**	**shamt**	**Func**			
mfhi	000000	0000000000		rd	00000	010000	mfhi $1	$1=HI	rd<-HI
mflo	000000	0000000000		rd	00000	010010	mflo $1	$1=LO	rd<-LO
mthi	000000	rs	000000000000000			010001	mthi $1	HI=$1	HI<-rd
mtlo	000000	rs	000000000000000			010011	mtlo $1	LO=$1	LO<-rd
mult	000000	rs	rt	0000000000		011000	mult $2,$3	{HI,LO}=$2*$3	{HI,LO}<-rt*rs；有符号相乘
multu	000000	rs	rt	0000000000		011001	multu $2,$3	{HI,LO}=$2×$3	{HI,LO}<-rt×rs；无符号相乘
div	000000	rs	rt	0000000000		011010	div $2,$3	HI=$2%$3 LO=$2/$3	HI<-rt%rs；LO<-rt/rs；有符号相除
divu	000000	rs	rt	0000000000		011011	divu $2,$3	HI=$2%$3 LO=$2/$3	HI<-rt%rs；LO<-rt/rs； 无符号相除

新增指令的译码如下：

```
assign multop=Rtype&!FUNC[5]&FUNC[4]&FUNC[3]&!FUNC[2]&!FUNC[1]&!FUNC[0];
assign multuop=Rtype&!FUNC[5]&FUNC[4]&FUNC[3]&!FUNC[2]&!FUNC[1]&FUNC[0];
assign divop=Rtype&!FUNC[5]&FUNC[4]&FUNC[3]&!FUNC[2]&FUNC[1]&!FUNC[0];
assign divuop=Rtype&!FUNC[5]&FUNC[4]&FUNC[3]&!FUNC[2]&FUNC[1]&FUNC[0];
assign mfloop=Rtype&!FUNC[5]&FUNC[4]&!FUNC[3]&!FUNC[2]&FUNC[1]&!FUNC[0];
assign mfhiop=Rtype&!FUNC[5]&FUNC[4]&!FUNC[3]&!FUNC[2]&!FUNC[1]&!FUNC[0];
assign mtloop=Rtype&!FUNC[5]&FUNC[4]&!FUNC[3]&!FUNC[2]&FUNC[1]&FUNC[0];
assign mthiop=Rtype&!FUNC[5]&FUNC[4]&!FUNC[3]&!FUNC[2]&!FUNC[1]&FUNC[0];
```

新增指令产生的数据相关问题处理代码：

```
assign
delay=(ALUDES[6]&((rssource&(RSI==ALUDES[4:0]))|(rtsource&(RTI==ALUDES[4:0]))))|((MULTBUSY|DI
VBUSY)&(hisource|hitarget|losource|lotarget))|((MULTBUSY|DIVBUSY)&(mulop|multop|multuop|divop|divuop
));//流水线停顿信号
assign WRITEPC=!delay;
assign WRITEIR=!delay;
```

IO 接口与 MiniSys 系统整合

对于 I^2C 总线控制器和 LCD 控制器，直接连接到 MiniSys Bus 相应地址及数据线即可正常工作。

本文第 4 章所设计的 SDRAM 控制器并不是按 MiniSys Bus 接口设计的，而

是按 Cache 接口设计的，所以并不能直接连接到 MiniSys Bus 上。为了测试 SDRAM 控制器，在 MiniSys Bus 与 SDRAM 控制器之间新增了一个接口部件（代码见附录二）。SDRAM 的片选信号 CS 接 FF80 的译码片选。这样，可以按如下方法使用 SDRAM 控制器：

1) 无论是读或者写，将所要进行读或写操作的 SDRAM 存储单元地址前保存到 FF44H 寄存器中。
2) 读 FF40H 寄存器（其定义如表 5.1），直到 SDRAM_BUSY 位无效。

表 5.1 FF40 寄存器域

31~2	1	0
未定义	SDRAM_READY	SDRAM_BUSY

3) 若写操作，向 FF84H 寄存器写所要写给 SDRAM 的数据，写操作到此就完成了；若是读操作，读取 FF84H 一次。
4) 对于读操作，读 FF40H 寄存器，直到其中的 SDRAM_READY 位有效。此时数据已被保存到 FF84H 寄存器中，读出即可。

5.2 MiniSys SoC 系统仿真测试

将乘除法运算单元及相应的译码指令加入 CPU 中，经过 Quartus II 7.2(32-Bit)编译，CPU 的主频率最高可达 24MHz，且 CPU 中的关键路径不是乘除法运算单元。可以说明，前面第 3 单设计的乘除法运算单元是适用于该 CPU 的。

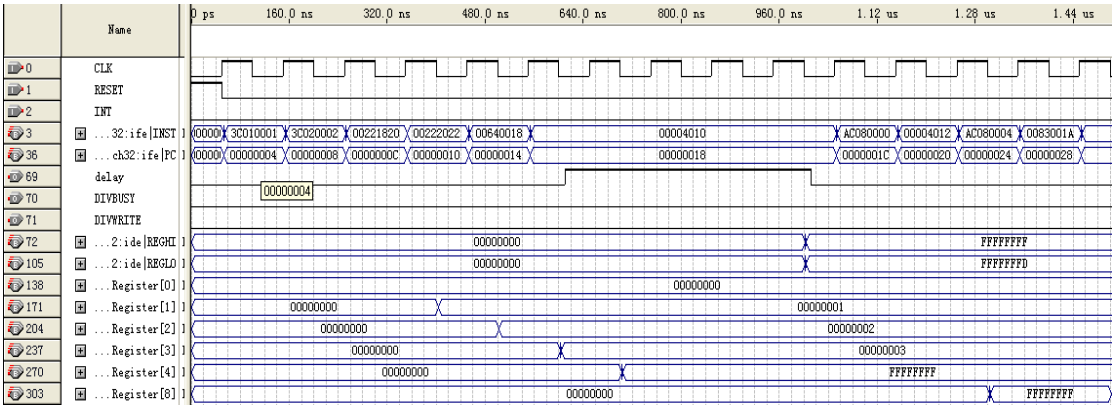

图 5.1 CPU 测试程序时序图 1

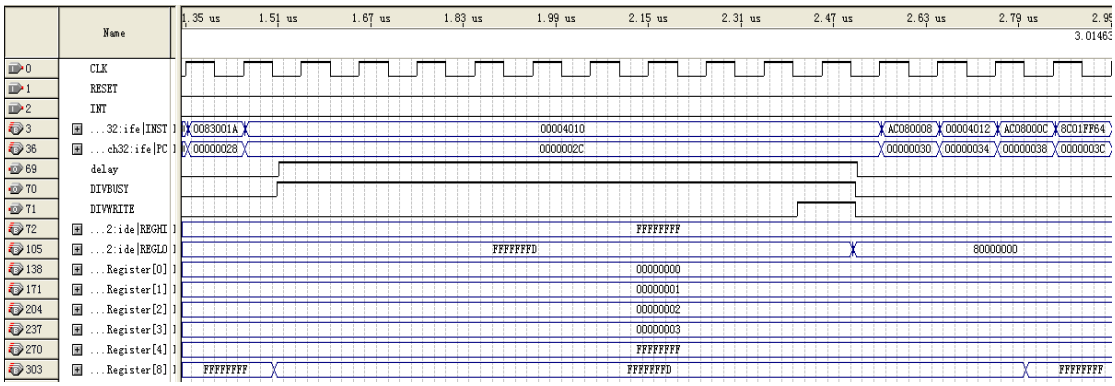

图 5.2 CPU 测试程序时序图 2

Addr	+0	+1	+2	+3	+4	+5	+6	+7
000	FFFFFFFF	FFFFFFFD	FFFFFFFF	80000000	00000000	00000000	00000000	00000000
008	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
010	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
018	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
020	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
028	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000

图 5.3 CPU 测试程序时内存单元

测试程序代码见附录一。图 5.1 中有 mult 指令和 mfhi 的数据相关造的成流水停顿。图 5.2 中有 div 指令和 mfhi 的数据相关造成的流水停顿。图 5.3 是内存单元最终运算结果，经分析可得结果正确，通过测试。

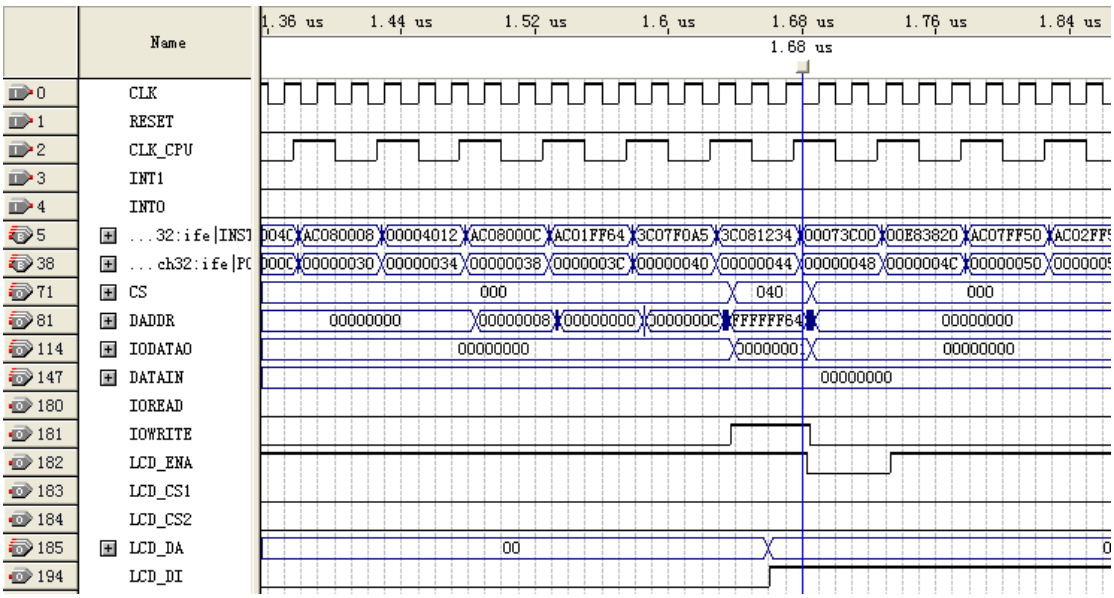

图 5.4 MiniSys SoC 系统中 LCD 测试时序

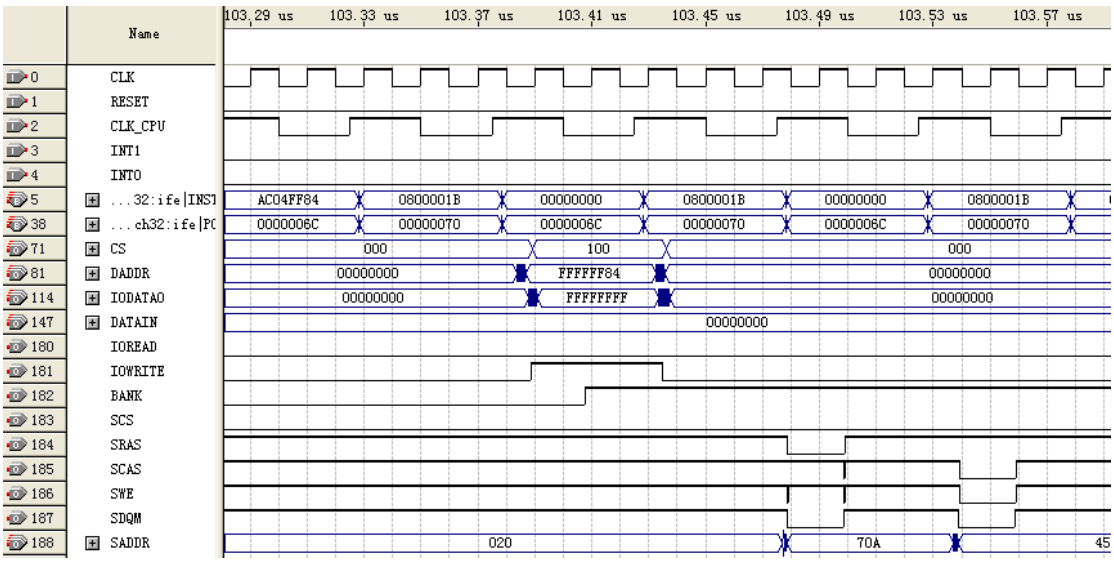

图 5.5 MiniSys SoC 系统中 SDRAM 控制器测试时序

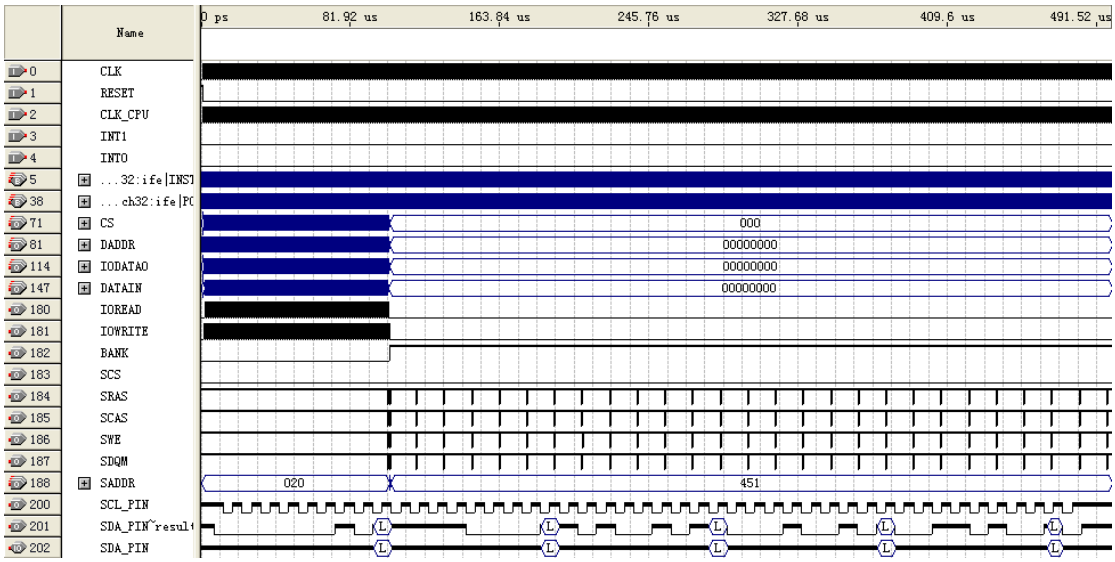

图 5.6 MiniSys SoC 系统 I^2C 总线控制器测试时序

如图 5.4 是测试程序中 LCD 的时序。图 5.5 是测试程序中 SDRAM 控制器部件。可以看出，在 103μs 程序检测到 SDRAM 空闲，就向 SDRAM 发出了写数据指令，且时序和数值均正确。

如图 5.6 显示的是测试程序中 I^2C 总线控制器的时序，可以看出是向 I^2C 总线上的 0x01 设备写数据 F0A51234，与应得测试结果相同。同时还可以看到，SDRAM 控制器每隔一段时间会向 SDRAM 发出刷新指令，设计刷新周期时间是 15μs 左右。

第 6 章 结论

6.1 总结

为了更加完善 MiniSys SoC 系统结构，增加该系统的实用性，并能够最终成为一个完整的嵌入式开发平台，需要对该系统做必要的补充。在本次毕业设计中，完成了对现有 CPU 流水结构的优化，增加了对新增部件的指令支持，构建了协处理器数据通道及 Cache 的数据通道[13]；设计并实现适合了 MiniSys CPU 的乘/除法运算单元及相关指令；此外，还成功地为 MiniSys SoC 系统新增了三种 I/O 接口部件，极大的增强了该系统的实用性和可扩展性。

本次设计的特色有以下几点：

- 优化的 CPU 流水线设计，并预留了协处理器和 Cache 的数据通道，为后续设计做提前做了准备。
- 结合 FPGA 设计的特点以及 MiniSys SoC 系统的需求，对原始的乘法累加算法和除法加减交替算法做了相应的改进，基于此改进算法设计实现了 MiniSys CPU 的乘/除法运算单元。
- I^2C 总线控制器的设计与实现，不仅解决了总线双向通信的难题，还用了一种十分合理的代码风格，使整个硬件电路达到优化的效果。

6.2 下一步的工作

在完成了本次毕业设计的工作后，已经使整个 MiniSys SoC 系统进一步完善与合理。接下来，应为 MiniSys CPU 增加 Cache 结构及相应指令，使 SDRAM 能够高效的融合到该系统中来。此外，实际应用中，浮点运算是必不可少的，所以浮点运算部件的增加也是比较重要的一项工作。浮点运算应由预留的协处理器 0 单元完成。在完成了以上工作的基础上，还应增加指令集，优化现有的硬件电路，特别是 CPU 的主频提高，这样才能使整个 MiniSys SoC 系统真正满足实际应用需求。

对于 IO 接口部件，应该继承增加实用的部件，如 CANBUS，USB 等。现有 SDRAM 控制器可以改进为一个通用的 SDRAM 控制器[14]，而不是专门为一种芯片设计以及 SDRAM 采用性能更高的 DDR2 SDRAM[15]。

致谢

首先，感谢我的指导老师杨全胜。本论文的选题、工作、撰写都是在他悉心的指导下完成的。他丰富的学术知识、严谨的治学态度是我学习的榜样。也要感谢他为我们的毕业设计提供的良好的硬件条件是我顺利完成的前题。

其次，感谢我的舍友和同学，感谢我的朋友和家人，你们的陪伴和鼓励，给我以前进的动力。

最后，感谢网络上一些不知名的朋友，感谢你们给我的指导和帮助。

参考文献

[1] 朱子玉，李亚民．CPU 芯片逻辑设计技术．清华大学出版社． 2005-01

[2] 澎澄谦．挑战 SoC—基于 NIOS 的 SoPC 设计与实践．清华大学出版社．

[3] 王健，朱怡健，朱敏．计算机组成原理(第 2 版)．东南大学出版社．2004-02

[4] 王月胜．MIPS32 位微处理器中乘除法单元的设计和实现：[硕士学位论文]．上海：同济大学电路与系统专业，2003

[5] 葛亮．一种定点运算部件的设计与实现：[硕士学位论文]．北京：中国科学院计算技术研究所计算机系统结构专业，2002

[6] 曹瑾．MIPS 嵌入式 CPU 中除法单元的设计实现：[硕士学位论文]．上海：同济大学信号与信息处理专业，2004

[7] 杨海涛，苏涛．基于 FPGA 的 SDRAM 控制器的设计和实现．西安电子科技大学．2007， 1 ：8-12

[8] 曹华，邓彬．使用 Verilog 实现基于 FPGA 的 SDRAM 控制器．电子科技大学．2005-1：53-60

[9] K4S161622D-TC/L10 Data Sheet，Samsung

[10] THE I^2-BUS SPECIFICATION VERSON 2.1 . Philips Semiconductors．JANUARY 2000

[11] WG12864A Data Sheet，Winstar

[12] KS0108B Data Sheet，Samsung

[13] Dominic Sweetman．See MIPS Run．机械工业出版社．2007-02

[14] 刘宇，李玉山，吕菱．一种通用 SDRAM 控制器的设计．电子设计应用．2003-12：19-21

[15] 须文波，胡丹．DDR2 SDRAM 控制器的 FPGA 实现．江南大学学报(自然科学版)．2006-04：145-148

附录一

测试程序：

```
WIDTH=32;
DEPTH=512;

ADDRESS_RADIX=HEX;
DATA_RADIX=HEX;

CONTENT BEGIN
    000   :   3C010001;          --lui  $1,1
    001   :   3C020002;          --lui  $2,2
    002   :   00221820;          --add $3,$1,$2
    003   :   00222022;          --sub $4,$1,$2
    004   :   00640018;          --mult      $3,$4
    005   :   00004010;          --mfhi      $8
    006   :   AC080000;          --sw  $8,0($0)
    007   :   00004012;          --mflo      $8
    008   :   AC080004;          --sw  $8,4($0)
    009   :   0083001A;          --div  $4,$3
    00A   :    00004010;         --mfhi $8
    00B   :    AC080008;         --sw  $8,8($0)
    00C   :    00004012;         --mflo $8
    00D   :    AC08000C;         --sw  $8,C($0)
    00E   :    AC01FF64;         --sw  $1,FF64($0)
    00F   :    3C07F0A5;         --lui  $7,F0A5
    010   :    3C081234;         --lui  $8,1234
    011   :    00073C00;         --sll  $7,$7,10
    012   :    00E83820;         --add $7,$7,$8
    013   :    AC07FF50;         --sw  $7,FF50($0)
    014   :    AC02FF54;         --sw  $2,FF54($0)
    015   :    00074302;         --srl  $8,$7,C
    016   :    AC08FF44;         --sw  $8,FF44($0)
    017   :    8C09FF40;         --lw  $9,FF40($0)
    018   :    31290001;         --addi      $9,$9,0001
    019   :    1029FFFC;         --beq $1,$9,FFFC
    01A   :    AC04FF84;         --sw  $4,FF84($0)
    01B   :    0800001B;         --J    1B
    [01C..1FF]   :    00000000;
END;
```

附录二

```
module
RSDRAM(CLK_CPU,CLK,CS,ADDR,IO
READ,IOWRITE,IODATA,RESET,SDR
AM_BUSY,SDRAM_READY,DAI,RDA
TAO,address);
    input CLK_CPU,CLK,RESET;
    input IOREAD,IOWRITE,CS;
    input [3:0]ADDR;
    input [31:0]DAI;
    input
SDRAM_BUSY,SDRAM_READY;
    input [31:0]IODATA;
    output reg[31:0]RDATAO;
    output reg[31:0]address;



    reg [31:0]DATA;
    reg clr,sign;
    reg set,st;
    reg READY;

    always@(posedge CLK or posedge
RESET)
    if(RESET)
    DATA<=0;
    else if(SDRAM_READY)
    DATA<=DAI;

    always@(posedge CLK or posedge
RESET)
    if(RESET)
    st<=0;
    else if(SDRAM_READY)
    st<=~st;

    always@(*)
    if(RESET)
    clr<=0;
    else if(clr!=sign)
clr<=sign;

    always@(*)
    if(RESET)
    set<=0;
    else if(set!=st)
    set<=st;

    always@(negedge CLK)
        if(clr!=sign)
        READY<=0;
    else if(set!=st)
        READY<=1;

    always@(posedge CLK_CPU or
posedge RESET)
    if(RESET)
    begin
    RDATAO<=0;  sign<=0;
    end
    else if(CS&&IOREAD)
    begin
    if(ADDR==4'b0000)
    beginRDATAO<={30'b0,READY,S
DRAM_BUSY};sign<=~sign;
    end
    else if(ADDR==4'b0100)
    RDATAO<=DATA;
    end

    always@(posedge CLK_CPU or
posedge RESET)
        if(RESET)
            address<=0;
        else if(CS&&IOWRITE)
            if(ADDR==4'b0100)
    address<=IODATA;

endmodule
```

附录三

乘/除法运算单元代码：

```
module MulDiv(CLK,RESET,ALUOP,ALUA,ALUB,ALUHI,ALULO,MULTWRITE,MULTBUSY,DIVWRITE,DIVBUSY);
    input CLK,RESET;
    input [4:0]ALUOP;
    input [31:0]ALUA,ALUB;
    output reg [31:0]ALUHI,ALULO;
    output reg MULTWRITE,MULTBUSY,DIVWRITE,DIVBUSY;


    reg [3:0]state;


    wire [36:0]temp[31:0];
    reg [37:0]result0,result1,result2,result3,result4;
    reg [33:0]result5;
    reg [31:0]A,B;
    reg sign,tm;
    reg [62:0]temp1[31:0];
    reg minus;



    assign temp[0]=B[0]?{5'b0,A}:37'b0;
    assign temp[1]=B[1]?{4'b0,A,1'b0}:37'b0;
    assign temp[2]=B[2]?{3'b0,A,2'b0}:37'b0;
    assign temp[3]=B[3]?{2'b0,A,3'b0}:37'b0;
    assign temp[4]=B[4]?{1'b0,A,4'b0}:37'b0;
    assign temp[5]=B[5]?{A,5'b0}:37'b0;

    assign temp[6]=B[6]?{5'b0,A}:37'b0;
    assign temp[7]=B[7]?{4'b0,A,1'b0}:37'b0;
    assign temp[8]=B[8]?{3'b0,A,2'b0}:37'b0;
    assign temp[9]=B[9]?{2'b0,A,3'b0}:37'b0;
    assign temp[10]=B[10]?{1'b0,A,4'b0}:37'b0;
    assign temp[11]=B[11]?{A,5'b0}:37'b0;

    assign temp[12]=B[12]?{5'b0,A}:37'b0;
    assign temp[13]=B[13]?{4'b0,A,1'b0}:37'b0;
    assign temp[14]=B[14]?{3'b0,A,2'b0}:37'b0;
```

```
assign temp[15]=B[15]?{2'b0,A,3'b0}:37'b0;
assign temp[16]=B[16]?{1'b0,A,4'b0}:37'b0;
assign temp[17]=B[17]?{A,5'b0}:37'b0;

assign temp[18]=B[18]?{5'b0,A}:37'b0;
assign temp[19]=B[19]?{4'b0,A,1'b0}:37'b0;
assign temp[20]=B[20]?{3'b0,A,2'b0}:37'b0;
assign temp[21]=B[21]?{2'b0,A,3'b0}:37'b0;
assign temp[22]=B[22]?{1'b0,A,4'b0}:37'b0;
assign temp[23]=B[23]?{A,5'b0}:37'b0;

assign temp[24]=B[24]?{5'b0,A}:37'b0;
assign temp[25]=B[25]?{4'b0,A,1'b0}:37'b0;
assign temp[26]=B[26]?{3'b0,A,2'b0}:37'b0;
assign temp[27]=B[27]?{2'b0,A,3'b0}:37'b0;
assign temp[28]=B[28]?{1'b0,A,4'b0}:37'b0;
assign temp[29]=B[29]?{A,5'b0}:37'b0;

assign temp[30]=B[30]?{5'b0,A}:37'b0;
assign temp[31]=B[31]?{4'b0,A,1'b0}:37'b0;

always@(posedge CLK or posedge RESET)
begin
    if(RESET)
    begin
        state<=4'b0;
        ALUHI<=0;ALULO<=0;
        MULTWRITE<=1'b0;MULTBUSY<=1'b0;
        DIVWRITE<=1'b0;DIVBUSY<=1'b0;
        sign<=1'b0;
        tm<=1'b0;
    end
    else
        case(state)
        4'b0000:
        begin
            MULTWRITE<=1'b0;DIVWRITE<=1'b0;
            MULTBUSY<=1'b0;DIVBUSY<=1'b0;
            if(ALUOP==5'b00110)
            begin
                A<=ALUA;B<=ALUB;
                MULTBUSY<=1'b1;
                sign<=1'b0;
                state<=4'b0001;
```

```
        end
        else if(ALUOP==5'b01110)
        begin
            A<=ALUA[31]?{~ALUA[30:0]+1}:ALUA;
            B<=ALUB[31]?{~ALUB[30:0]+1}:ALUB;
            minus<=ALUA[31]^ALUB[31];
            A[31]<=1'b0;B[31]<=1'b0;
            MULTBUSY<=1'b1;
            sign<=1'b1;
            state<=4'b0001;
        end
        else if(ALUOP==5'b00111)
        begin
            if(ALUB[31])
            begin
                A<=ALUA;B<=ALUB;
                DIVBUSY<=1'b1;
                state<=4'b1101;
            end
            else
            begin
                A<=ALUA;B<=ALUB;
                DIVBUSY<=1'b1;
                sign<=1'b0;
                state<=4'b0100;
            end
        end
        else if(ALUOP==5'b01111)
        begin
            A<=ALUA[31]?{~ALUA[30:0]+1}:ALUA;
            B<=ALUB[31]?{~ALUB[30:0]+1}:ALUB;
            minus<=ALUA[31]^ALUB[31];
            A[31]<=1'b0;B[31]<=1'b0;
            tm<=ALUA[31];
            DIVBUSY<=1'b1;
            sign<=1'b1;
            state<=4'b0100;
        end

    end
    4'b0001:
    begin
        result0<=temp[0]+temp[1]+temp[2]+temp[3]+temp[4]+temp[5];
        result1<=temp[6]+temp[7]+temp[8]+temp[9]+temp[10]+temp[11];
```

```
                result2<=temp[12]+temp[13]+temp[14]+temp[15]+temp[16]+temp[17];
                result3<=temp[18]+temp[19]+temp[20]+temp[21]+temp[22]+temp[23];
                result4<=temp[24]+temp[25]+temp[26]+temp[27]+temp[28]+temp[29];
                result5<=temp[30]+temp[31];
                state<=4'b0010;
            end
            4'b0010:
            begin
                state<=4'b0011;

    {ALUHI,ALULO}<=result0+{result1,6'b0}+{result2,12'b0}+{result3,18'b0}+{result4,24'b0}+{r
esult5,30'b0};
            end
            4'b0011:
            begin
                state<=4'b0000;
                MULTWRITE<=1'b1;
                if(sign)
                begin
            {ALUHI,ALULO}<=minus?(~{ALUHI[30:0],ALULO}+1):{ALUHI,ALULO};
                    ALUHI[31]<=minus;
                end
            end
                4'b0100:
            begin
                DIVBUSY<=1'b1;
                state<=4'b0101;
            temp1[3]={31'b0,A}-{B,31'b0};
            temp1[2]=temp1[3][62]?(temp1[3]+{1'b0,B,30'b0}):(temp1[3]-{1'b0,B,30'b0});
            temp1[1]=temp1[2][61]?(temp1[2]+{2'b0,B,29'b0}):(temp1[2]-{2'b0,B,29'b0});
            temp1[0]=temp1[1][60]?(temp1[1]+{3'b0,B,28'b0}):(temp1[1]-{3'b0,B,28'b0});
            ALULO[31:28]=~{temp1[3][62],temp1[2][61],temp1[1][60],temp1[0][59]};
            end
            4'b0101:
            begin
                state<=3'b0110;
            temp1[3]=temp1[0][59]?(temp1[0]+{4'b0,B,27'b0}):(temp1[0]-{4'b0,B,27'b0});
            temp1[2]=temp1[3][58]?(temp1[3]+{5'b0,B,26'b0}):(temp1[3]-{5'b0,B,26'b0});
            temp1[1]=temp1[2][57]?(temp1[2]+{6'b0,B,25'b0}):(temp1[2]-{6'b0,B,25'b0});
            temp1[0]=temp1[1][56]?(temp1[1]+{7'b0,B,24'b0}):(temp1[1]-{7'b0,B,24'b0});
            ALULO[27:24]=~{temp1[3][58],temp1[2][57],temp1[1][56],temp1[0][55]};
            end
            4'b0110:
            begin
```

```
                state<=4'b0111;
            temp1[3]=temp1[0][55]?(temp1[0]+{8'b0,B,23'b0}):(temp1[0]-{8'b0,B,23'b0});
            temp1[2]=temp1[3][54]?(temp1[3]+{9'b0,B,22'b0}):(temp1[3]-{9'b0,B,22'b0});
            temp1[1]=temp1[2][53]?(temp1[2]+{10'b0,B,21'b0}):(temp1[2]-{10'b0,B,21'b0});
            temp1[0]=temp1[1][52]?(temp1[1]+{11'b0,B,20'b0}):(temp1[1]-{11'b0,B,20'b0});
            ALULO[23:20]=~{temp1[3][54],temp1[2][53],temp1[1][52],temp1[0][51]};
                end
                4'b0111:
                begin
                    state<=4'b1000;
            temp1[3]=temp1[0][51]?(temp1[0]+{12'b0,B,19'b0}):(temp1[0]-{12'b0,B,19'b0});
            temp1[2]=temp1[3][50]?(temp1[3]+{13'b0,B,18'b0}):(temp1[3]-{13'b0,B,18'b0});
            temp1[1]=temp1[2][49]?(temp1[2]+{14'b0,B,17'b0}):(temp1[2]-{14'b0,B,17'b0});
            temp1[0]=temp1[1][48]?(temp1[1]+{15'b0,B,16'b0}):(temp1[1]-{15'b0,B,16'b0});
            ALULO[19:16]=~{temp1[3][50],temp1[2][49],temp1[1][48],temp1[0][47]};
                end
                4'b1000:
                begin
                    state<=4'b1001;
            temp1[3]=temp1[0][47]?(temp1[0]+{16'b0,B,15'b0}):(temp1[0]-{16'b0,B,15'b0});
            temp1[2]=temp1[3][46]?(temp1[3]+{17'b0,B,14'b0}):(temp1[3]-{17'b0,B,14'b0});
            temp1[1]=temp1[2][45]?(temp1[2]+{18'b0,B,13'b0}):(temp1[2]-{18'b0,B,13'b0});
            temp1[0]=temp1[1][44]?(temp1[1]+{19'b0,B,12'b0}):(temp1[1]-{19'b0,B,12'b0});
            ALULO[15:12]=~{temp1[3][46],temp1[2][45],temp1[1][44],temp1[0][43]};
                end
                4'b1001:
                begin
                    state<=4'b1010;
temp1[3]=temp1[0][43]?(temp1[0]+{20'b0,B,11'b0}):(temp1[0]-{20'b0,B,11'b0});
            temp1[2]=temp1[3][42]?(temp1[3]+{21'b0,B,10'b0}):(temp1[3]-{21'b0,B,10'b0});
            temp1[1]=temp1[2][41]?(temp1[2]+{22'b0,B,9'b0}):(temp1[2]-{22'b0,B,9'b0});
            temp1[0]=temp1[1][40]?(temp1[1]+{23'b0,B,8'b0}):(temp1[1]-{23'b0,B,8'b0});
            ALULO[11:8]=~{temp1[3][42],temp1[2][41],temp1[1][40],temp1[0][39]};
                end
                4'b1010:
                begin
                    state<=4'b1011;
temp1[3]=temp1[0][39]?(temp1[0]+{24'b0,B,7'b0}):(temp1[0]-{24'b0,B,7'b0});
            temp1[2]=temp1[3][38]?(temp1[3]+{25'b0,B,6'b0}):(temp1[3]-{25'b0,B,6'b0});
            temp1[1]=temp1[2][37]?(temp1[2]+{26'b0,B,5'b0}):(temp1[2]-{26'b0,B,5'b0});
            temp1[0]=temp1[1][36]?(temp1[1]+{27'b0,B,4'b0}):(temp1[1]-{27'b0,B,4'b0});
            ALULO[7:4]=~{temp1[3][38],temp1[2][37],temp1[1][36],temp1[0][35]};
                end
                4'b1011:
```

```
                begin
                    state<=4'b1100;
    temp1[3]=temp1[0][35]?(temp1[0]+{28'b0,B,3'b0}):(temp1[0]-{28'b0,B,3'b0});
        temp1[2]=temp1[3][34]?(temp1[3]+{29'b0,B,2'b0}):(temp1[3]-{29'b0,B,2'b0});
        temp1[1]=temp1[2][33]?(temp1[2]+{30'b0,B,1'b0}):(temp1[2]-{30'b0,B,1'b0});
        temp1[0]=temp1[1][32]?(temp1[1]+{31'b0,B}):(temp1[1]-{31'b0,B});
        ALUHI=temp1[0][32]?(temp1[0][31:0]+B):temp1[0][31:0];
        ALULO[3:0]=~{temp1[3][62],temp1[2][61],temp1[1][60],temp1[0][59]};
                end
                4'b1100:
                begin
                    DIVWRITE<=1'b1;
                    state<=4'b0;
                    if(sign)
                    begin
                        ALUHI<=tm?(~ALUHI[30:0]+1'b1):ALUHI[30:0];
                        ALUHI[31]<=tm;
                        ALULO<=minus?(~ALULO[30:0]+1'b1):ALULO;
                        ALULO[31]<=minus;
                    end
                end
                4'b1101:
                begin
                    DIVWRITE<=1'b1;
                    state<=4'b0;
                    if(A<B)
                    begin
                        ALUHI=A;ALULO=32'b0;
                    end
                    else
                    begin
                        ALUHI=A-B;ALULO=32'b1;
                    end
                end
                default:
                begin
                    ALUHI<=0;ALULO<=0;state<=4'b0000;
                end
                endcase
    end
endmodule
```