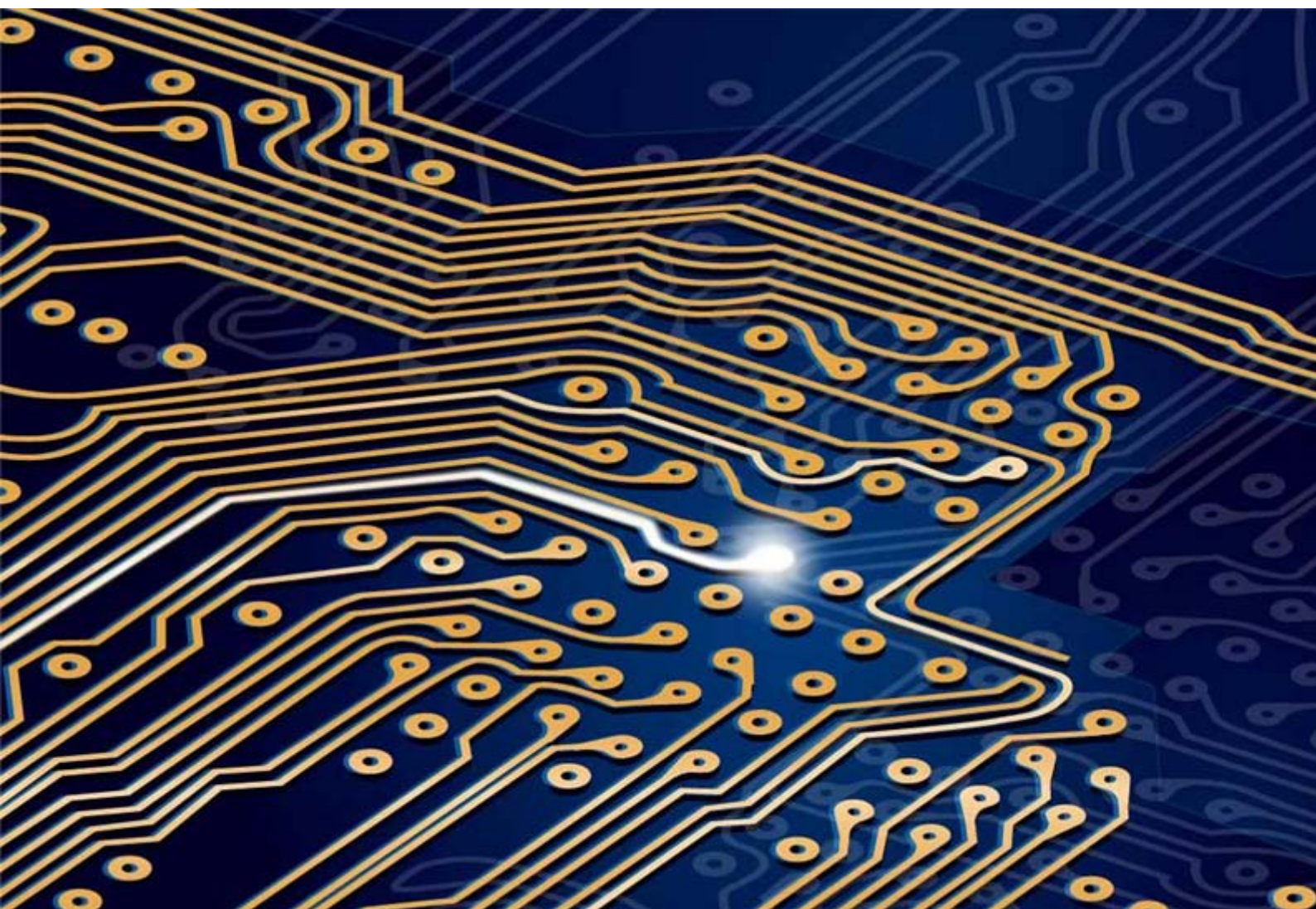


**Engintime**

# Dream Logic 实验指导



北京英真时代科技有限公司

# Dream Logic 实验指导

英真时代 编著

版本 3.6

北京英真时代科技有限公司

### 版权与授权声明

北京英真时代科技有限公司保留本电子书籍的修改和正式出版的所有权利。您可以从北京英真时代科技有限公司免费获得本电子书籍，但是授予您的权利只限于使用本电子书籍实现个人学习和研究的目的，在未获得北京英真时代科技有限公司许可的情况下，您不能以任何目的传播、复制或抄袭本书之部分或全部内容。

© 北京英真时代科技有限公司，保留所有权利。

## 北京英真时代科技有限公司

地址：北京市房山区拱辰大街 98 号 7 层 0825

邮编：102488

电话：010-60357081

手机：18501161622 / 18500560938

QQ：964515564

邮箱：support@tevation.com

网址：<http://www.engintime.com>

# 目录

DREAM LOGIC简介.....	7
第一部分数字电路实验 .....	8
实验 1 集成门电路的组合电路 .....	9
实验 2 编码器和译码器 .....	22
实验 3 加法器 .....	29
实验 4 触发器 .....	33
实验 5 计数器和移位寄存器 .....	38
实验 6 顺序脉冲发生器 .....	45
实验 7 序列信号的产生和检测 .....	48
实验 8 存储器 .....	51
实验 9 密码电子锁的设计.....	55
实验 10 七人表决器的设计.....	56
实验 11 智力竞赛抢答器的设计 .....	58
实验 12 路口交通灯管理系统的设计 .....	61
实验 13 电梯控制器的设计.....	66
实验 14 数字钟的设计 .....	72
第二部分 计算机组成原理实验.....	74
实验 1 实验环境的使用 .....	75
实验 2 寄存器实验.....	87
实验 3 运算器实验.....	96
实验 4 程序计数器实验 .....	103
实验 5 存储器实验.....	108
实验 6 控制器实验.....	112
实验 7 数据通路实验 .....	117
实验 8 中断实验.....	125
实验 9 指令和微指令设计实验 .....	131
实验 10 阵列乘法器实验 .....	137
第三部分 MIPS微体系结构处理器实验.....	142

<b>第 1 章体系结构概述 .....</b>	<b>143</b>
一、    体系结构 .....	143
二、    MIPS体系结构 .....	144
三、    MIPS指令 .....	145
四、    MIPS微体系结构处理器 .....	147
五、    利用DREAM LOGIC开发MIPS微处理器 .....	148
<b>第 2 章单周期MIPS微处理器 .....</b>	<b>149</b>
一、    单周期MIPS微处理器设计过程 .....	149
二、    实验任务 .....	150
<b>第 3 章多周期MIPS微处理器 .....</b>	<b>158</b>
一、    多周期MIPS微处理器的开发背景 .....	158
二、    实验任务 .....	158
<b>第 4 章五级流水线MIPS微处理器 .....</b>	<b>161</b>
一、    流水线技术 .....	161
二、    实验任务 .....	162
<b>第四部分微机原理与接口技术实验 .....</b>	<b>169</b>
实验 1 实验环境的使用 .....	170
实验 2 可编程并行接口 8255 .....	180
实验 3 可编程定时器/计数器 8253 .....	185
实验 4 可编程中断控制器 8259A .....	190
实验 5 串行通信接口 8250 .....	201
实验 6 DMA控制器 8237A .....	205
实验 7 数/模转换器DAC0832 .....	210
实验 8 模/数转换器ADC0809 .....	213
附录A DM1000 八位模型机规格说明 .....	217
附录B MIPS指令(32 位) .....	224
附录C数字电路实验芯片功能说明 .....	230
优先编码器 74LS148 .....	231
3-8 译码器 74LS138 .....	232
7 段数码显示器 74LS48 .....	233
4 位超前进位加法器 74LS283 .....	234
4 位算术逻辑运算器 74LS181 .....	235
D触发器 .....	236
JK触发器 .....	236
SR触发器 .....	237
T触发器 .....	238
四位双向移位寄存器 74LS194 .....	238

---

8 选 1 数据选择器 74LS151D .....	239
二-五进制计数器 74LS90 .....	240
参考文献.....	241

# Dream Logic 简介

获得人生中的成功需要的专注与坚持不懈多过天才与机会。

——C. W. Wendte

Dream Logic 是一款集成了数字电路、模拟电路以及数字系统、数模混合系统设计与仿真的 EDA 软件，Dream Logic 具有元器件库丰富、原理图绘制方便、交互式仿真、可视化程度高、扩展性强等特点。Dream Logic 还配有精心设计的实验教学方案以及大量的电路设计图样例，便于读者学习和研究。

Dream Logic 主要具有以下特色：

- **与 CodeCode.net 平台深度整合。** Dream Logic 已经完全接入了 CodeCode.net 平台。首先，用户可以使用在 CodeCode.net 注册的账号登录 Dream Logic 软件，获取该软件的使用授权；其次，Dream Logic 需要用到的所有实验项目模板，都托管在 CodeCode.net 上的 Git 远程库中，用户可以使用 Dream Logic 集成的 Git 功能，十分方便的将这些 Git 远程库克隆到本地使用；同时，用户使用 Dream Logic 设计的电路图，也可以推送到 CodeCode.net 平台上进行托管，然后使用浏览器就可以直接查看电路图。
- **元器件种类齐全。** Dream Logic 中提供了十几个型号库，其中包含各类数字器件和模拟器件，为不同功能、不同规模的电路和系统设计与仿真提供了强大的支撑。在 Dream Logic 的实际使用过程中，用户还可以根据需求自定义新的功能器件。
- **绘制原理图方便快捷。** 绘制原理图时，用户可以快速地将元器件放置到原理图中，然后通过网络、标签、总线等多样化的完成电路连线。
- **可绘制模块化的、可复用的功能电路。** 可将任意一个具有特定功能的电路封装成一个电路模块，从而可以重复使用。该功能极大地方便了复杂电路系统的设计，也符合复杂系统自顶向下功能模块划分、自底向上功能模块实现的设计原则。而且各个模块可以进行单独仿真，从而验证其功能的正确性。这就使诸如八位模型计算机、MIPS 微体系结构处理器等复杂系统的设计成为可能。
- **可视化程度高。** 在仿真过程中，可以高亮显示具有高电平的网络，还可以使用指示灯、数码管等器件显示电路的输出，这就使得电路的工作状态一目了然。此外，还提供了逻辑分析仪、电压表等可视化的仪器仪表实时显示电路的状态。在仿真计算机系统时，可以显示寄存器、存储器、指令源代码等信息，还可以进入子模块电路的内部查看其工作状态。这些可视化功能大大方便了电路的功能测试与验证。
- **交互式仿真。** 在仿真过程中，用户可以通过手动控制单周期时钟源、交互式数字常量等器件实时控制电路的输入，从而动态改变电路工作的状态，达到交互式仿真的目的。
- **提供多种体系结构微处理器。** 提供了适合用于完成计算机组成原理实验的八位模型机 DM1000，适合用于完成微机原理与接口技术实验的 Intel 8086 微处理器，适合用于计算机体系结构实验的十六位单周期 MIPS 微处理器、十六位多周期 MIPS 微处理器以及十六位五级流水线 MIPS 微处理器，用户可以将它们用于学习或研究，或者以它们为基础设计出新的微处理器。
- **完全开源的指令集和汇编器。** 为 8086、DM1000 和 MIPS 微处理器提供了开源的指令汇编器和微指令汇编器，这些软件完全使用 C 语言编写，用户可以通过修改源代码来添加自己设计的指令和微指令，或者以它们为基础编写出新的汇编器。

# 第一部分 数字电路实验

数字电路实验是按照组合逻辑电路实验、时序逻辑电路实验、有限状态机实验、数字系统设计实验的顺序从易到难安排的。每个实验在任务的安排上遵循由易到难的原则，采用由功能验证到功能设计实现的实验模式，满足不同层次学生的实验需要，使每一个学生都能从实验中受益。

**组合逻辑电路实验**包含基本门电路实验、编码器和译码器实验以及加法器实验，通过这些实验，读者可以掌握基本逻辑门设计组合逻辑电路的方法，熟悉 74LS 系列常用组合逻辑芯片的功能和使用方法，并使用这些芯片设计一定规模的组合逻辑电路，实现相应的功能。

**时序逻辑电路实验**以触发器、计数器和寄存器为代表，包含 74LS 系列的主要时序逻辑芯片，通过这些实验，读者可以熟悉 74LS 系列时序逻辑芯片的功能和使用方法，掌握时序逻辑电路的特点和一般的设计方法。

**有限状态机实验**以交通灯设计为代表，通过交通灯的设计，读者可掌握有限状态机的概念，应用场景以及设计方法，学会使用有限状态机设计复杂数字系统的控制器。

**数字系统设计实验**属于综合设计实验，该部分实验需要读者使用前面实验中掌握的知识和方法，灵活运用组合逻辑电路，时序逻辑电路以及有限状态机，分模块实现数字系统的各个子功能，然后构建一个功能复杂的数字系统。通过这些实验，将加深读者对数字电路理论知识的理解，培养读者的知识应用和数字系统设计能力。

数字电路课程是计算机专业以及电子专业的基本理论课程，数字电路实验可为读者后续的计算机组成原理等课程打下坚实的基础。



# 实验 1 集成门电路的组合电路

**实验性质：**验证+设计

**建议学时：**2 学时

## 一、实验目的

- 熟悉 Dream Logic 的基本使用方法。
- 熟悉在 Dream Logic 上绘制组合逻辑电路并仿真的过程。
- 学会设计具有特定功能的基本组合逻辑电路，并通过仿真测试其逻辑功能、修改其电路以完善最终的设计。

## 二、实验内容

请读者按照下面的步骤完成实验内容，掌握 Dream Logic 软件和 CodeCode.net 平台的基本使用方法，为完成后续实验做好准备。

### 2.1 任务（一）实验软件的基本使用方法

#### 从 CodeCode.net 平台领取任务

CodeCode.net 平台是用于教师在线布置实验任务，并统一管理学生提交的作业的平台。如果没有使用到 CodeCode.net 平台，可以跳过此部分。

1. 读者通过浏览器访问<https://www.codecode.net>，可以打开 CodeCode.net 平台的登录页面。
2. 在登录页面中，读者输入帐号和密码，点击“登录”按钮，可以登录 CodeCode.net 平台。
3. 登录成功后，在“课程”列表页面中，可以找到数字电路相关的实验课程。点击此课程的链接，可以进入该课程的详细信息页面。
4. 在课程的详细信息页面中，可以查看“课程描述”，该信息对于完成实验十分重要，建议读者认真阅读。点击左侧导航中的“任务”链接，可以打开任务列表页面。
5. 在任务列表中找到本次实验对应的任务，点击右侧的“领取任务”按钮，可以进入“领取任务”页面。
6. 在“领取任务”页面中，“新建项目名称”、“新建项目路径”、“项目所在的群组”一般使用默认值即可。点击“领取任务”按钮后，可以创建一个个人项目用于完成本次实验，并自动跳转到该项目的详情页面。
7. 个人项目的详情页面主要包括左侧的导航栏、项目信息、文件列表、readme.md 文件内容等，如图 1-1 所示。
8. 在图 1-1 中，点击红色方框中的按钮，可以复制当前项目的 URL，在本实验后面的练习中会使用这个 URL 将此项目克隆到读者计算机的本地磁盘中，从而供读者进行修改。

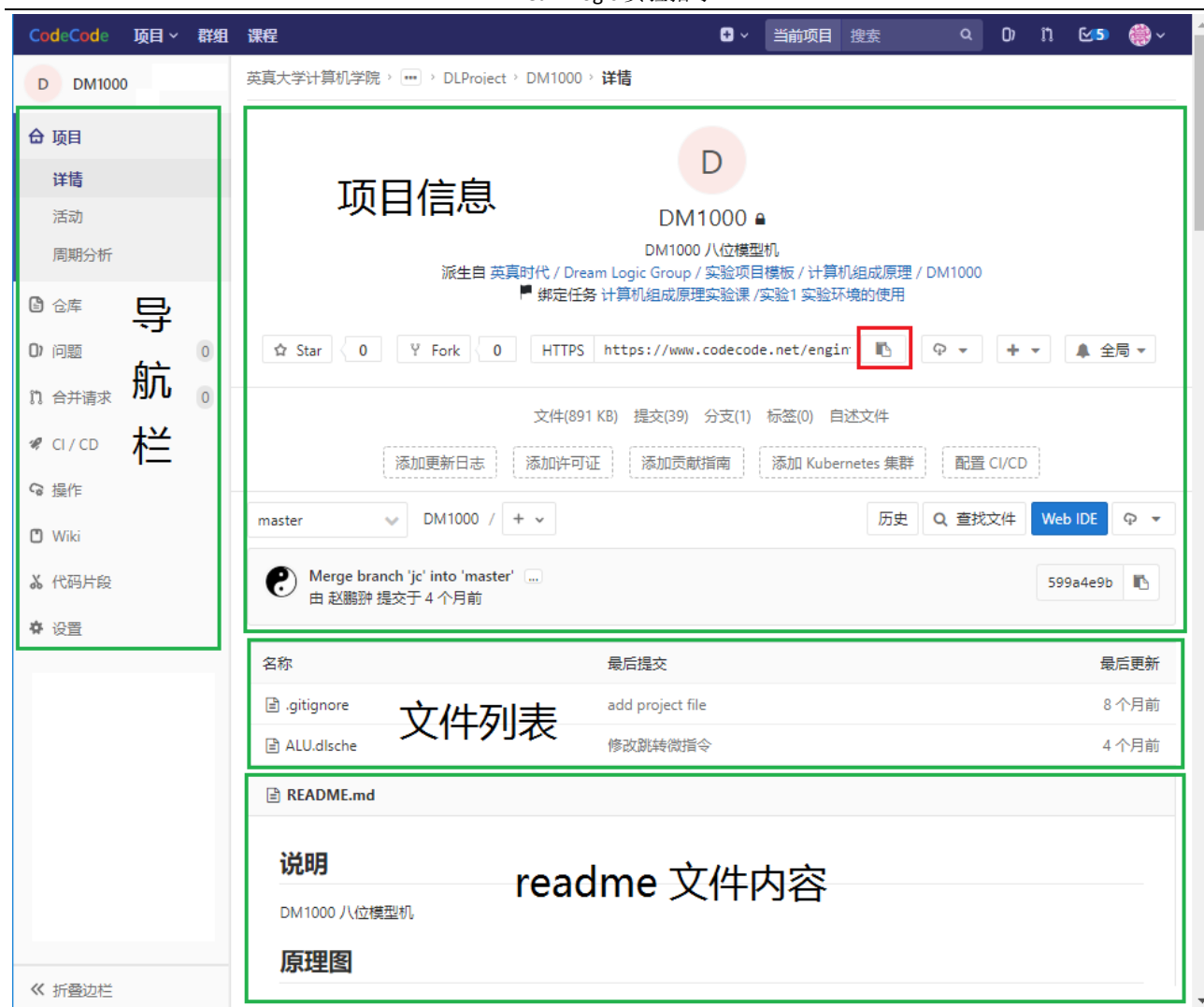


图 1-1: 个人项目的详情页面

### 启动 Dream Logic

在安装有 Dream Logic 的计算机上，可以使用两种方法来启动 Dream Logic：

- 在桌面上双击“Engintime Dream Logic”图标。
- 或者
- 点击“开始”菜单，在“所有程序”中的“Engintime Dream Logic”中选择“Engintime Dream Logic”即可启动程序。

启动 Dream Logic 后会首先打开“登录”窗口，可以使用两种方法完成登录。

- 如果读者在 CodeCode.net 平台上注册了帐号，在连接互联网的情况下，可以使用 CodeCode.net 平台中已注册的用户名和密码进行登录。
- 如果读者还没有 CodeCode.net 平台上的帐号，可以点击左下角“从加密锁获取授权”按钮，获取授权之后，完成登录。

### 主窗口布局

Dream Logic 的主窗口布局由下面的若干元素组成：

- 顶部的菜单栏、工具栏。
- 停靠在左侧的有项目管理器窗口、型号库窗口等。
- 停靠在右侧的通常是属性窗口。
- 打开原理图文档后，中间区域是电路图绘制区域。

**提示：**菜单栏、工具栏和各种工具窗口的位置可以随意拖动。如果想恢复窗口的默认布局，选择“窗口”菜单中的“重置窗口布局”即可。

### 将 CodeCode.net 平台上的项目克隆到本地

如果读者从 CodeCode.net 平台领取了任务，可以在领取任务后使用 Dream Logic 软件将读者的个人项目克隆到本地磁盘中，从而对其进行修改；如果读者没有 CodeCode.net 平台账号无法领取任务，那就只能直接将实验模板项目克隆到本地磁盘中了。

#### ● 将领取任务后新建的个人项目克隆到本地

1. 选择 Dream Logic 菜单“文件->新建->从 Git 远程库新建项目”，打开“从 Git 远程库新建项目”对话框。
2. 在“Git 远程库 URL(G)”中填入读者个人项目 Git 远程库的 URL 地址（在图 1-1 中，点击红色方框中的按钮，可以复制 URL）。
3. “项目文件夹名称”填入“test”。
4. “项目位置”选择新建项目需要保存到的磁盘目录。
5. 点击“确定”按钮后会弹出一个 Windows 控制台窗口，并开始运行 Git 克隆命令将 Git 远程库克隆到本地，一定要等克隆成功后再关闭该窗口。
6. 克隆项目成功后，在对话框中选择“克隆成功，打开新建项目”就可以打开新建的项目。

#### ● 直接将实验模板项目克隆到本地

由于 CodeCode.net 平台上提供的实验模板是开放的，所有的人都可以访问。所以，Dream Logic 也可以直接将实验模板克隆到本地磁盘。步骤如下：

1. 选择 Dream Logic 菜单“文件->新建->从 Git 远程库新建项目”，打开“从 Git 远程库新建项目”对话框。
2. 在“Git 远程库 URL(G)”中填入空白项目模板 Git 远程库的 URL 地址：<https://www.codecode.net/engintime/Dream-Logic/Project-Template/blank.git>
3. “项目文件夹名称”填入“test”。
4. “项目位置”选择新建项目需要保存到的磁盘目录。
5. 点击“确定”按钮后会弹出一个 Windows 控制台窗口，并开始运行 Git 克隆命令将 Git 远程库克隆到本地，一定要等克隆成功后再关闭该窗口。
6. 克隆项目成功后，在对话框中选择“克隆成功，打开新建项目”就可以打开新建的项目。

#### 注意：

1. 为了能正常使用从 Git 远程库新建项目功能，用户需要在本地安装 Git 客户端程序，并且在安装 Git 的过程中会有一个步骤询问是否设置 Windows 环境变量，此时一定要设置上 Windows 环境变量。
2. 文件路径中不允许包含中文，请读者在使用之初就养成使用英文命名项目或文件的习惯。

新建项目成功后，Dream Logic 会自动打开项目，并在左侧的“项目管理器”窗口中显示项目包含的所有文件。可通过菜单“视图->项目管理器”打开项目管理器窗口。刚刚新建的项目下已经包含了一个空白的原理图文件 top.dlsche，并且已经自动打开了此文件，读者可以在此原理图文件中绘制电路图，并对电路图进行仿真。

### 原理图的平移和缩放

读者可以用鼠标将原理图平移或者缩放到合适的位置。方法为：

1. 在绘图区按下鼠标中键并拖动鼠标，可移动绘图区域。
2. 在绘图区滚动鼠标滚轮可以缩放绘图区域。在软件底部的状态栏中显示了当前光标的位置坐标以及缩放比例。

## 绘制简单的原理图

接下来，读者可以按照下面的步骤在原理图文件 top.dlsche 中绘制一个验证“与门”功能的组合逻辑电路，进而掌握在原理图中绘制电路的基本方法。步骤如下：

1. 选择菜单栏中的“视图->型号库”，打开“型号库”窗口。Dream Logic 在该窗口中提供了大量的常用器件，并根据器件的类别将这些器件放到了多个型号库中，方便用户选用。
2. 在“型号库”窗口顶部的下拉列表中选择“逻辑门.dlcomp”，打开逻辑门型号库。
3. 在逻辑门型号库中，双击与门“AND”所在行，然后将鼠标移动到原理图中，就可以在原理图中放置器件了。
4. 将鼠标移动到原理图中的任意位置（同时可以通过鼠标中键平移和缩放原理图，以寻找一个合适的位置），然后单击左键，完成器件的放置。这样，就在原理图中绘制了一个与门。

**小技巧：**放置完器件后，按下 Tab 键可以重复放置同型号的器件，而按下 Esc 键可以退出器件的放置。将鼠标移到器件上并单击左键可选中该器件，此时会在右侧的“属性”窗口中显示选中对象的属性值。

5. 在型号库顶部的下拉列表中选择“数字信号源.dlcomp”，打开数字信号源型号库。
6. 在原理图中放置“时钟”和“一位交互式数字信号源”器件。它们将用于控制与门的两个输入信号。
7. 打开“数字信号显示.dlcomp”型号库，在原理图中放置一个“数字探针”，该探针用于检测与门的输出信号。当探针检测到高电平时灯亮，低电平时灯灭。
8. 所有器件放置完成后，如图 1-2 所示。在绘制原理图时，通常遵循左侧输入，右侧输出的原则。

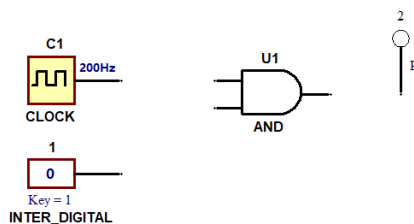


图 1-2：在原理图中绘制器件

器件绘制完成后，需要将各个器件的管脚连接起来，形成一个可工作的电路。步骤如下：

1. 选择菜单栏中的“绘制->网络”，开始绘制网络连线。
2. 将光标移动到器件管脚的末端，如图 1-3 所示。管脚末端会有一个白色的小点。



图 1-3：管脚末端的连接点

3. 当光标捕获到管脚末端的连接点时，会显示一个红色的交叉线。这就意味着，如果在此时单击鼠标左键，就会以该点为起点开始绘制一条网络，并且该网络与管脚是连接到一起的。
4. 移动光标到另一个器件管脚末端的连接点，出现红色交叉线后，单击左键，这样就在两个管脚之间绘制了一条网络。

**注意：**在绘制网络的过程中，网络节点之间的细实线或者虚线是绘制网络的预览，单击左键后细实线变为粗实线才表示一个网络线段绘制成功。此外，光标除了能捕获管脚末端的连接点以外，还能捕获网络上的任意点。

5. 按下键盘上的“Esc”键可结束一条网络的绘制。
6. 按下 Tab 键可以继续绘制新的网络连线。

7. 将各个器件的所有管脚使用网络完成连接后，如图 1-4 所示。

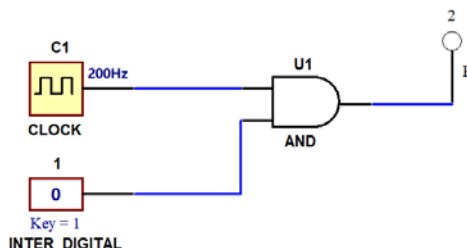


图 1-4：使用网络连接器件的管脚

### 交互式仿真

通过前面的实验内容，读者已经在原理图 top.dlsche 中绘制了一个“与门”的功能验证电路。接下来，读者按照下面的步骤，使用 Dream Logic 提供的交互式仿真功能，检验与门的逻辑功能：

1. 首先，为了仿真的效果比较便于观察，需要先将时钟频率设置的低一些。将鼠标移动到时钟上并单击选中，将其属性栏中的时钟频率修改为 10，也就是将时钟频率设置为 10Hz。
2. 选择菜单栏中的“仿真->启动仿真”，或按下键盘快捷键“F5”启动仿真。启动仿真前需要确保当前打开的原理图是 top.dlsche，因为任何时刻启动仿真都是对当前打开的原理图进行仿真。
3. 启动仿真后，观察电路中网络和器件管脚颜色的变化，以及指示灯的亮灭情况。若网络或器件管脚为红色，则表示数字信号 1，也就是高电平；若为浅灰色，则表示数字信号 0，即低电平。
4. 按下键盘上的按键“1”可以让原理图中的一位交互式数字信号在 0 和 1 之间切换，根据仿真结果检验与门的逻辑。
5. 选择菜单栏中的“仿真->结束仿真”可以结束仿真。

读者可以按照下面的步骤将时钟替换为一个一位交互式数字信号源再进行仿真。

1. 将鼠标移动到原理图中的时钟上，单击左键，选中这个时钟器件。
2. 按下键盘上的“delete”键，或选择菜单栏中的“编辑->删除”，删除选中的时钟器件。
3. 在原理图中添加一个一位交互式数字信号源器件，替换时钟作为与门的输入。如图 1-5 所示：

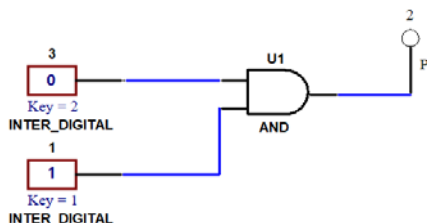


图 1-5：使用一位交互式数字信号源替换时钟

4. 启动仿真，通过键盘上的按键“1”和“2”控制两个数字信号源的输出，观察电路仿真情况。进一步验证与门的逻辑功能。
5. 结束仿真。

### 原理图常用操作

通过上面的练习，读者对原理图的绘制和仿真有了基本的掌握。接下来是绘制原理图过程中常用操作的详细介绍，读者在后续实验过程中可以参考该部分内容。

- **平移原理图。**在绘图区按下鼠标中键并拖动鼠标，可移动绘图区域。
- **缩放原理图。**在绘图区滚动鼠标滚轮可以缩放绘图区域。
- **新建原理图文件。**方法如下：
  1. 在左侧的“项目管理器”窗口中右键点击项目名称节点（根节点）。
  2. 在弹出的菜单中选择“添加->新建文件”，会弹出“添加新文件”对话框。

3. 在“添加新文件”对话框中,选择“原理图文件”模板,在“文件名称”中输入新建的原理图文件名称。文件位置默认就是项目所在文件夹,所以一般不需要修改。
4. 点击“确定”按钮后,新建的原理图文件就会被添加到项目中,并自动打开。

● **添加器件。**绘制原理图时,首先需要在原理图中添加所需器件。所有器件都只能从 Dream Logic 提供的型号库中获取。打开型号库的方法为:

1. 选择菜单栏中的“视图->型号库”,打开型号库窗口。
2. 在型号库顶部的下拉列表中选择合适的型号库。被选中型号库中的所有型号都会显示在型号列表中,选中某个型号后会在下方的预览窗口中显示型号的图符。

使用型号库中的型号在原理图中绘制器件的方法有以下三种:

1. 在型号库中选中一个型号,点击“型号库”窗口工具栏中的“放置器件”,然后在原理图中的合适位置单击鼠标左键完成器件放置。
2. 在型号库中的一个型号上单击右键,在弹出的菜单中选择“放置器件”,然后在原理图中的合适位置单击鼠标左键完成器件放置。
3. 直接双击型号库中选中一个型号,然后在原理图中的合适位置单击鼠标左键完成器件放置。

**小技巧:**在器件随鼠标移动的过程中,用户可以通过按下键盘上的“Esc”按钮取消绘制器件命令。在器件随鼠标移动的过程中,用户可以按下键盘上的空格键旋转器件到一个合适的方向后,再放置到原理图中。在器件放置完成后,可通过反复按下“Tab”重复放置相同型号的器件。

- **器件预选中、选中。**将鼠标移动到器件上,器件颜色会改变,说明器件被预选中了,单击鼠标左键就可以选中该器件。
- **全选。**按下键盘上的“Ctrl+A”可以选中原理图中的所有对象。
- **矩形选择。**选择菜单栏“编辑”中的“矩形选择”,然后在原理图中的合适位置单击鼠标左键,作为矩形的一个顶点,接着移动鼠标,可以画出一个矩形,再次单击左键,就可以选中矩形区域内的所有对象。矩形选择分为左右两种矩形,它们绘制的矩形选择区域颜色不同,选择方式也不同。其中右矩形只会选中那些完全包裹在矩形区域中的对象,而左矩形会选中那些与矩形区域有交集的对象。
- **多选。**单击预选中的器件的同时按下“Ctrl”键,可以逐个选中多个对象。
- **取消选中。**单击已经被选中的对象,就可以取消选中此对象。
- **旋转器件。**可使用旋转功能旋转器件,有以下三种旋转方法:
  - 1) 选中器件后,点击菜单栏中的“编辑”,选择“旋转”,即可将所选器件逆时针旋转 90°。
  - 2) 选中器件后,单击右键,选择菜单中的“旋转”,即可将所选器件逆时针旋转 90°。
  - 3) 选中器件后,按下空格键,可将所选器件逆时针旋转 90°。
 使用同样的方法,可以对任何一个或多个选中对象进行旋转。
- **平移器件。**当器件的位置需要调整时,可使用平移功能。首先选中器件,然后点击菜单栏“编辑”中的“平移”,或者在选中器件后,单击右键,选择菜单中的“平移”。然后,单击原理图中任意一点作为平移起始点,之后选中的器件就会随着光标移动。最后,在合适的位置单击鼠标左键即可将选中器件平移到新的位置。使用同样的方法,可以对任何一个或多个选中对象进行平移。
- **修改对象属性。**选中对象后,可以在右侧的“属性”窗口中查看或者修改对象的属性。选择菜单栏“视图”中的“属性”可以打开“属性”窗口。
- **单周期时钟。**单周期时钟是一个数字信号源器件,在“数字信号源”型号库中,如图 1-6 所示。其中的“key=A”是单周期时钟的控制键,对应于键盘上的按键“A”。在原理图中选中单周期时钟器件后,可在属性窗口中将其控制键修改为键盘上的其它按键。当启动仿真后,单周期时钟默认输出高电平信号,按下其控制键后,开始输出低电平信号,抬起按键后恢复输出高电平信号。这样,在按键按下然后抬起的过程中,产生了一个负脉冲,包含一个时钟下降沿和一个时钟上升沿。



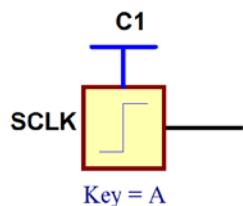


图 1-6: 单周期时钟

- **一位交互式数字信号。**一位交互式数字信号是一个数字信号源器件，在“数字信号源”型号库中，如图 1-7 所示。其中的“key=4”是其控制键，对应于键盘上的按键“4”。在原理图中选中一位交互式数字信号器件后，可在属性窗口中将其控制键修改为键盘上的其它按键。当启动仿真后，一位交互式数字信号显示数字 0 时，输出低电平信号；显示数字 1 时，输出高电平信号。通过按下控制键可以使一位交互式数字信号的输出在高电平和低电平之间切换。

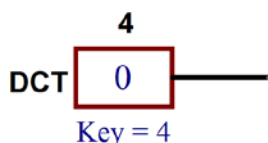


图 1-7: 一位交互式数字信号

- **使用网络连接器件的管脚。**当器件放置完成后，可以使用“网络”连接器件的管脚，操作步骤为：
  1. 选择菜单栏中的“绘制->网络”，将光标移动到器件管脚的末端，当光标捕获到管脚末端的连接点时，会显示红色交叉线，此时单击鼠标左键，就会以该点为起点开始绘制一条网络。
  2. 移动光标到另一个器件管脚末端的连接点，出现红色交叉线后，单击左键，这样就在两个管脚之间绘制了一条网络。
  3. 按下键盘上的“Esc”可结束网络的绘制。

**小技巧：**如果需要连接的两个管脚距离较远，网络可能需要由多个线段组成，在绘制的过程中就需要多次点击鼠标左键，如果点击鼠标左键后发现位置不合适，可以选择“编辑”菜单中的“撤销”(快捷键 Ctrl+Z)，取消刚刚点击左键绘制的网络线段。在绘制网络的过程中，按下空格键可以在网络的四种绘制模式之间进行切换。

- **使用网络标签连接器件的管脚。**当两个器件的管脚距离较远，不方便使用网络进行连接时，通常需要使用两个具有相同名称的网络标签为它们建立连接关系。步骤如下：
  1. 选择菜单栏中的“绘制->网络标签”。
  2. 将光标移动到器件管脚的末端，当光标捕获到管脚末端的连接点时，会显示红色交叉线，此时单击鼠标左键，就会在此管脚上放置一个网络标签。
  3. 使用同样的方法在另外一个管脚上也放置一个网络标签。
  4. 按下“Ctrl”键的同时，用鼠标依次点击刚刚放置的两个网络标签，这样可以同时选中两个网络标签对象。
  5. 在“属性”窗口中的“名称”栏中输入一个合适的网络名称，例如“NetLabel1”。这样，两个被选中的网络标签就被修改为相同的名称了，它们标识的两个管脚之间就建立了连接关系。图 1-8 是一个一位交互式数字信号与数字探针使用网络标签完成连接后的原理图。

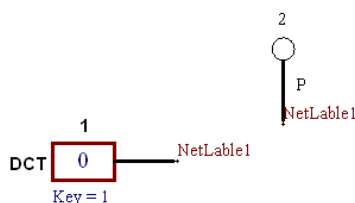


图 1-8: 使用网络标签连接器件的管脚

**注意:**所有的网络和网络标签都具有一个网络名称属性。选中网络或者网络标签后,在“属性”窗口中可以查看他们的网络名称。通常情况下,网络标签是可以用来给网络命名的。而且本质上,只要网络或者网络标签具有相同的名称,就是表示它们之间建立了连接关系。所以,可以在多个管脚末端的连接点或者网络上放置具有相同名称的网络标签来为它们建立连接关系。

- **打印原理图。**选择“文件”菜单中的“打印”,可以打印当前原理图,打印纸张等属性可以自行设置。如果当前使用的计算机没有连接打印机,可以先将原理图打印到 XPS 文件,然后将 XPS 文件放入连接了打印机的计算机中打印即可。

注意,打印前,需将全部要打印的内容移动到坐标系的第一象限内(红蓝坐标原点右上方的区域)。因为只能将坐标系第一象限的内容打印出来。

## 提交作业

如果读者是通过从 CodeCode.net 平台领取任务创建的个人项目,并将个人项目克隆到本地进行实验,实验结束后可以将本地已更改的项目再推送到 CodeCode.net 平台的个人项目中,方便教师通过 CodeCode.net 平台查看读者提交的作业。步骤如下:

1. 在左侧的“项目管理器”窗口中,右键点击项目节点,在弹出的菜单中,选择“Git”中的“推送当前分支到 Git 远程库”菜单项,弹出“推送当前分支到 Git 远程库”的对话框。
2. 在“推送当前分支到 Git 远程库”对话框中,输入本次项目更改的提示信息。
3. 点击“推送”按钮,可以将当前项目推送到 CodeCode.net 平台的个人项目中。

可以按照下面的步骤查看推送后的项目:

1. 使用浏览器访问<https://www.codecode.net>,使用已注册的帐号登录CodeCode.net平台。
2. 在“课程”列表中选择相应的课程,打开课程的详细信息页面,点击左侧导航栏的“任务”链接,打开任务列表页面。
3. 在任务列表页面打开本次实验对应的任务,在任务详情中点击“查看项目详情”按钮,可以跳转到读者的个人项目页面。
4. 在个人项目页面中,可以查看当前项目的全部内容。
5. 在项目左侧的导航栏中点击“仓库”中的“提交”链接,可以打开提交列表页面。
6. 在提交列表页面中,点击最后一次提交,可以查看此次提交发生变更的文件。

**技巧:**在 Dream Logic 的项目管理器窗口中,选中项目节点,点击鼠标右键,在弹出的右键菜单中,选择“Git”中的“使用浏览器访问 Git Origin”菜单项,可以自动打开本地项目在 CodeCode.net 平台上对应的个人项目。

## 2.2 任务(二)简单的组合逻辑电路

### 预备知识

数字电路可以分为**组合逻辑电路**和**时序逻辑电路**两大类。

组合逻辑电路的输出仅仅取决于输入的值。换句话说,它组合当前输入值来确定输出值。例如,逻辑门就是组合电路。时序电路的输出取决于当前输入值和之前的输入值。换句话说,它取决于输入的序列。组合电路是没有记忆的,但是时序电路是有记忆的。

组合逻辑电路在结构上不存在输出到输入的反馈通路,因此输出状态不影响输入状态。组合逻辑的特点是:任意时刻的输出状态取决于该时刻输入信号的状态,而与信号作用前电路状态无关。

组合逻辑分析,就是根据已知逻辑电路图,找出组合逻辑电路的输入与输出关系,确定在什么样的输入取值组合下对应的输出为 1。

组合逻辑电路是数字电路的最基本的一类,设计比较简单。一般根据设计要求,列出真值表,或用卡诺图,或用逻辑表达式进行化简,最后得到逻辑电路图并画出具体电路。有了逻辑图,在画出具体电路时,



先要根据实际条件选取器件，器件的数量和类型要尽可能少，而且要注意用摩根公式进行正负逻辑之间的转换。在实际应用中，可靠性高、成本低、维修方便是选择元器件的重要原则。

所谓的“正逻辑”，就是自变量和函数均是高电平有效，即把高电平定为逻辑“1”，把低电平定为逻辑“0”。“负逻辑”则正好相反，把高电平定为逻辑“0”，而低电平定为逻辑“1”。

### ● 奇偶校验电路

奇偶校验电路是最简单的检验方法，广泛用于计算机和数字通信中。欲传送的二进制代码称为信息码，例如  $A_1A_2A_3$ ；在传输信道中，由于干扰等原因，接收到的码可能发生了变化，例如发送的是 101，而接收到的却是 100。一般一位错的可能性远大于两位错、三位错。假定我们只考虑一位错，那么如何能发现这种错误呢？利用奇偶校验是很简便的。所谓奇偶校验，就是在信息码之外再加一位校验码  $F$ 。校验码  $F$  是在发送端产生并和其他信息码  $A_1A_2A_3$  一同发送到接收端。这样发送的代码就变成  $A_1A_2A_3F$  了，这个码通常叫做码字。接收端根据所接收到的码字，就可以判断出传输是否出错。如果有错，就要求重发。码字中 1 的个数如果是奇数，就是奇校验；如果是偶数，就叫偶校验。

当信息码  $A_1A_2A_3$  中 1 的个数为奇数时， $F=1$ ，则码字中 1 的个数就是偶数；当信息码  $A_1A_2A_3$  中 1 的个数为偶数时， $F=0$ ，码字中 1 的个数依然是偶数。这就是偶校验。

当码字  $A_1A_2A_3F$  中 1 的个数为偶数时，检错码  $E=0$ ，说明传输无误；为奇数时， $E=1$ ，说明传输中发生了错误，需要重发。

### ● “菊花链”电路

在计算机中，有一种 CPU 查询中断源的电路，叫“菊花链”。CPU 每执行完一条机器指令之后，都要查看一下有没有中断请求，如果有，就要找到中断源，然后转入相应的中断服务程序；如果没有，就继续执行下一条机器指令。“菊花链”电路的任务就是用来查询中断源是哪一个。

假定中断源有 3 个，分别是  $I_1$ 、 $I_2$ 、 $I_3$ ，优先级依次降低，低电平表示有中断；CPU 送来的查询信号为  $B$ ，也是低电平有效。图 1-9 给出了这种“菊花链”电路。当  $B$  为高电平时，无论中断源有无请求， $U_{2A}U_{2B}U_{2C}$  的 3 个输出  $I_1'$ 、 $I_2'$ 、 $I_3'$  均为 0。当  $B$  为低电平时，若  $I_1$ 、 $I_2$ 、 $I_3$  都为高电平，即无中断请求， $U_{2A}U_{2B}U_{2C}$  的 3 个输出为 0 保持不变；若  $I_1$  为低电平，不论  $I_2$ 、 $I_3$  为何值， $U_{2A}$  的输出  $I_1'$  变为 1。而由于门  $U_{1B}$  被封闭，所以，信号  $B$  的低电平就无法再向右传送，于是  $I_2'$ 、 $I_3'$  也就维持 0 不变。只有  $I_1$  为高电平时， $I_2$  的请求才能通过  $U_{2B}$  送出去。故  $I_1$  的优先级比  $I_2$  高。同样， $I_2$  的优先级比  $I_3$  高。

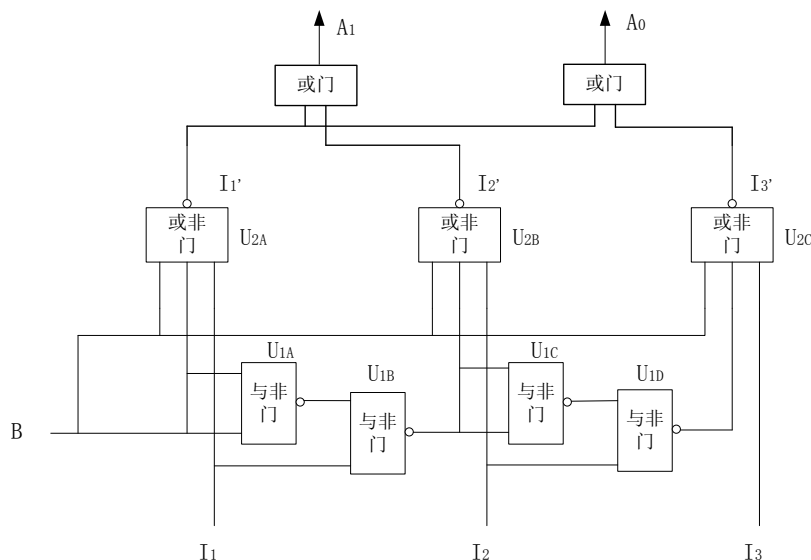


图 1-9：三中断源自动判优电路

CPU 要识别三个中断请求中优先级最高的是哪个源，还需要对以上的输出  $I_1'$ 、 $I_2'$ 、 $I_3'$  进行编码。图 1-9 中使用 2 个或门编码  $I_1'$ 、 $I_2'$ 、 $I_3'$ ，输出为  $A_1A_0$ ，注意要留一个（如  $A_1A_0=00$ ）分配给没有任何源请求的状态。如果有更多的中断源，“菊花链”还可以继续扩展下去。“菊花链”也叫串行排队电路。

在掌握了本任务的预备知识后，读者可以按照下面方法之一在本地创建一个项目，用于完成本次任务：

#### 方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建个人项目，然后使用 Dream Logic 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

#### 方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，就需要使用 Dream Logic 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的 URL

为 <https://www.codecode.net/engintime/Dream-Logic/Project-Template/Digital-Circuit/Lab001.git>

### 一位全加器

请读者在数字电路相关的教材中学习一位全加器的原理，然后按照下面的步骤查看一位全加器的原理图，并对其进行仿真验证：

1. 打开项目下的一位全加器原理图文件 Adder.dlsche。
2. 启动仿真后，根据表 1-1 给出的真值表验证全加器的功能。
3. 尝试写出输出端 S（和），以及输出端 Co（进位）的逻辑表达式，掌握一位全加器的工作原理。

输入端			输出端	
Ci	A	B	S	Co
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

表 1-1：一位全加器真值表(1 表示高电平，0 表示低电平)

### 奇校验电路

请读者在数字电路相关的教材中学习奇校验电路的原理，然后按照下面的步骤查看奇校验电路的原理图，并对其进行仿真验证。步骤如下：

1. 在之前验证一位全加器时克隆的项目中找到原理图文件 oddcheck.dlsche，打开此文件就可以看到奇校验电路的原理图。
2. 启动仿真，并将奇校验电路真值表 1-2 填写完整。验证奇校验电路的功能时，假设接收端总是接收到正确的校验码 F。仿真结束后，尝试写出校验码 F 和检错码 E 的逻辑表达式。并说明奇校验电路的局限性。

发送端				接收端			检错码
A1	A2	A3	F	B1	B2	B3	E
0	0	0		0	0	0	
0	1	0		0	1	0	
1	1	0		1	1	0	
0	1	0		0	0	0	
1	0	0		1	1	0	
1	0	1		1	1	0	

表 1-2: 奇校验真值表

3. 修改原理图文件 oddcheck.dlsche, 将此奇校验电路扩展为可用于校验 4 位数据 A1~A4 的电路。并通过仿真确保电路能够正常工作。
4. 提交作业。

### 2.3 任务（三）设计三中断源自动判优电路

请读者按照下面方法之一在本地创建一个项目，用于完成本次任务：

#### 方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建个人项目，然后使用 Dream Logic 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

#### 方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，就需要使用 Dream Logic 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的 URL

为 <https://www.codecode.net/engintime/Dream-Logic/Project-Template/blank.git>

请读者按照下面的步骤完成本次任务：

1. 参考图 1-9，在原理图文件 top.dlsche 中设计一个三中断源自动判优电路。需要用到的逻辑门可以在型号库“逻辑门”中找到。
2. 使用型号库“数字信号源”中的一位交互式数字信号控制输入信号 B、I<sub>1</sub>、I<sub>2</sub>、I<sub>3</sub>。使用型号库“数字信号显示”中的数字探针连接输出信号 A<sub>0</sub>和 A<sub>1</sub>，用于表示输出结果。
3. 启动仿真后，根据表 1-3 给出的真值表测试电路，确保电路准确无误。
4. 提交作业。

查询信号	中断源			输出	
B	I <sub>1</sub>	I <sub>2</sub>	I <sub>3</sub>	A <sub>0</sub>	A <sub>1</sub>
1	X	X	X	0	0
0	0	0	0	1	1
0	0	0	1	1	1
0	0	1	0	1	1
0	0	1	1	1	1
0	1	0	0	1	0
0	1	0	1	1	0
0	1	1	0	0	1
0	1	1	1	0	0

表 1-3: 三中断源自动判优电路真值表(1 表示高电平, 0 表示低电平, X 表示任意值)

### 2.4 任务（四）设计偶校验电路

请读者按照下面方法之一在本地创建一个项目，用于完成本次任务：

#### 方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建个人项目，然后使用 Dream Logic 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

#### 方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，就需要使用 Dream Logic 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的 URL

为 <https://www.codecode.net/engintime/Dream-Logic/Project-Template/blank.git>

请读者按照下面的步骤完成本次任务：

1. 参考本实验中的奇校验电路，在原理图文件 top.dlsche 中设计一个偶校验电路。假设接收端总

是能接收到正确的校验码 F。当检错码 E=0 时，说明接收的数据无误。尝试写出校验码 F 和检错码 E 的逻辑表达式。

- 参考表 1-2 设计偶校验真值表，要求至少包含五行数据，并且包含校验成功和校验失败的情况。
- 提交作业。

## 2.5 任务（五）设计一位全减器

请读者按照下面方法之一在本地创建一个项目，用于完成本次任务：

### 方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建个人项目，然后使用 Dream Logic 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

### 方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，就需要使用 Dream Logic 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的 URL

为 <https://www.codecode.net/engintime/Dream-Logic/Project-Template/blank.git>

请读者按照下面的步骤完成本次任务：

- 参考本实验中的一位全加器，在原理图文件 top.dlsche 中设计一个一位全减器。当输入信号  $C_i=1$  时表示有借位输入，当输出信号  $C_o=1$  时需要向高位借位。输入信号 A 是被减数，B 是减数，输出信号 S 是差。S 和  $C_o$  可以接一个 16 进制 7 段数码管，该器件在型号库“数字信号显示”中。
- 使用一位全减器真值表 1-4 测试电路，确保电路准确无误。
- 提交作业

输入端			输出端	
$C_i$	A	B	S	$C_o$
0	0	0	0	0
0	0	1	1	1
0	1	0	1	0
0	1	1	0	0
1	0	0	1	1
1	0	1	0	1
1	1	0	0	0
1	1	1	1	1

表 1-4：一位全减器真值表

## 三、思考与练习

- 有一个水箱由大、小两台水泵 ML 和 MS 供水，如图 1-10 所示。水箱中设置了三个水位检测元件 A、B、C。水面低于检测元件时，检测元件给出高电平；水面高于检测元件时，检测元件给出低电平。现要求当水位超过 C 点时水泵停止工作；水位低于 C 点且高于 B 点时 MS 单独工作；水位低于 B 点而高于 A 点时 ML 单独工作；水位低于 A 点时 ML 和 MS 同时工作。试用门电路设计一个控制两台水泵的逻辑电路，要求尽量简单。

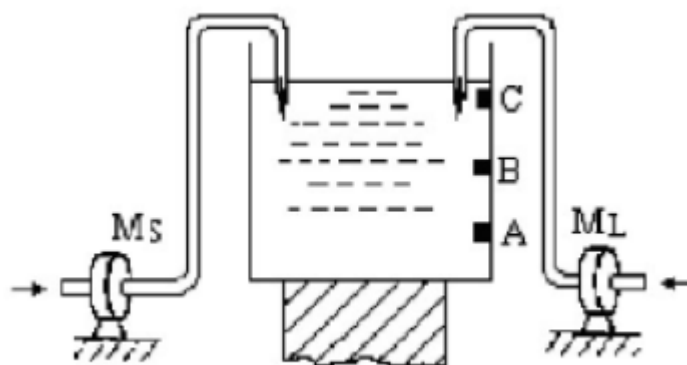


图 1-10：水箱供水

- 设计一交通灯检测电路。红、绿、黄三只灯正常工作时只能一只灯亮，否则，将会发出交通灯故障信号。
- 人类有四种基本血型 A、B、AB、O 型，输血者和受血者的血型必须符合一定的原则（如图 1-11），尝试设计一个检验输血者与受血者血型是否符合的电路。

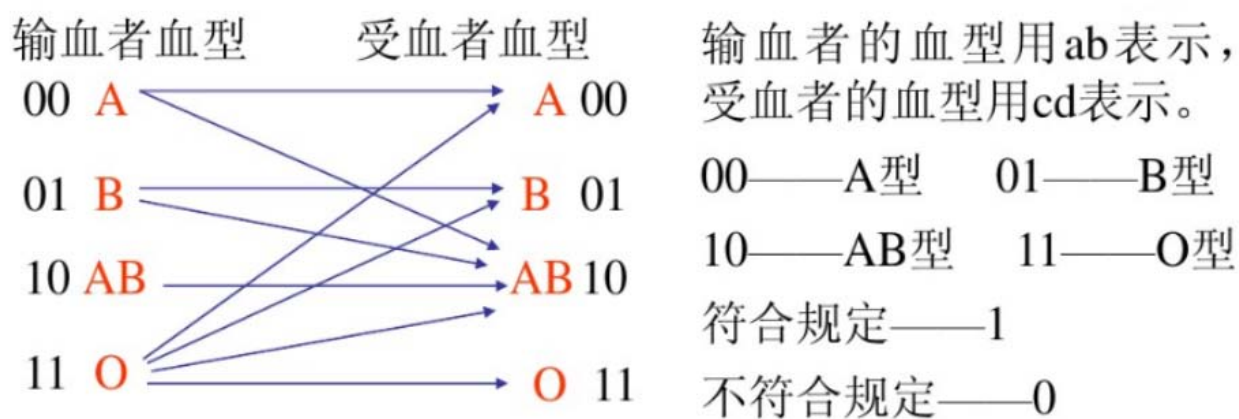


图 1-11：血型匹配规则

## 实验 2 编码器和译码器

实验性质：验证+设计

建议学时：2 学时

### 一、实验目的

- 熟悉编码器的功能与基本应用。
- 熟悉译码器的功能与基本应用。
- 熟练使用编码器和译码器设计功能电路。

### 二、预备知识

#### 2.1 编码和译码

编码是把一组有特定含义（事件）的输入信号按一定的规律编成不同二进制代码输出的过程。一个事件对应唯一一组特定的编码，而一组特定的编码表示一个事件。用来完成编码工作的电路称为编码器。

译码是编码的反过程，就是把一组包含特定信息的二进制码翻译成一组高低电平信号，其中只有一个输出呈现有效状态。实现译码功能的组合逻辑电路称为译码器。它是数字系统中最常用的逻辑构件之一。常用的译码器有：2 线-4 线译码器，3 线-8 线译码器，由它们可以构成 4 线-16 线译码器、4 线-10 线译码器等。

#### 2.2 多外设共享地址总线排队电路

计算机中的 CPU、各外设都可以访问存储器。但它们的优先级别不同，这应该事先规定好。可以将 CPU、各外设一律称作设备，假定设备 1 的优先级最高，依次是设备 2、设备 3……所谓“访问”，就是设备和存储器交换（读或写）数据的过程。该过程是这样进行的：某设备要访问存储器，首先要发出一个占用总线的请求信号，若没有比它优先级更高的设备也在请求的话，它的请求便得以响应。这时它发出的地址码就能送到地址总线上，存储器使用地址总线上地址码，确定要访问的单元（存放数据的地方），其中的数据就通过数据总线和该设备进行交换。现在我们只讨论从设备发出请求信号到把地址码送到地址总线这一段的工作原理。

为了和芯片的功能配合，假定设备要占用总线的请求为 0，不请求是 1；把各设备的请求信号加到优先编码器的输入端，进行优先编码，它输出的每一个有效码，既对应着一个一个的设备，又表示现在在申请占用总线的设备中，它的优先级最高（注意，一定要留一个无效码，表示没有任何设备申请）。然后，通过译码，让其输出打开一组传输门。所对应的设备通过该组门，把地址码送往地址总线（如图 2-1）。

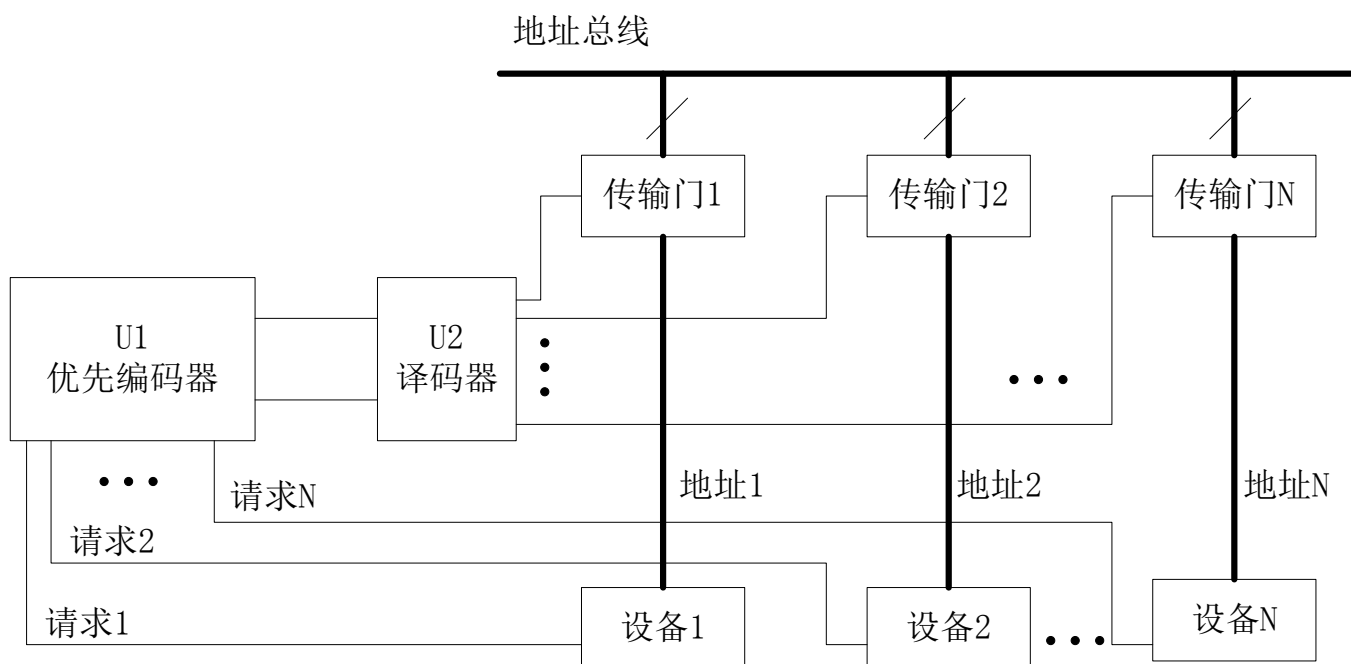


图 2-1: 多外设共享地址总线排队电路原理图

### 三、实验内容

#### 3.1 任务（一）设计普通 BCD 编码器

请读者按照下面方法之一在本地创建一个项目，用于完成本次任务：

##### 方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建个人项目，然后使用 Dream Logic 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

##### 方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，就需要使用 Dream Logic 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的 URL

为 <https://www.codecode.net/engintime/Dream-Logic/Project-Template/blank.git>

图 2-2 所示为一个普通 BCD 编码器的结构框图和逻辑电路图。普通 BCD 编码器有十个输入信号  $I_0 \sim I_9$ ，每个输入信号连接一个代表十进制数（0 到 9）的信号，任意一个时刻只能有一个输入信号是高电平。其中输入信号  $I_0$  为高电平时，表示输入 0； $I_9$  为高电平时，表示输入 9。有四个输出信号  $D_0 \sim D_3$ ，它们组成一个 BCD 码用于表示编码后的十进制数，其中  $D_3$  为高位， $D_0$  为低位。

请读者按照下面的步骤完成本次任务：

1. 参考图 2-2，在原理图文件 top.dlsche 中设计一个普通 BCD 编码器。
2. 对编码器进行仿真，填写普通 BCD 编码器的真值表 2-3。注意确认 BCD 码与十进制数之间的关系。

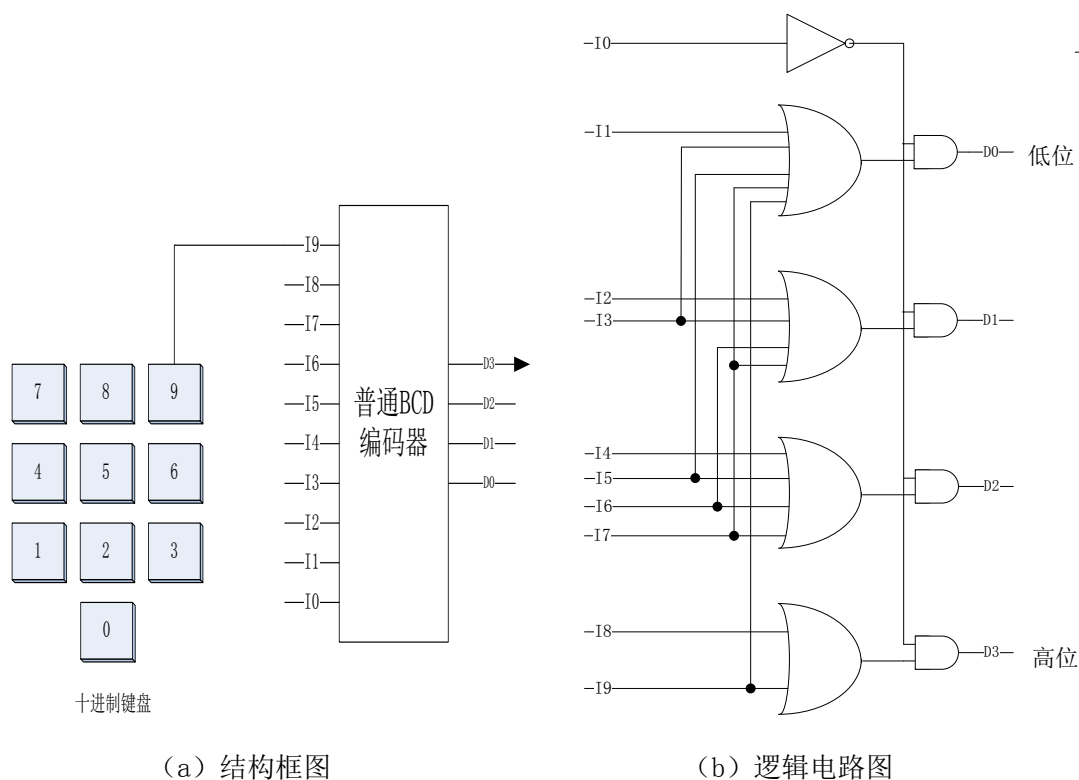


图 2-2：普通 BCD 编码器

输入	输出 BCD 码			
十进制数字信号	D3	D2	D1	D0
I0=1	0	0	0	0
I1=1	0	0	0	1
I2=1				
I3=1				
I4=1				
I5=1				
I6=1				
I7=1				
I8=1				
I9=1				

表 2-3：普通 BCD 编码器真值表

### 3.2 任务（二）常用编码器和译码器

请读者按照下面方法之一在本地创建一个项目，用于完成本次任务：

#### 方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建个人项目，然后使用 Dream Logic 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

#### 方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，就需要使用 Dream Logic 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的 URL

为 <https://www.codecode.net/engintime/Dream-Logic/Project-Template/Digital-Circuit/Lab002.git>



### 8-3 优先编码器 74LS148

完成任务（一）后，读者可以了解到普通编码器对输入信号是有限制的，即在任意一个时刻所有输入信号中只允许一个输入信号有效，否则编码器将发生混乱。为了解决这一问题，可采用**优先编码器**，它允许多个输入信号同时有效，但是只对其中优先级最高的输入信号产生编码，而忽略掉优先级较低的输入信号。

请读者按照下面的步骤验证 8-3 优先编码器 74LS148 的功能：

1. 打开项目下的 74LS148.dlsche 文件。该原理图文件中绘制了 74LS148 的内部原理图。其输入信号包括级联入（使能）EI，低电平有效，还有 8 个输入信号 D0~D7，也是低电平有效，D0 优先级最低，D7 优先级最高。输出信号包括编码 A0~A2，以及级联出 E0 和 GS。
2. 启动仿真。根据 74LS148 的功能表 2-4，验证其功能。并将功能表补充完整。需要说明的是，在此原理图中使用了“八位交互式数字信号”控制 D0~D7 的输入，读者可以选中此器件，然后在右侧的属性窗口中编辑此器件的“数字信号”属性，在其中填入一个两位的十六进制数后按回车，就可以设定该器件输出的信号了。
3. 观察填写完成的功能表，当输入信号有效时（D7~D0 中至少有一位输入为 0），输出信号 A2~A0 的值与输入信号有何关系？根据输出信号的值能确定当前是哪一位最高优先级的输入信号有效吗？

级联入（使能）	输入	输出	级联出
EI	D7 D6 D5 D4 D3 D2 D1 D0	A2A1A0	E0 GS
1	X X X X X X X X	1 1 1	1 1
0	1 1 1 1 1 1 1 0	1 1 1	1 0
0	1 1 1 1 1 1 0 X	1 1 0	1 0
0	1 1 1 1 1 0 X X	1 0 1	1 0
0	1 1 1 1 0 X X X	1 0 0	1 0
0	1 1 1 0 X X X X		
0	1 1 0 X X X X X		
0	1 0 X X X X X X		
0	0 X X X X X X X		
0	1 1 1 1 1 1 1 1		

表 2-4：74LS148 功能表

### 3-8 译码器 74LS138

3-8 译码器 74LS138 有 3 个输入信号 A、B、C（C 表示最高位，A 表示最低位），8 个输出信号 Y0~Y7，另有 3 个使能输入信号 G1、G2A 和 G2B。由 3-8 译码器的功能表 2-5 可以看出，译码器根据 C，B，A 三个输入信号和 G1，G2A，G2B 三个使能输入信号的条件，决定哪个输出信号有效（低电平有效）。也就是说，在作为译码器使用时，当 Y0~Y7 中的某一位输出低电平时，对应的一组输入信号被译码。

请读者按照下面的步骤验证 3-8 译码器 74LS138 的功能：

1. 打开项目下的 74LS138.dlsche。该原理图文件中绘制了 74LS138 的内部原理图。
2. 启动仿真。根据 74LS138 的功能表 2-5，验证其功能。并将功能表补充完整。

门控		输入	输出
G1	G2A+G2B	C B A	Y7 Y6 Y5 Y4 Y3 Y2 Y1 Y0
X	1	X X X	1 1 1 1 1 1 1 1
0	X	X X X	1 1 1 1 1 1 1 1
1	0	0 0 0	1 1 1 1 1 1 1 0
1	0	0 0 1	1 1 1 1 1 1 0 1
1	0	0 1 0	1 1 1 1 1 0 1 1

1	0	0 1 1	1 1 1 1 0 1 1 1
1	0		
1	0		
1	0		
1	0		

表 2-5: 74LS138 功能表 (G2A+G2B 中的加号表示逻辑或)

3. 观察填写完成的功能表, 3-8 译码器的三个使能输入端输入何种信号时, 译码器才能正常工作。尝试写出输出信号 Y0、Y3 和 Y7 的逻辑表达式。

### 七段数码管显示译码器 74LS48

在设计一个数字系统时, 常常需要使用数码显示器件, 将测量或运算结果以数字方式显示出来, 以便监视系统的工作状况。请读者按照下面的步骤进行试验, 测试 74LS48 七段数码管显示译码器的功能:

1. 打开项目下的原理图 74LS48.dlsche。该原理图文件中使用 74LS48 设计了一个用于将输入的 4 位二进制信号显示为十进制数字的电路。
2. 启动仿真。根据 74LS48 的功能表 2-6, 验证其功能。并将功能表补充完整。

输入					输出					显示数字			
LT	RBI	D	C	B	A	BI\RBO	a	bc	d		e	f	g
1	1	0000				1	1111110						0
1	X	0001				1	0110000						1
1	X	0010				1	1101101						2
1	X	0011				1	1111001						3
1	X	0100				1	0110011						4
1	X	0101				1							
1	X	0110				1							
1	X	0111				1							
1	X	1000				1							
1	X	1001				1							
X	X	X	X	X	X	0	0000000						
0	X	X	X	X	X	1	1111111						

表 2-6: 74LS48 功能表

3. 将原理图中的器件 74LS48 替换为 74LS47 (在型号库窗口的 74LS 功能芯片型号库中可以找到 74LS47), 并在各个输出端加上反相器 (非门)。仿真测试, 说明 74LS47 与 74LS48 的有何不同?

### 三态门

由于在后面的实验任务中会用到三态门, 而且三态门也是一种非常重要的器件, 所以需要读者按下列步骤熟悉三态门的工作原理和使用方法:

1. 打开项目下的原理图 tristate.dlsche。该原理图文件中将两个输入信号分别通过一个三态门连接到了同一条总线上, 这样就可以通过控制三态门的状态来决定是将哪个信号输出到总线上, 从而避免总线上的信号发生冲突。这里的三态门是低电平使能, 也就是说当控制端信号为低电平时, 三态门打开。
2. 启动仿真。根据表 2-7, 验证三态门的工作原理, 并将表格补充完整。开始仿真后, 当总线为蓝色时, 说明总线上的信号发生了冲突, 是一个不确定的值, 在一个实际可工作的数字系统中是绝对不允许出现这种情况的。
3. 修改原理图, 将总线上连接的信号数量从两个扩展为四个。首先, 在左侧型号库窗口的“逻辑门”型号库中找到低电平使能三态门的型号 LTri\_Gate, 使用此型号在原理图文件 tristate.dlsche 中再添加两个三态门, 然后将原有的两个输入信号扩展为四个输入信号, 最后完成仿真验证。
4. 提交作业。

通过上述实验不难得出，当多个三态门同时向总线输出数据时，为避免发生数据冲突，同一时刻必须只能有一个三态门打开（通过控制端的低电平将其使能），而其他三态门关闭。这样，就能确保同一个时刻总线只与其中一个三态门导通。

输入信号		控制信号		输出信号
I1	I2	C1	C2	总线
1	0	0	0	冲突
0	0	1	0	0
0	1	1	0	1
1	0	1	0	0
1	1	1	0	1
0	0	0	1	
0	1	0	1	
1	0	0	1	
1	1	0	1	

表 2-7：三态门测试

### 3.3 任务（三）设计多外设共享地址总线排队电路

请读者按照下面方法之一在本地创建一个项目，用于完成本次任务：

#### 方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建个人项目，然后使用 Dream Logic 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

#### 方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，就需要使用 Dream Logic 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的 URL

为 <https://www.codecode.net/engintime/Dream-Logic/Project-Template/blank.git>

根据本实验预备知识中介绍的多外设共享地址总线排队电路，并参考图 2-1，在原理图文件 top.dlsche 中设计一个多外设共享地址总线排队电路，并完成仿真验证。要求在电路中模拟 0、1、2、3 共 4 个设备，优先级逐次降低，0 号设备的请求优先级最高，4 号设备的请求优先级最低。设备地址码为 2 位，设备地址依次为 0, 1, 2, 3，所以地址总线位宽为 2。

既然是模拟这些设备，就不需要真的设计出来这些设备，只需要让这些设备能够产生总线请求信号，可以产生地址信号并发送到地址总线即可。读者可以参考图 2-3 设计总线请求信号，然后参考图 2-4 设计设备地址信号与地址总线的连接关系，最后将编码器和译码器连接好就可以了。

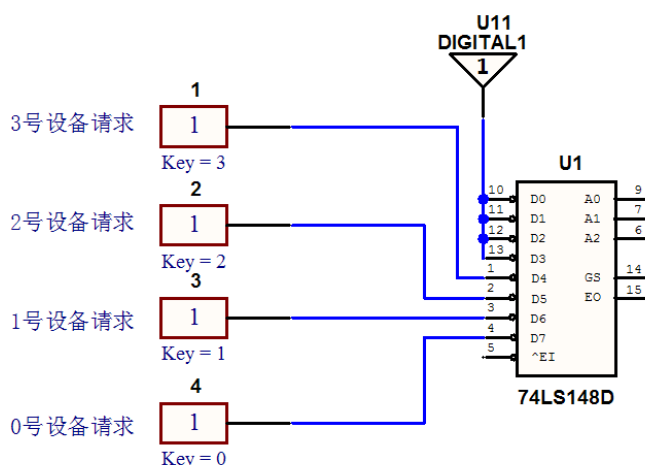


图 2-3: 设备的总线请求信号与编码器的连接方法

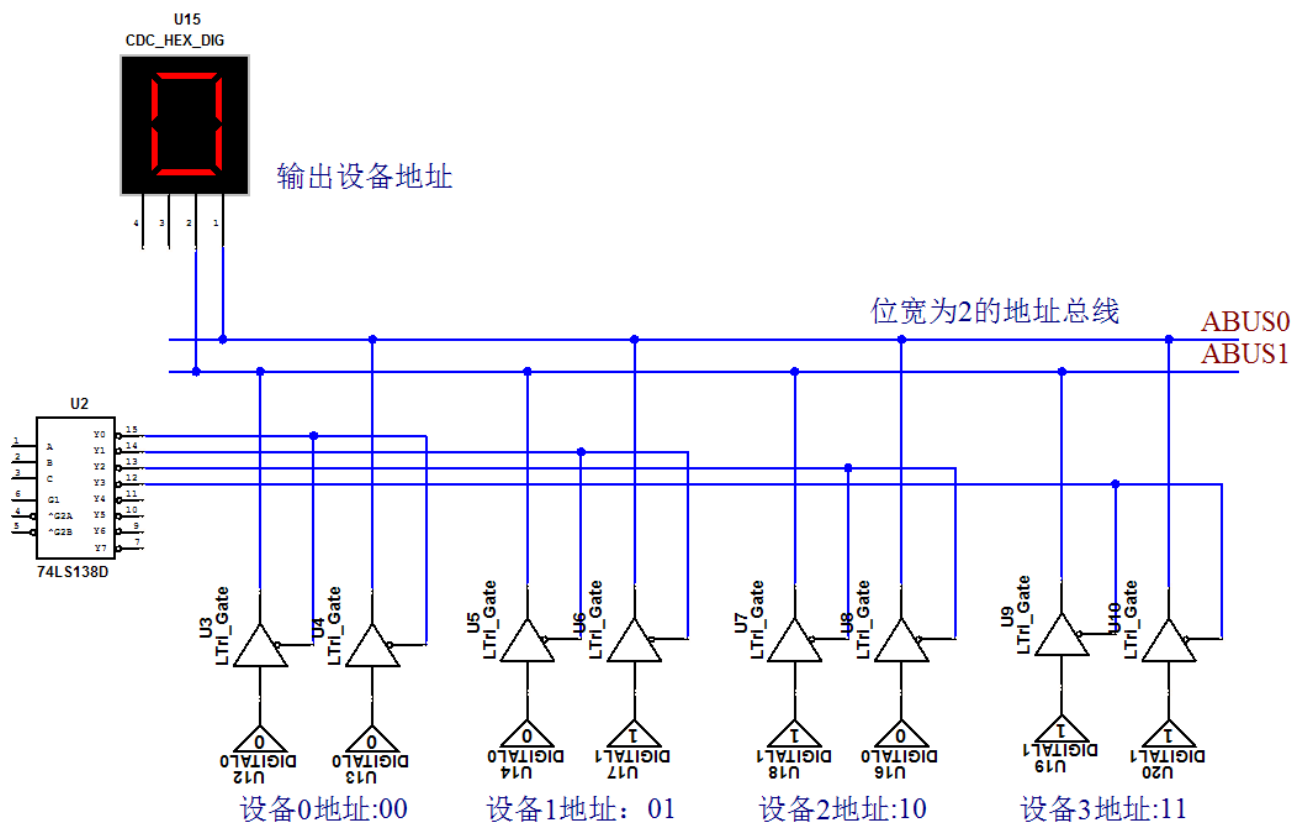


图 2-4: 译码器控制三态门将指定的设备地址发送到地址总线

## 四、思考与练习

1. 仿照 74LS148 的电路原理图，设计一个 4 位优先编码器并成功功能验证。
2. 仿照 74LS138 的电路原理图，设计一个 2-4 译码器并成功功能验证。
3. 使用两片 74LS138 设计一个 4-16 译码器。
4. 在什么条件下，编码器、译码器必须留一个无效码？什么时候可以不留？
5. 设计一个设备工作状态指示器电路，要求用红、黄、绿三个指示灯表示三台设备的工作情况：绿灯亮表示全部正常；红灯亮表示有一台不正常；黄灯亮表示两台不正常；红、黄灯全亮表示三台都不正常。
6. 用两片 74LS151 数据选择器设计实现一位全加器，实现电路并仿真验证其正确性。提示：可将全加器的三位输入 A, B, Ci 作为 8 选 1 数据选择器 74LS151 的选择信号，需要两个 8 选 1 数据选择器，分别输出和 S 以及高位进位信号 Co。
7. 用 74LS138 实现一位全减器并仿真验证其正确性。
8. 在计算机系统中通常有多个优先级不同的中断请求，当多个中断源都向 CPU 发出中断请求时，需要选出优先级最高的中断请求，进行优先服务，这时可通过一个优先编码器译出最高中断级别，根据级别就可以确定调用哪个中断服务程序。假设有 0~15 共 16 个中断源，0 号中断请求优先级为 0，优先级最高，15 号中断请求优先级为 15，优先级最低。请设计电路，根据当前中断请求，输出对应的优先级。**提示：**使用两片 74LS148 优先编码器和一片 74LS158 数据选择器设计一个扩展的 16-4 优先编码器，16 个输入端接各个中断请求，4 位输出二进制数就是当前最高优先级中断的级别。

## 实验 3 加法器

实验性质：验证+设计

建议学时：2 学时

### 一、实验目的

- 了解常用加法器的设计方法。
- 掌握四位超前进位并行加法器 74LS283。
- 掌握算术逻辑运算器（ALU）74LS181。

### 二、预备知识

加法器是计算机或其他数字系统中对二进制数进行运算处理的组合逻辑器件。按进位信号产生的方法不同，可分为串行进位加法器和并行进位加法器两种。

#### 2.1 串行进位加法器

串行进位加法器逻辑结构如图 3-1 所示，它由多个一位全加器串行连接而成。每个一位全加器有三个输入（加数 A，被加数 B，由低位输入的进位信号 CI）和两个输出（和数 S，向高位的进位信号 CO），然后按照图 3-1 所示进行串行连接。

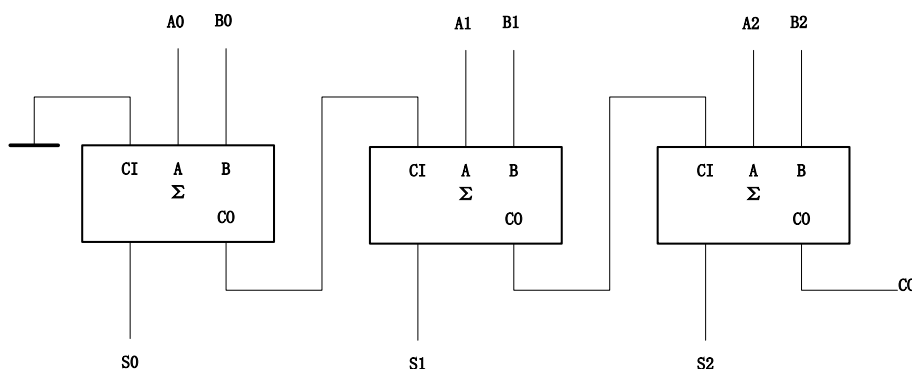


图 3-1：串行进位加法器

#### 2.2 超前进位并行加法器

串行进位加法器须将低位全加器产生的进位信号逐位向高一位全加器上传递。因此，串行进位加法器求和的最高位的输出结果必须等到各位进位信号逐位传递后才能形成，工作速度很慢。为了提高工作速度，可采用四位超前进位并行加法器，其原理如下：

设有两个四位二进制加数  $A=A_4A_3A_2A_1$ ， $B=B_4B_3B_2B_1$ 。并行进位加法器的 S（和）以及 C（进位）的逻辑表达式如下：

$$S_1=A_1 \oplus B_1 \oplus C_0, \quad C_1=A_1B_1 + (A_1 \oplus B_1)C_0$$

$$S_2=A_2 \oplus B_2 \oplus C_1, \quad C_2=A_2B_2 + (A_2 \oplus B_2)C_1$$

$$S_3=A_3 \oplus B_3 \oplus C_2, \quad C_3=A_3B_3 + (A_3 \oplus B_3)C_2$$

$$S_4=A_4 \oplus B_4 \oplus C_3, \quad C_4=A_4B_4 + (A_4 \oplus B_4)C_3$$

设  $G_i$  为进位生成项， $P_i$  为进位传递项，即有：

$$G_i=A_iB_i \cdots \cdots \text{表达式 1}$$

$$P_i=A_i \oplus B_i \cdots \cdots \text{表达式 2}$$

则各位和的表达式变为

$$S_i=P_i \oplus C_{i-1} \cdots \cdots \text{表达式 3}$$

各进位表达式变为

$$C_i = G_i + P_i C_{i-1} \dots \dots \dots \text{表达式 4}$$

采用递推公式，进位表达式变为

$$C_1 = G_1 + P_1 C_0 \dots \dots \dots \text{表达式 5}$$

$$C_2 = G_2 + P_2 C_1 = G_2 + P_2 G_1 + P_2 P_1 C_0 \dots \dots \dots \text{表达式 6}$$

$$C_3 = G_3 + P_3 C_2 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 C_0 \dots \dots \dots \text{表达式 7}$$

$$C_4 = G_4 + P_4 C_3 = G_4 + P_4 G_3 + P_4 P_3 G_2 + P_4 P_3 P_2 G_1 + P_4 P_3 P_2 P_1 C_0 \dots \dots \dots \text{表达式 8}$$

最终的进位表达式 5~8 表明，最低位的进位信号  $C_0$  可以超前传送到  $C_2$ 、 $C_3$ 、 $C_4$ 。将这些进位表达式 5~8 带入上面的各位和的表达式 3 中（请读者自行完成表达式的推导），则可以看到最低位的进位信号  $C_0$  也可以超前传送到  $S_2$ 、 $S_3$ 、 $S_4$  上，从而大大加快了进位信号的传递速度，进而加快运算速度。

并行进位运算虽然能显著加快运算速度，但同时也带来了运算电路器件数目增加的问题。在数字系统设计过程中，需要根据设计的目的来权衡运算速度和电路面积这一对矛盾。

### 三、实验任务

#### 3.1 任务（一）了解常用的加法器

请读者按照下面方法之一在本地创建一个项目，用于完成本次任务：

##### 方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建个人项目，然后使用 Dream Logic 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

##### 方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，就需要使用 Dream Logic 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的 URL

为 <https://www.codecode.net/engintime/Dream-Logic/Project-Template/Digital-Circuit/Lab003.git>

#### 四位超前进位并行加法器 74LS283

请读者按照下面的步骤验证四位超前进位并行加法器 74LS283 的功能：

1. 打开项目下的原理图文件 74LS283.dlsche。该原理图文件中绘制了 74LS283 的内部原理图。其输入信号包括两个加数  $A_1 \sim A_4$ 、 $B_1 \sim B_4$ ，以及进位输入  $C_0$ 。输出信号包括和  $S_1 \sim S_4$ ，最高进位输出  $C_4$ 。
2. 启动仿真。根据 74LS283 的功能表 3-1，验证其加法运算功能。并将表格补充完整。注意表格中使用的是十六进制数。
3. 在 74LS283 中使用一个红色矩形框将所有用于计算  $S_1$  的逻辑门包含在其中。并写出通过这些逻辑门计算  $S_1$  的表达式，然后证明这个表达式与本实验 2.2 节中推导的  $S_1$  的表达式（将表达式 2 带入表达式 3 可得到）等效。绘制红色矩形框的方法是，选择“绘制”菜单中的“矩形”，然后在原理图中将所有用于计算  $S_1$  的逻辑门包含在其中，绘制完毕后，选中矩形框，在右侧的“属性”视图中，将矩形框的“填充”属性修改为“否”，将轮廓的“颜色”修改为红色。
4. 请读者再尝试证明在 74LS283 中使用逻辑门计算  $S_2$  的表达式与本实验 2.2 节中推导的  $S_2$  的表达式是等效的。

输入信号			输出信号	
四位加数 A	四位加数 B	进位输入 $C_0$	四位和 S	进位输出 $C_4$
3	5	0	8	0
3	5	1	9	0
7	9	0		
7	9	1		
F	F	0		
F	F	1		

表 3-1: 74LS283 功能验证表

## 算术逻辑运算单元 74LS181

算术逻辑运算单元 (ALU) 74LS181 是一种功能较强的组合逻辑电路, 它能进行多种算术运算和逻辑运算。它的基本逻辑结构是超前进位加法器。

$S_3 \sim S_0$ : 工作方式选择。

$M$ : 当  $M=1$  时, 进行逻辑运算;  $M=0$  时, 进行算术运算。

$A=A_3 \sim A_0$ ,  $B=B_3 \sim B_0$ : 参加运算的两个数 (注脚 3 表示最高位)。

$F_3 \sim F_0$ : 运算结果。

$CN$ : 当做加运算时, 表示最低位进位输入,  $CN=1$  时, 无进位输入;  $CN=0$  时, 有进位输入。当做减法时, 表示最低位借位,  $CN=1$ , 有借位;  $CN=0$ , 无借位。

$CN4$ : 最高位进位输出, 低电平有效。

$AEQB$ : 当  $F_3 \sim F_0$  全为高电平时为 1, 否则为 0。

$G$  称为进位发生输出,  $P$  称为进位传送输出。它们是为了便于实现多芯片 ALU 之间的超前进位的。

方式	$M=1$ 逻辑运算	$M=0$ 算术运算	
$S_3 S_2 S_1 S_0$	逻辑运算	$CN=1$ (无进位)	$CN=0$ (有进位)
0 0 0 0	$F=A$ (非)	$F=A$ (直传)	$F=A$ 加 1
0 0 0 1	$F=/(A+B)$	$F=A+B$	$F=(A+B)$ 加 1
0 0 1 0	$F=(/A)B$	$F=A+/B$	$F=(A+/B)$ 加 1
0 0 1 1	$F=0$	$F=$ 负 1	$F=0$
0 1 0 0	$F=/(AB)$	$F=A$ 加 $A(/B)$	$F=A$ 加 $A/B$ 加 1
0 1 0 1	$F=/B$	$F=(A+B)$ 加 $A/B$	$F=(A+B)$ 加 $A/B$ 加 1
0 1 1 0	$F=A \oplus B$ (异或)	$F=A$ 减 $B$ 减 1	$F=A$ 减 $B$
0 1 1 1	$F=A/B$	$F=A(/B)$ 减 1	$F=A(/B)$
1 0 0 0	$F=/A+B$	$F=A$ 加 $AB$	$F=A$ 加 $AB$ 加 1
1 0 0 1	$F=/(A \oplus B)$	$F=A$ 加 $B$	$F=A$ 加 $B$ 加 1
1 0 1 0	$F=B$	$F=(A+/B)$ 加 $AB$	$F=(A+/B)$ 加 $AB$ 加 1
1 0 1 1	$F=AB$ (与)	$F=AB$ 减 1	$F=AB$
1 1 0 0	$F=1$	$F=A$ 加 $A$	$F=A$ 加 $A$ 加 1
1 1 0 1	$F=A+/B$	$F=(A+B)$ 加 $A$	$F=(A+B)$ 加 $A$ 加 1
1 1 1 0	$F=A+B$ (或)	$F=(A+/B)$ 加 $A$	$F=(A+/B)$ 加 $A$ 加 1
1 1 1 1	$F=A$ (直传)	$F=A$ 减 1	$F=A$ (直传)

表 3-2: 74LS181 真值表

**提示:** 1—高电平, 0—低电平。中文的“加”和“减”表示算数运算。其他的符号表示逻辑运算, 例如“/”表示逻辑取反, “+”表示逻辑或。表中的  $A$  和  $B$  分别表示 4 位二进制数  $A_3A_2A_1A_0$ 、 $B_3B_2B_1B_0$ 。

请读者按照下面的步骤验证算术逻辑运算单元 74LS181 的功能:

1. 打开项目下的原理图文件 74LS181.dlsche。该原理图文件中绘制了 74LS181 的内部原理图。
2. 启动仿真, 根据表 3-2 对 74LS181 进行功能验证, 由于 74LS181 的运算功能很多, 可以选择一部分功能进行验证, 请读者自行设计一个功能验证记录表, 至少要验证表 3-2 中有阴影的功能项, 并记录所有的输入输出信号。

## 3.2 任务(二) 加法器与译码器综合设计

请读者按照下面方法之一在本地创建一个项目, 用于完成本次任务:

## 方法一: 从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务, 从而在 CodeCode.net 平台上创建个人项目, 然后使用 Dream Logic 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

### 方法二：不从 CodeCode.net 平台领取任务

如果读者不使用CodeCode.net平台，就需要使用Dream Logic提供的“从Git远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的URL

为 <https://www.codecode.net/engintime/Dream-Logic/Project-Template/blank.git>

本任务要求读者在原理图文件 top.dlsche 中使用一片 74LS181 设计一个简单的运算单元（ALU），使该 ALU 在 3 位输入信号  $X_0 \sim X_2$  控制下可以完成 8 种不同的运算，并通过仿真进行功能验证。具体要求如下：

通过 6 个输入信号 S3、S2、S1、S0、M、CN 可以控制 74LS181 完成 48 种不同的运算。假设一个使用 74LS181 做运算器的 4 位处理器，它的指令编码中只有 3 位可用于对运算方式进行编码（支持  $2^3=8$  种不同运算）。请读者设计一个译码电路，该译码电路可以接收 3 位运算方式编码，然后输出 6 个控制信号，使用这 6 个控制信号控制 74LS181 完成 8 种不同的运算。

假设这 3 位编码分别为  $X_2$ 、 $X_1$ 、 $X_0$ ，译码输出最小项的组合控制信号分别为 S3、S2、S1、S0、M、CN。 $X_2X_1X_0$  的值可以为  $0 \sim 7$ ，分别对应表 3-3 中的 8 种不同的运算。

**提示：**读者应首先根据表 3-3 设计一个真值表，该真值表的输入信号是  $X_0 \sim X_2$ ，输出信号是 74LS181 的 6 个控制信号，然后根据真值表设计译码电路。译码电路可以使用一片 74LS138 将输入信号译码后，再单独设计一个编码电路，通过基本的逻辑门将 8 个信号编码为 6 个控制信号；也可以使用基本的逻辑门将 3 个输入信号直接译码为 6 个控制信号。

$X_2$	$X_1$	$X_0$	74LS181 功能
0	0	0	F=A 加 B
0	0	1	F=A 加 B 加 1
0	1	0	F=A 减 B
0	1	1	F=A 减 B 减 1
1	0	0	F=A+B（或）
1	0	1	F=AB（与）
1	1	0	F=A⊕B（异或）
1	1	1	F=¬A（非）

表 3-3：控制信号与 74LS181 运算方式对应表

## 四、思考与练习

1. 使用两片 74LS181 设计一个 8 位的 ALU 运算器。
2. 设计一个 3 位的串行进位加法器。
3. 设计一个 3 位的超前进位加法器。
4. 用 74LS181 设计一个 1 位十六进制数转换为 2 位十进制数 8421BCD 码的电路。**提示：**当十六进制数  $\leq 9$  时，直接传送（即加 0）；当十六进制数  $\geq 10$  时，加 6 便是十进制数，进位输出就是十位。这里可用直接转换法，例如十六进制数由 74LS181 的 A 端输入，同时对 A 进行判断，判断电路的输出接到 B2、B1 上，并令 B3B0=00；当  $A \leq 9$  时，判断电路输出 B2B1=00；当  $A \geq 10$  时，判断电路输出 B2B1=11。74LS181 的输出（连同进位输出）就是变换结果。



# 实验 4 触发器

实验性质：验证+设计

建议学时：2 学时

## 一、实验目的

- 掌握各类锁存器的工作方式。
- 掌握各类触发器的触发方式、逻辑功能及原理。
- 学会运用触发器设计基本时序电路。

## 二、预备知识

### 2.1 时序逻辑电路

时序逻辑电路的输出往往反馈到输入端，与输入变量一起决定电路的输出。时序逻辑电路的特点是：任意时刻输出不仅取决于该时刻的输入值，而且还与原来的状态有关。因此时序逻辑电路具有记忆功能，这是它与组合逻辑电路的本质区别。

### 2.2 锁存器的基本特性

锁存器在电路上具有两个稳定的物理状态，所以它们能记忆一位二进制数。它们具有以下特性：

1. 有两个互补的输出端 $Q$ 和 $\bar{Q}$ 。当 $Q=1$ 时， $\bar{Q}=0$ ；而当 $Q=0$ 时， $\bar{Q}=1$ 。
2. 有两个稳定状态。通常将 $Q=1$ 和 $\bar{Q}=0$ 称为“1”状态，而把 $Q=0$ 和 $\bar{Q}=1$ 称为“0”状态。若输入不发生变化，锁存器必定处于其中一个状态，并且长期保持下去。
3. 在输入信号的作用下，锁存器可以从一个稳定状态转换到另一个稳定状态。

我们把输入信号发生变化前的锁存器状态称为**现态**，用 $Q^n$ 和 $\bar{Q}^n$ 来表示，而把输入信号发生变化后锁存器所进入的状态，称为**次态**，用 $Q^{n+1}$ 和 $\bar{Q}^{n+1}$ 来表示。若用 $X$ 来表示输入信号的集合，则锁存器的次态是它的现态和输入信号的函数，即

$$Q^{n+1} = f(Q^n, X)$$

该函数称为锁存器的次态方程，又称状态方程，它是描述时序电路的通用表达式。由于每一种具体的锁存器都有自己特定的状态方程，因此，也称为特征方程。常用的锁存器有基本 SR 锁存器、门控 SR 锁存器和门控 D 锁存器。

### 2.3 触发器

锁存器虽然能记忆一位二进制数，但接受的输入数据是在使能信号 EN 控制下进行的。EN 是电位信号，如果在 EN 有效期间，输入数据受到瞬时的干扰发生电平变化，那么锁存器就可能锁存错误的数据。为了提高锁存器工作的可靠性，人们又创新改进，推出边沿方式工作的触发器。

触发器是一种同步双稳态器件，同样用来记忆一位二进制数。所谓同步，是指触发器的记忆状态按时钟规定的启动指示点（脉冲边沿）来改变。触发器可在时钟脉冲的正沿（上升沿）改变状态，也可以在时钟脉冲的负沿（下降沿）改变状态。

### 2.4 触发器的应用

触发器是构成复杂时序逻辑电路最基本的组成单元。按照触发器在时序逻辑电路中的作用，它的应用主要有以下几个方面：

1. 用作并行数据寄存器。一个触发器只存储一位二进制数，若并行存储  $n$  位二进制数，则需要  $n$  个触发器。这  $n$  个触发器按并行方式连接就构成并行数据寄存器，简称寄存器。

2. 用作计数器。1 个触发器用作计数器时，可记忆两个状态；2 个触发器按串行方式连接成 2 位计数器时，可以记忆  $2^2=4$  个状态；n 个触发器按串行方式连接成 n 位计数器时，可记忆  $2^n$  个状态。这  $2^n$  个状态对应了  $2^n$  个时钟脉冲。换句话说，计数器记忆时钟脉冲的个数。
3. 用作分频器。一个触发器接成计数模式按时钟脉冲的固定频率工作，若时钟的频率为  $f_n$ ，则该触发器 Q 端输出信号的频率为  $f_n/2$ 。触发器的这种应用称为分频。

## 2.5 时序电路的设计与测试

同步时序电路的特点是：电路中时间的划分是以时钟脉冲为依据的。其设计的主要步骤是：根据设计要求写出动作说明，列出状态图或状态转换表，然后进行状态化简和状态分配，再根据所选触发器确定其驱动方程，然后画出电路图。当然，在设计中有时考虑自启动也是必不可少的。在进行设计时，不要拘泥于以上程式，应该融会贯通，灵活掌握。对于所设计的逻辑电路图，必须进行实验检测，只有实际电路符合设计要求时，才能证明设计是正确的。

时序电路的功能测试分静态和动态两种方法。静态测试就是直流稳态测试，就是测试电路的状态转换真值表。测试时，时钟脉冲由逻辑开关提供，电路输出用数字探针指示。动态测试是指，在时钟输入端输入矩形波或方波信号（自动时钟源），用逻辑分析仪观察电路各级的工作波形，它不仅可以看到电路的稳态情况，而且还可以观察到电路的过渡态（或叫瞬态）。

## 三、实验任务

请读者按照下面方法之一在本地创建一个项目，用于完成本次任务：

### 方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建个人项目，然后使用 Dream Logic 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

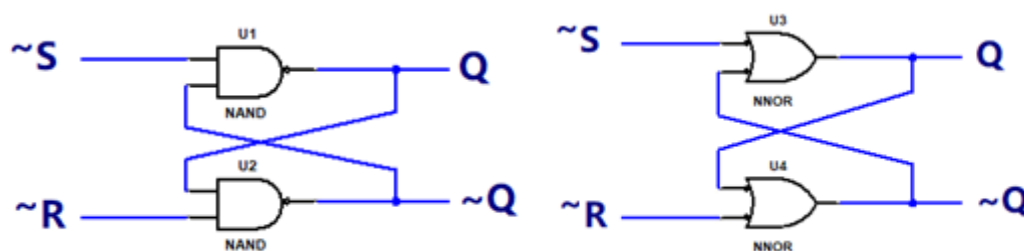
### 方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，就需要使用 Dream Logic 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的 URL

为 <https://www.codecode.net/engintime/Dream-Logic/Project-Template/Digital-Circuit/Lab004.git>

### 基本 SR 锁存器

基本 SR 锁存器有三种形式，如图 4-1 所示。 $\sim S$  和  $\sim R$  表示低电平有效，S 和 R 表示高电平有效。请读者按照下面的步骤仿真验证基本 SR 锁存器的功能：



第三种形式——或非门（注意 R 和 S 的位置）

图 4-1：基本 SR 锁存器的三种形式

1. 打开项目下的原理图文件 SR.dlsche。
2. 启动仿真。根据第三种形式的仿真结果完成 SR 锁存器的特征表 4-2，并根据特征表，写出状态方程。在表 4-2 中的现态是指，当 S 或 R 发生变化后进入的状态，此时新的 Q 值作为次态。例如，读者通过按键盘上的 5、6 键，将 S 和 R 设置为 0，然后启动仿真，此时 Q 的值为 1，读者按下键盘上的 5，使 R 变为 1，则此瞬间 S、R、Q 的值分别为 0、1、1，这个就是表中的第 4 个现态，记录此时 Q 的值为 0，称为次态。

**注意：**在启动仿真之前，不要将第二种形式的 $\sim S$ 和 $\sim R$ 同时设置为 1，这样会导致无法启动仿真。

现态			次态
S	R	Q	$Q^{n+1}$
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	状态不定
1	1	1	状态不定

表 4-2：由或非门组成的基本 SR 锁存器特征表

### 3.1 门控 SR 锁存器

门控 SR 锁存器如图 4-3 所示。它是在基本 SR 锁存器的基础上加以改进，增加一级输入与非门，由使能控制信号 EN 进行控制。EN 有效时，锁存器才接收数据输入信号；EN 无效时，锁存器拒绝接收数据输入信号。门控 SR 锁存器也叫电平触发 SR 触发器，通常使用时钟 CLK 作为门控信号，也就是触发信号。

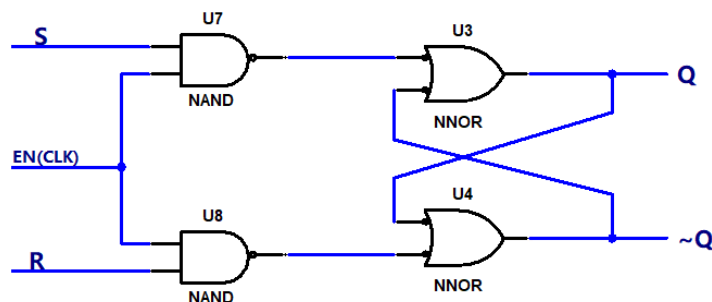


图 4-3：门控 SR 锁存器（电平触发 SR 触发器）

按照下面的步骤验证门控 SR 锁存器的功能：

1. 打开项目下的原理图文件 SR\_gate.dlsche。
2. 参考表 4-2 为门控 SR 锁存器设计一个特征表，只考虑 EN=1 的情况即可。通过仿真验证门控 SR 锁存器的功能，并将其特征表填写完整。

### 3.2 门控 D 锁存器

图 4-5 是门控 D 锁存器。它与门控 SR 锁存器相同处在于第一级都是两个与非门，不同处在于只有一个数据输入端 D。D 输入经过一个非门加到原来门控 SR 锁存器的 R 输入端，变成互补输入，所以 D 锁存器是门控 SR 锁存器的一种改进形式。其工作原理是：当数据输入 D=1 且使能控制 EN=1，锁存器置 1；当 D=0 且 EN=1 时，锁存器置 0。门控 D 触发器也称为电平触发 D 触发器，通常使用时钟 CLK 作为门控信号，也就是触发信号。

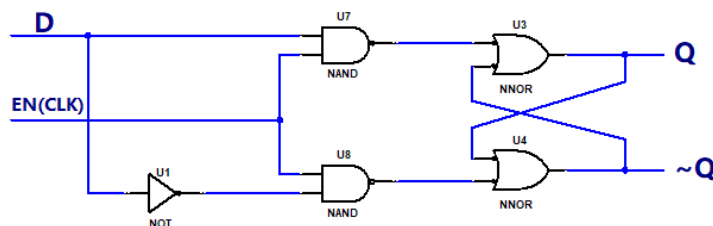


图 4-5: 门控 D 锁存器（电平触发 D 触发器）

按照下面的步骤验证门控 D 锁存器的功能：

1. 打开项目下的原理图文件 D\_gate.dlsche。
2. 参考表 4-2 为门控 D 锁存器设计一个特征表，只考虑 EN=1 的情况即可。通过仿真验证门控 D 锁存器的功能，并将其特征表填写完整。

### 3.3 边沿触发 D 触发器

电平触发 D 触发器在 EN（CLK）的有效电平期间输出 Q 始终跟随输入 D 变化，输出与输入的状态保持相同，所以又将其称为“透明的 D 型锁存器”。

为了提高触发器的可靠性，增强抗干扰能力，希望触发器的次态仅仅取决于 CLK 信号下降沿（或上升沿）到达时刻输入信号的状态。可以将其比作一个瞬时采样器，在时钟的边沿对输入 D 进行采样，而与时钟边沿前后的输入无关，这样就只需要保证时钟沿到来的那个瞬间，输入数据 D 稳定在一个有效值即可。为了实现这一设想，人们相继研制成了各种边沿触发器。边沿触发 D 触发器是由两个电平触发 D 触发器组成的。

请读者按下列步骤验证边沿触发 D 触发器的功能：

1. 打开项目下的原理图文件 D\_edge.dlsche。
2. 启动仿真，根据表 4-7 验证边沿触发 D 触发器的功能。

输入		输出	
CLK	D	Q	$\sim Q$
X	X	保持	保持
↑（上升沿）	1	1	0
↑（上升沿）	0	0	1

表 4-7: 边沿触发 D 触发器功能表

### 3.4 JK 触发器

在上述边沿触发 D 触发器实验中，使用 D 触发器的内部电路进行仿真测试其功能。在实际的应用中，需要将 D 触发器的内部电路封装起来，仅向外提供输入/输出管脚，以便使用。下面，就对另外一个常用的边沿触发器 JK 触发器进行功能验证。步骤如下：

1. 打开项目下的原理图文件 jk.dlsche。
2. 启动仿真。按 JK 触发器的功能表 4-8 进行验证。

输入					输出	
PR	CLR	CLK	J	K	Q	$\sim Q$
1	1	X	X	X	X	X
1	0	X	X	X	1(置 1)	0
0	1	X	X	X	0(清 0)	1
0	0	X	X	X	保持	保持
0	0	↑	1	0	1	0
0	0	↑	0	1	0	1
0	0	1	0	0	保持	保持
0	0	↑	1	1	$\sim Q^n$ (翻转)	$Q^n$

表 4-8: JK 触发器的功能表（ $Q^n$  是触发转换前的状态）

### 3.5 二分频器

数字电路中的分频是指将源信号的频率降低为原来的  $N$  分之一，就称为  $N$  分频。所以这里说的二分频就是将源信号的频率降低为原来的二分之一。实现分频的电路称为分频器。

请读者按照下面的步骤观察一个二分频电路的实现方法，理解分频器的原理：

1. 打开项目下的原理图文件 `freq_divider2.dlsch`。该原理图文件中绘制了一个使用 D 触发器实现的二分频电路，并分别将源信号 CLK 和分频后的信号 CLK\2 接到了逻辑分析仪的管脚 1 和管脚 2，这样就可以很方便的使用逻辑分析仪比较两个信号的波形。
2. 启动仿真。
3. 按下键盘上的 R 键初始化 D 触发器。
4. 双击逻辑分析仪 XLA1，打开逻辑分析仪窗口，观察两个不同频率的波形。为了便于观察和比较波形，也可以先按 F6 暂停仿真再进行观察。在逻辑分析窗口中可以调整比例对波形进行缩放，还可以使用窗口底部的滚动条拖动波形。

**注意：**时钟频率和仿真步长有一定的关系，仿真步长要小于时钟周期，这样仿真引擎才能对一个周期内的电路状态完成采样。

### 3.6 设计四分频器

请读者参考二分频器的设计方法，使用 D 触发器设计一个四分频器，并仿真验证其功能。步骤如下：

1. 在项目下新建一个原理图文件 `freq_divider4.dlsche`。新建原理图文件的方法可以参考实验一中的相关内容。
2. 使用 D 触发器设计一个四分频电路，将源信号 CLK 和分频后的信号 CLK\4 接到逻辑分析仪的管脚 1 和管脚 2。在原理图中添加逻辑分析仪的方法是：选择菜单“仪器->逻辑分析仪”，将鼠标移动到原理图中，单击鼠标左键即可完成绘制。在型号库“触发器”中可以找到 D 触发器，在型号库“数字信号源”中可以找到单周期时钟和时钟。
3. 通过仿真检验四分频电路的功能，确保其可以产生正确的信号。

### 提交作业

在提交作业前，读者需要将四分频器电路 SVG 图形文件的链接添加到 README.md 文件中，这样，当使用浏览器查看提交后的线上项目时，就可以从 README.md 文件中看到四分频器的电路了。方法如下：

1. 双击“项目管理器”中的 README.md 文件，使用一种合适的文本编辑器或者 Markdown 编辑器打开此文件。
2. 在 README.md 文件的末尾添加如下的文本：  
`# 四分频器电路`  
`![raw svg](freq_divider4.dlsche.svg)`
3. 保存 README.md 文件。
4. 提交作业。

# 实验 5 寄存器和计数器

实验性质：验证+设计

建议学时：2 学时

## 一、实验目的

- 掌握使用触发器设计寄存器和计数器的方法。
- 掌握计数器和移位寄存器的电路结构和工作原理。

## 二、预备知识

### 2.1 寄存器

由锁存器或触发器组成、一次能够并行存储  $N$  位比特数据的逻辑部件称为寄存器。寄存器是计算机和数字系统中最常见的功能部件。锁存器与触发器构成的寄存器工作方式不同。

- **锁存器构成的寄存器：**寄存器中每一位对应一个锁存器，所有的锁存器共用一个门控信号  $EN$ ，当  $EN$  有效时，所有锁存器的输入传送到输出端， $EN$  无效后，寄存器的输出与输入隔离，若  $EN$  始终无效，那么寄存器就一直保存之前存入的数据不变。74LS373 就是由 8 个 D 锁存器构成的 8 位寄存器。
- **触发器构成的寄存器：**寄存器中每一位对应一个触发器，所有的触发器共用一个时钟  $CLK$ ，当时钟上升沿（或下降沿）到来时，所有触发器的输入端数据复制到输出端，寄存器的输出会一直保持不变，直到下个时钟上升沿（或下降沿）到来时，才会重新写入输入端的数据。74LS374 就是由 8 个 D 触发器构成的 8 位寄存器。

常用的寄存器大多由 D 触发器构成，这是因为 D 触发器采用时钟边沿触发方式，工作十分可靠，且只有一个数据输入端，使用也很方便。

### 2.2 移位寄存器

在时钟信号的控制下，将寄存器的数据向左或向右移位的寄存器称为移位寄存器。图 5-1 所示的是一个 4 位的右移寄存器逻辑图，它的结构很简单，只需要把左面一位触发器的输出端接到右面一位触发器的输入端，即连接关系满足  $D_i = Q_{i-1}$ ，同时把所有触发器的时钟端接在一起，用同步脉冲信号控制。

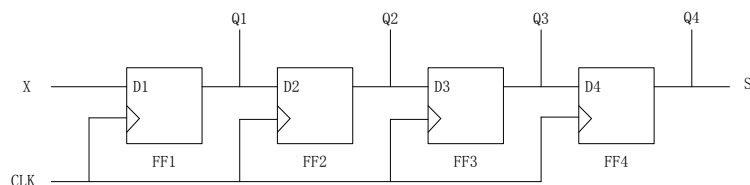


图 5-1：右移寄存器逻辑图

实际应用中常常采用中规模通用移位寄存器，它是将若干触发器和逻辑门集成在一块芯片上，从而为使用带来很大方便。在移位控制信号作用下，既能左移又能右移的，则称之为双向移位寄存器。移位寄存器可以临时寄存信息，也可以加工或传送数据，还可作为基本部件用于各种时序逻辑电路，如计数器、脉冲分配器、序列码发生器、序列码检测器等。

74LS299 就是一种 8 位的通用移位寄存器，它有多种工作模式，可并行置数、左移、右移、保持数据，也可以实现并入并出、并入串出、串入串出、串入并出操作。

通用移位寄存器用途十分广泛，在计算机系统中可用作累加寄存器、缓冲寄存器、串-并转换器等。

### 2.3 计数器

计数器的功能是记忆脉冲的个数，它是数字系统中应用最广泛的基本时序逻辑构件。计数器所能记忆

脉冲的最大数目称为计数器的**模**，用字母 M 表示。构成计数器的核心元件是触发器。

计数器的种类繁多，分类方法也不同。①按计数功能来分，可分为加法计数器、减法计数器、可逆计数器（加/减计数器）；②按进位基数来分，可分为二进制计数器、十进制计数器、任意进制计数器；③按进位方式来分，可分为同步计数器、异步计数器。

### 1. 同步计数器

同步计数器电路中，所有触发器的时钟都与同一个时钟源连在一起，每个触发器的状态变化都与时钟同步。

### 2. 异步计数器

异步计数器中各个触发器的时钟不是来自于同一个时钟源。第一个触发器的状态变化与时钟同步，其他则依次滞后。异步计数器的工作原理有点像“多米诺骨牌”。

### 3. 计数器的预置功能

计数器通常有一个预置控制端 LD，用来设置计数器开始计数时的初始值。预置分为两种方式：

- 同步预置（同步置数）。预置控制信号 LD 有效后并不立即将计数器的输入值加载到输出端，而是要等待时钟有效边沿（上升沿或下降沿）到来时才能完成预置功能，即预置的实现与时钟有效边沿同步。
- 异步预置（异步置数）。一旦预置信号 LD 有效，计数器立即置数，而与时钟无关。

### 4. 计数器的复位（清零）功能

计数器通常有一个复位控制端 CLR，用来清零计数器。复位功能也分为同步复位和异步复位，其含义与同步置数与异步置数的含义类似。

### 5. 时钟有效边沿的选择

若计数器对时钟的上升沿进行计数，即每来一个时钟上升沿，计数器加 1 或减 1，那么可称该计数器为上升沿计数器；反之则可称之为下降沿计数器。

### 6. 计数器级联

通过将计数器进行级联可以扩展计数器的位数，也就是计数器的模。

## 三、实验任务

请读者按照下面方法之一在本地创建一个项目，用于完成本次任务：

#### 方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建个人项目，然后使用 Dream Logic 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

#### 方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，就需要使用 Dream Logic 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的 URL

为 <https://www.codecode.net/engintime/Dream-Logic/Project-Template/Digital-Circuit/Lab005.git>

### 3.1 锁存器构成的寄存器和触发器构成的寄存器

比较锁存器和触发器构成的寄存器工作方式的不同之处。

1. 打开项目下的原理图 reg.dlsche，结合原理图中的相关注释，熟悉原理图中的器件及其功能。
2. 启动仿真。
3. 通过键盘按键 E，设置寄存器的使能端输入 EN=0（使能），观察电路的状态。
4. 选择八位交互式数字信号源，在属性栏中将其数字信号值改为 08。然后，观察两个寄存器的输出值，有何不同？
5. 按下并抬起键盘按键 C，向寄存器 74LS377D 输入一个时钟上升沿，再次观察它的输出值，有何变

化？

通过前面的实验步骤可知，在两种类型的寄存器都使能的情况下，锁存寄存器的输出始终与输入端的数据保持一致，而触发寄存器的输入数据必须等待时钟上升沿到来时，才能传送到寄存器的输出端。下面检验使能端 EN 对寄存器的控制作用。

1. 通过键盘按键 E，设置寄存器使能输入端 EN=1（不使能）。
2. 将八位交互式数字信号源的值修改为 4b。
3. 观察两个寄存器的输出值是否为 4b？按下并抬起键盘按键 C，寄存器 74LS377D 的值有变化吗？
4. 设置 EN=0，再次观察寄存器输出值是否为 4b？
5. 按下键盘按键 C 不动，观察触发寄存器的输出值是否为 4b？抬起按键 C 后呢？

通过实验可知，当锁存寄存器不使能时，它的输出值保持不变，不会受输入值的影响，一旦使能，则立即将输入数据传送到输出端；当触发寄存器不使能时，即使时钟上升沿到来，也不能将输入端的数据传送到输出端。

### 3.2 移位寄存器

学习右移寄存器的工作原理。请读者按下列步骤进行实验：

1. 打开项目下的原理图文件 shift\_reg.dlsche，查看 4 位右移寄存器的电路。
2. 启动仿真，观察各个 D 触发器的输出值，蓝色网络表示无效电平。
3. 通过按键 D 设置第一个触发器的数据输入 D1=1。
4. 通过按键 C 输入一个时钟脉冲，观察 Q1~Q4 的值，可以看到 Q1=1。
5. 通过按键 D 设置 D1=0。
6. 通过按键 C 逐个输入时钟脉冲，可以看到 Q1~Q4 指示灯依次点亮。表示输入数据在向右移位。

通过上述实验可知，右移寄存器的工作原理是，在同一个时钟驱动下，一序列二进制数依次向右传递。

### 设计循环移位寄存器

假设 Q1、Q2、Q3、Q4 的初始值为 1000，需要在时钟的驱动下进行循环右移，该如何改进电路？请读者在右移寄存器的基础上直接修改此电路图，实现一个 4 位循环右移寄存器。

**提示：**实现循环移位，需要将 Q4 端接到左侧第一个寄存器的数据输入端，形成一个数据环路。启动仿真后，需要使用触发器的异步置数端 PR 和异步复位端 CLR 将各个触发器初始化为对应的值，也就是 Q1 为 1，Q2、Q3、Q4 为 0，然后再触发时钟产生循环移位。

### 3.3 移位寄存器 74LS194

验证移位寄存器 74LS194（参见附录 C）的功能，注意 SR 触发器的使用方法：

1. 打开项目下的原理图 74LS194.dlsche。该原理图文件中绘制了 74LS194 的内部原理图。
2. 启动仿真，根据表 5-2 对 74LS194 进行功能验证。

输入								输出				功能		
CLR	Mode		CLK	Serial		Parallel				QA	QB		QC	QD
	S1	S0		SLSR		A	B	C	D					
0	X	X	X	X	X	X	X	X	X	0000				清除
1	X	X	X	X	X	X	X	X	X	Q <sub>A0</sub>	Q <sub>B0</sub>	Q <sub>C0</sub>	Q <sub>D0</sub>	保持
1	11		↑	X	X	a	b	c	d	a	b	c	d	并行送数
1	01		↑	X	1	X	X	X	X	1	Q <sub>An</sub>	Q <sub>Bn</sub>	Q <sub>Cn</sub>	右移
1	01		↑	X	0	X	X	X	X	0	Q <sub>An</sub>	Q <sub>Bn</sub>	Q <sub>Cn</sub>	
1	10		↑	1	X	X	X	X	X	Q <sub>Bn</sub>	Q <sub>Cn</sub>	Q <sub>Dn</sub> 1		左移
1	10		↑	0	X	X	X	X	X	Q <sub>Bn</sub>	Q <sub>Cn</sub>	Q <sub>Dn</sub> 0		
1	00		X	X	X	X	X	X	X	Q <sub>A0</sub>	Q <sub>B0</sub>	Q <sub>C0</sub>	Q <sub>D0</sub>	保持



表 5-2: 74LS194 功能表

### 3.4 使用 JK 触发器设计的同步计数器

测试一个由 3 个 JK 触发器构成的模 8 同步计数器。实验步骤如下:

1. 打开项目下的原理图文件 m8.dlsche。这是一个由 3 个 JK 触发器构成的 3 位模 8 同步计数器, 结合电路注解, 理解其工作原理。
2. 启动仿真, 通过按键 C 依次输入时钟脉冲, 观察计数显示器的计数值。

### 设计模 16 同步计数器

在电路中添加一个 JK 触发器以及必要的逻辑门, 实现一个模 16 同步计数器, 并通过仿真验证。

### 3.5 使用 D 触发器设计加法计数器

接下来, 读者可以学习使用 D 触发器设计一个模 7 加法计数器的完整过程, 使读者掌握计数器设计的一种基本方法。加法计数器是一种简单的有限状态机, 它在时钟沿的驱动下, 由当前稳定状态 (当前计数值) 进入下一个状态 (计数器加 1 后的值)。请读者按照下列步骤, 完成模 7 加法计数器的设计:

1. 首先在项目下新建一个原理图文件 m7.dlsche。
2. 模 7 加法计数器可以表示的值为  $0 \sim 6$ , 至少需要 3 位二进制数才能表示, 所以需要 3 个 D 触发器。每一个 D 触发器存储一位二进制数。在原理图中依次放置 3 个 D 触发器, 各个触发器之间留出一定的间隔, 以便放置其他器件。如图 5-3 所示。

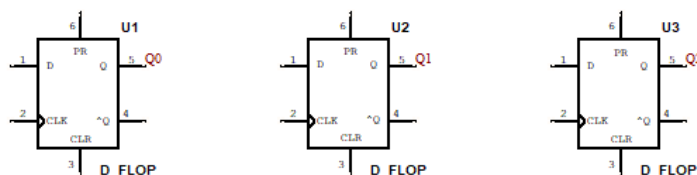


图 5-3: 模 7 加法计数器的 3 个 D 触发器

3. 为了可以随时清零计数器, 使其从 0 开始计数, 就需要在原理图中添加一个手动的复位按键 R, 如图 5-4 所示。

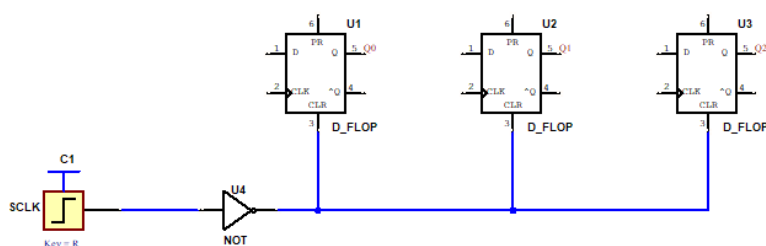


图 5-4: 模 7 加法计数器的复位控制

4. 完成复位电路后, 启动仿真, 测试复位功能。按下复位按键 R 后抬起, 使  $Q_2Q_1Q_0=000$  (计数器值为 0)。

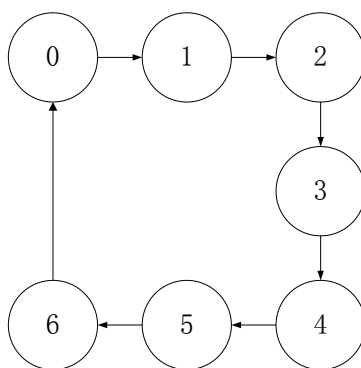


图 5-5: 模 7 加法计数器的状态转换图

5. 模 7 加法计数器的状态转换如图 5-5 所示。当  $Q_2Q_1Q_0$  稳定为某个状态时, 表示一个唯一的数, 例如  $Q_2Q_1Q_0=011$  时表示 3, 那么下一个计数值一定是  $Q_2Q_1Q_0=100$ , 也就是当前状态决定了下一个状态。所以, 只需得到次态方程  $Q^{n+1}=f(Q_{2n}, Q_{1n}, Q_{0n})$  就能通过逻辑电路实现当前状态到下一个状态的转换。状态转换如表 5-6 所示。

时钟 个数	PS (现态)	NS (次态)
	$Q_2^n Q_1^n Q_0^n$	$Q_2^{n+1} Q_1^{n+1} Q_0^{n+1}$
0	000	0 0 1
1	0 0 1	0 1 0
2	0 1 0	0 1 1
3	0 1 1	1 0 0
4	1 0 0	1 0 1
5	1 0 1	1 1 0
6(循环)	1 1 0	000

表 5-6: 模 7 加法计数器的状态转换表

**提示:**  $Q_2^n$ 、 $Q_1^n$ 、 $Q_0^n$  分别表示当前各位触发器的值,  $Q_2^{n+1}$ 、 $Q_1^{n+1}$ 、 $Q_0^{n+1}$  分别表示时钟上升沿到来后各位触发器的值。

6. 由模 7 加法计数器的状态转换表, 可得到各位的次态方程。当前计数值为 3、4、5 时, 下一个状态  $Q_2^{n+1}$  为 1。若用  $m0 \sim m6$  表示计数值 0~6, 那么

$$Q_2^{n+1} = m3 + m4 + m5 \quad (m_x \text{ 为高电平时有效})$$

同理可得

$$Q_1^{n+1} = m1 + m2 + m5$$

$$Q_0^{n+1} = m0 + m2 + m4$$

7. 可以由  $Q_2^n$ 、 $Q_1^n$ 、 $Q_0^n$  通过 3-8 译码器得到  $m0 \sim m5$  这 6 个值对应的信号。当译码输出任意一位有效时, 就能确定当前计数值。例如,  $m5$  有效 (3-8 译码器输出低电平表示有效), 那么当前计数值为 5。于是得到最终的电路如图 5-7 所示。注意在图 5-7 中, 计数器输出的  $Q_2Q_1Q_0$  通过网络标签与数码管和 3-8 译码器的输入端连接, 这样既可显示计数器的值, 又对计数器的现态完成了译码。

8. 读者仿照图 5-7 绘制完成电路后, 可以仿真验证模 7 加法计数器的功能。

### 设计模 7 减法计数器

通过上述实验步骤, 最终设计了一个模 7 加法计数器。请读者在项目下新建一个原理图文件 `m7_minus.dlsche`, 并仿照上述步骤设计一个模 7 减法计数器。计数过程为  $0 \rightarrow 6 \rightarrow 5 \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 1 \rightarrow 0 \rightarrow 6 \rightarrow \dots$ 。

**提示:**

- ① 先画出模 7 减法计数器的状态转换表,
- ② 根据状态转换表, 写出次态方程;
- ③ 根据次态方程, 完成电路设计。

### 提交作业

在提交作业前, 读者需要将模 7 加法计数器和减法计数器 SVG 图形文件的链接添加到 README.md 文件中, 这样, 当使用浏览器查看提交后的线上项目时, 就可以从 README.md 文件中看到电路了。方法如下:

1. 双击“项目管理器”中的 README.md 文件, 使用一种合适的文本编辑器或者 Markdown 编辑器打开此文件。
2. 在 README.md 文件的末尾添加如下的文本:

```
# 模 7 加法计数器
```

```
![raw svg](m7.dlsche.svg)
```

```
# 模 7 减法计数器
```

```
![raw svg](m7_minuse.dlsche.svg)
```

## 3. 保存 README.md 文件。

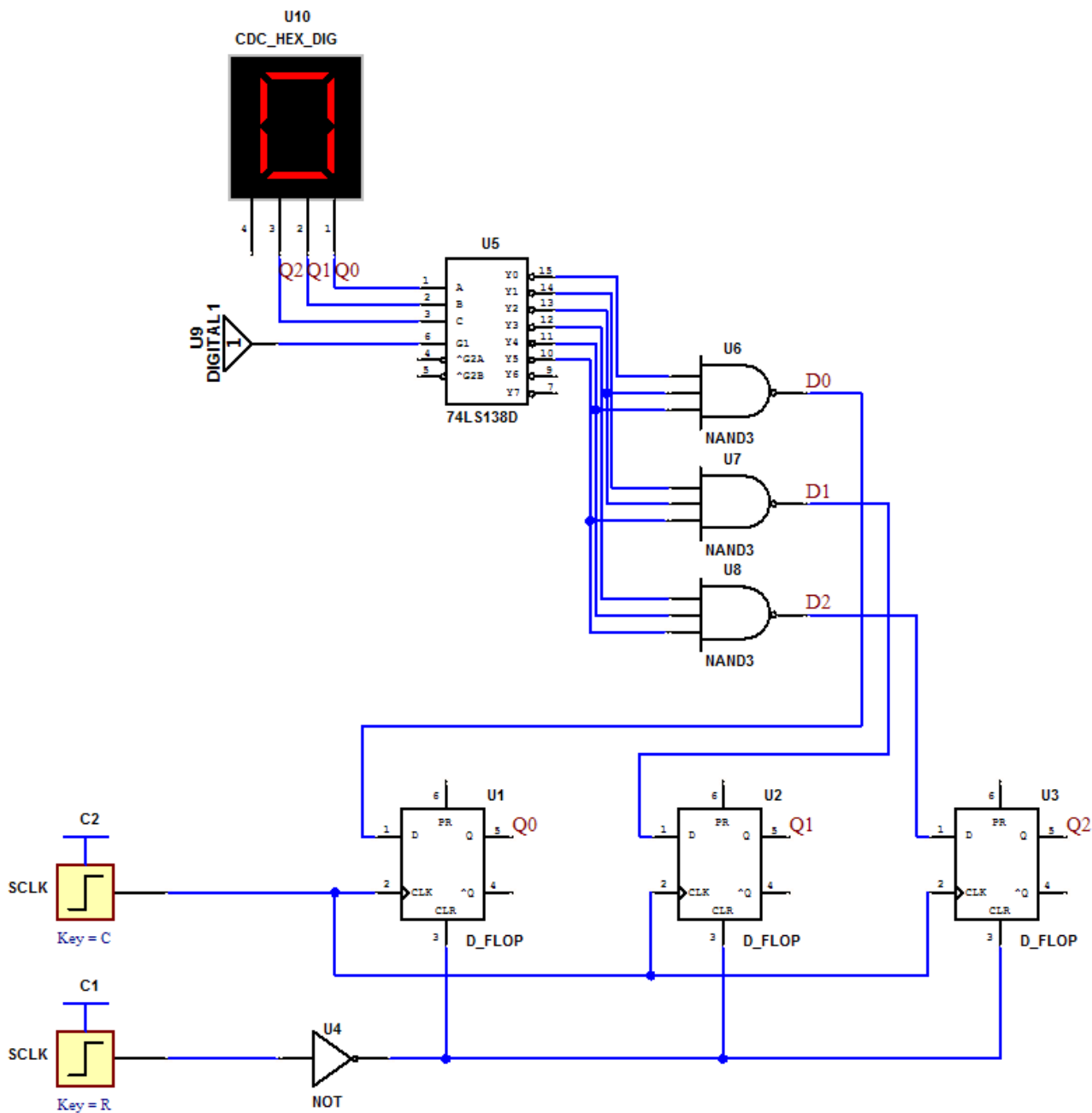


图 5-7：模 7 加法计数器

## 3.6 异步模 16 加法计数器

仿真验证异步模 16 加法计数器：

1. 打开项目下的原理图文件 m16.dlsche。
2. 启动仿真，验证其计数功能，并尝试描述其工作原理。

## 设计异步模 10 加法计数器

将异步模 16 加法计数器改造为异步模 10 加法计数器。通过改造电路，使计数器的计数到达 9 后又从 0 开始计数。提示，可以在计数值为 10 时，立即清零，这样就成了一个模 10 的计数器。

## 3.7 同步置数/异步清零十六进制加法计数器 74LS161

仿真验证 74LS161 的功能：

1. 打开项目下的原理图文件 74LS161.dlsche。该原理图文件中绘制了 74LS161 的内部原理图。

2. 启动仿真，根据表 5-8 的内容对 74LS161 进行功能验证。

输入								输出			
CLR	LOAD	ENT	ENP	CLK	ABC	D		QA	QB	QC	QD
0	X	X	X	X	X	X	X	X	X	X	0000
1	0	X	X	↑	abc	d		a	b	c	d
1	1	1	1	↑	X	X	X	X	X	X	计数(加 1)
1	1	0	X	X	X	X	X	X	X	X	保持
1	1	X	0	X	X	X	X	X	X	X	保持

表 5-8: 74LS161 功能表

## 四、思考与练习

1. 在并行输入、串行输出的转换中，若二进制数码高位在前、低位在后，应采取何种移位方式？
2. 时序电路自启动的作用何在？它和人工预置的方法比较，对电路的作用有何异同？
3. 环形计数器的最大优点和最大缺点各是什么？为什么环形计数器的输出在译码时不存在竞争冒险？
4. 并使用两片 74LS161 芯片设计一个模 37 加法计数器。提示，可以使用两片 74LS48 显示两位十进制的计数值。

# 实验 6 顺序脉冲发生器

实验性质：设计

建议学时：2 学时

## 一、实验目的

- 掌握顺序脉冲发生器的原理。
- 掌握顺序脉冲发生器的设计方法。

## 二、预备知识

### 2.1 定时脉冲/节拍脉冲

按固定时间顺序再现的脉冲序列称为定时脉冲，也称为节拍脉冲。一个数字系统之所以有条不紊的工作，完全是受到定时脉冲的指挥。

### 2.2 顺序脉冲发生器

在数字系统和计算机中，往往需要机器按照人们事先规定的顺序进行运算或操作，这就要求机器的控制部分不仅能正确地发出各种控制信号，而且要求这些控制信号在时间上有一定的先后顺序。用顺序脉冲发生器可以实现这一功能。顺序脉冲也叫相位脉冲，或节拍脉冲。计算机之所以能一步一步地运行，就是要靠节拍脉冲一拍一拍地指挥。

顺序脉冲发生器用于产生时间上有先后顺序的脉冲，通常可以用移位寄存器产生，也可以由计数器和译码器组合而成，计数器状态提供译码器的地址码，译码器将该地址代码译成有一定顺序的点位脉冲。

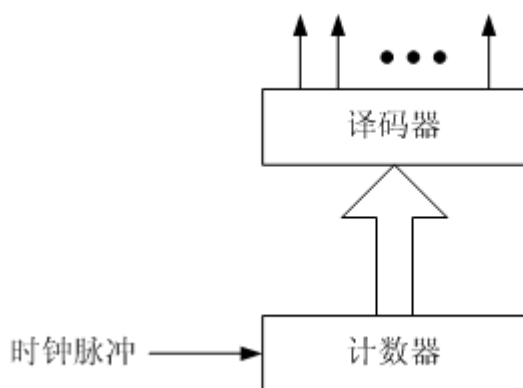


图 6-1：顺序脉冲实现原理图

## 三、实验内容

请读者按照下面方法之一在本地创建一个项目，用于完成本次任务：

### 方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建个人项目，然后使用 Dream Logic 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

### 方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，就需要使用 Dream Logic 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的 URL

为 <https://www.codecode.net/engintime/Dream-Logic/Project-Template/Digital-Circuit/Lab006.git>

### 3.1 自然二进制码同步八进制计数器顺序脉冲发生器

观察一个由自然二进制码同步八进制计数器构成的顺序脉冲发生器的波形，理解顺序脉冲发生器的工作原理。

1. 打开项目下的原理图文件 binary.dlsche。原理图中包含一个由 JK 触发器构成的同步模 8 计数器，一个 3-8 译码器和一个逻辑分析仪。3-8 译码器输出的 8 相顺序脉冲分别与逻辑分析仪的 8 个通道连接。通过逻辑分析仪可以观察顺序脉冲波形。
2. 启动仿真，观察计数器值的变化情况。双击逻辑分析仪，观察脉冲波形，理解顺序脉冲发生器的工作原理。

### 3.2 设计格雷码同步八进制计数器顺序脉冲发生器

在项目下新建一个原理图文件 gray.dlsche，在其中设计一个格雷码同步八进制计数器，并使用该计数器以及 3-8 译码器实现一个 8 相顺序脉冲发生器。格雷码如表 6-2 所示。产生的顺序脉冲信号应与图 6-3 中的波形一致。

十进制数	8421 码	余 3 码	格雷码
0	0000	0011	0000
1	0001	0100	0001
2	0010	0101	0011
3	0011	0110	0010
4	0100	0111	0110
5	0101	1000	1110
6	0110	1001	1010
7	0111	1010	1000
8	1000	1011	1100
9	1001	1100	1101

表 6-2：常用 BCD 码

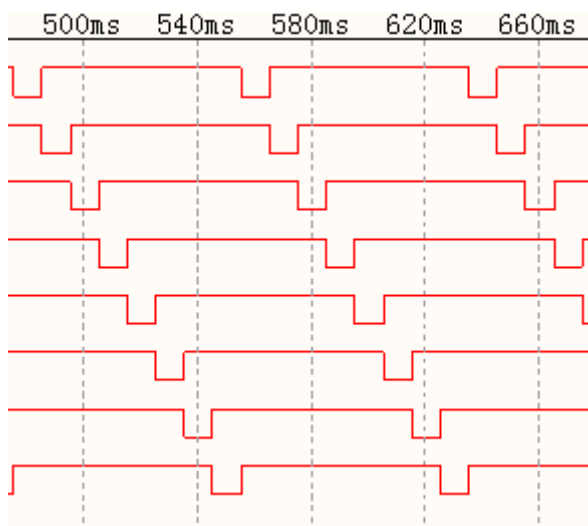


图 6-3：顺序脉冲信号

#### 提交作业

在提交作业前，读者需要将顺序脉冲发生器 SVG 图形文件的链接添加到 README.md 文件中，这样，当使用浏览器查看提交后的线上项目时，就可以从 README.md 文件中看到电路了。方法如下：

1. 双击“项目管理器”中的 README.md 文件，使用一种合适的文本编辑器或者 Markdown 编辑器打开此文件。
2. 在 README.md 文件的末尾添加如下的文本：

# 格雷码同步八进制计数器顺序脉冲发生器

![raw svg](gray.dlsche.svg)

3. 保存 README.md 文件。
4. 提交作业。

## 四、思考与练习

1. 计数器采用异步有何优缺点？
2. 计数器状态采用格雷码的目的是什么？
3. 若用移位寄存器获得节拍脉冲信号，其电路是怎样的？
4. 使用 74LS90 设计一个占空比为 50% 的十分频器。



# 实验 7 序列信号的产生和检测

实验性质：设计

建议学时：2 学时

## 一、实验目的

- 掌握序列信号发生器的设计方法。
- 掌握序列信号检测器的设计方法。

## 二、预备知识

### 2.1 序列信号发生器原理

在数字信号的传输和数字系统的测试中，有时需要用到一组特定的串行数字信号，称之为序列信号。产生序列信号的电路称为序列信号发生器。

序列信号发生器可由计数器和数据选择器构成，其结构如图 7-1 所示。其中的锁存输出功能是为了消除序列信号产生时可能出现的毛刺现象。

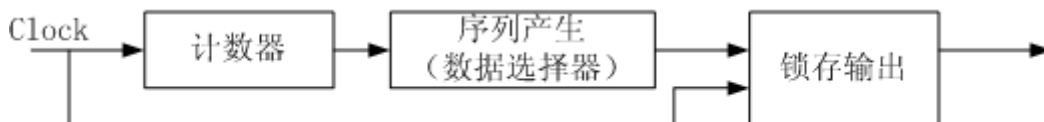


图 7-1：序列信号发生器结构图

### 2.2 序列信号检测器的基本工作过程

序列信号检测器用于检测一组或多组由二进制码组成的脉冲序列信号，在数字通信中有着广泛的应用。当序列信号检测器连续收到一组串行二进制码后，如果这组码与检测器中预先设定好的码相同，则输出 1，否则输出 0。由于这种检测的关键在于正确码的接收必须是连续的，这就要求检测器必须记住前一次的正确码及正确序列，直到在连续的检测中所收到的每一位码都与预置的对应码相同。在检测过程中，任何一位不相等都将回到初始状态重新开始检测。

## 三、实验内容

请读者按照下面方法之一在本地创建一个项目，用于完成本次任务：

### 方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建个人项目，然后使用 Dream Logic 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

### 方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，就需要使用 Dream Logic 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的 URL

为 <https://www.codecode.net/engintime/Dream-Logic/Project-Template/Digital-Circuit/Lab007.git>

### 3.1 8 选 1 数据选择器 74LS151

由于设计序列信号发生器的时候，需要用到 8 选 1 数据选择器 74LS151（参见附录 C），所以在这里先了解一下它的功能：

1. 打开项目下的原理图文件 74LS151.dlsche。该原理图文件中绘制了 74LS151 的内部原理图。
2. 启动仿真，根据表 7-2 验证 74LS151 的功能。

输入				输出	
数据选择			选通	Y	W
C	B	A	G		
X	X	X	1	01	
000			0	D0	/D0
001			0	D1	/D1
010			0	D2	/D2
011			0	D3	/D3
100			0	D4	/D4
101			0	D5	/D5
110			0	D6	/D6
111			0	D7	/D7

表 7-2: 74LS151 功能表(“/”表示逻辑取反)

### 3.2 序列信号发生器

在本实验中读者将看到一个序列信号发生器是如何工作的，并尝试产生不同的序列信号。

1. 打开项目下的原理图文件 happen.dlsche。在此原理图中提供了一个使用二-五进制计数器 74LS90（参见附录 C）和 8 选 1 数据选择器 74LS151 组成的序列信号发生器，通过为 74LS151 输入端提供的信号来产生“10110”序列信号。
2. 启动仿真，打开逻辑分析仪观察序列信号的波形，验证序列信号发生器的功能。
3. 修改为 74LS151 输入端提供的信号来产生“01110”序列信号。

### 3.3 设计序列信号检测器

接下来，读者需要在之前学习的序列信号发生器的基础上使用 D 触发器设计一个序列信号检测器：

1. 在项目下新建一个原理图文件 check.dlsche。
2. 在参考本实验 3.2 节的基础上，先设计一个可以产生序列信号“11110”的序列信号发生器，然后使用 D 触发器设计一个序列信号检测器，当检测到序列信号“11110”时，点亮指示灯。在设计序列信号检测电路时，可以参考图 7-3，其中提供了序列信号“101”的检测电路。注意，序列信号发生器与检测器要使用同一个手动时钟。检测结果信号还可以接入一个计数器，并使用十六进制数码管显示出检测到的序列信号数量。
3. 仿真验证序列信号检测器的功能。

### 提交作业

在提交作业前，读者需要将序列信号检测器 SVG 图形文件的链接添加到 README.md 文件中，这样，当使用浏览器查看提交后的线上项目时，就可以从 README.md 文件中看到电路了。方法如下：

1. 双击“项目管理器”中的 README.md 文件，使用一种合适的文本编辑器或者 Markdown 编辑器打开此文件。
2. 在 README.md 文件的末尾添加如下的文本：  
# 序列信号检测器  
![raw svg](check.dlsche.svg)
3. 保存 README.md 文件。
4. 提交作业。

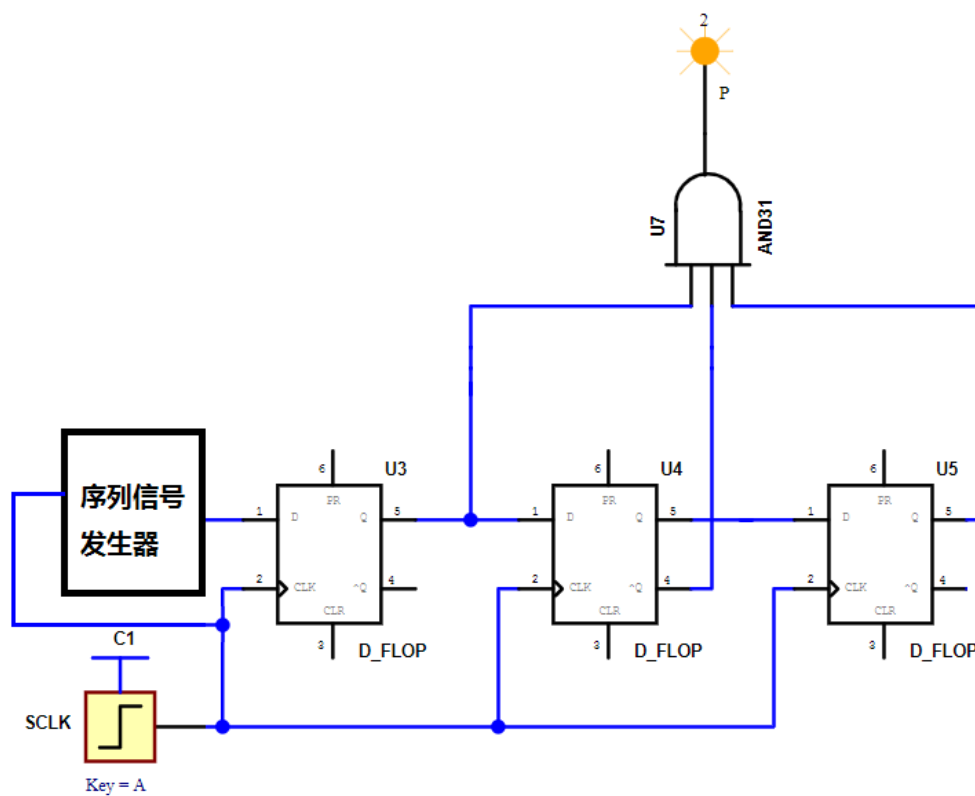


图 7-3：序列信号“101”的检测器

#### 四、思考与练习

1. 信号发生器根据结构不同, 可分为反馈移位型和计数型两种。自行查阅资料, 了解不同结构的信号发生器的特点。

# 实验 8 存储器

实验性质：验证+设计

建议学时：2 学时

## 一、实验目的

- 了解大规模集成存储器的工作原理。
- 熟悉大规模集成存储器的工作特性、使用方法以及应用。
- 熟悉有限状态机的相关理论知识，掌握有限状态机的设计方法。

## 二、预备知识

### 2.1 随机存取存储器

随机存取存储器可将数据随机地读出或写入存储器中任一位（或若干位），因而又称为读写存储器，简称 RAM。RAM 主要由地址译码器、存储矩阵、读/写控制器以及输入输出电路等组成，如图 8-1 所示。

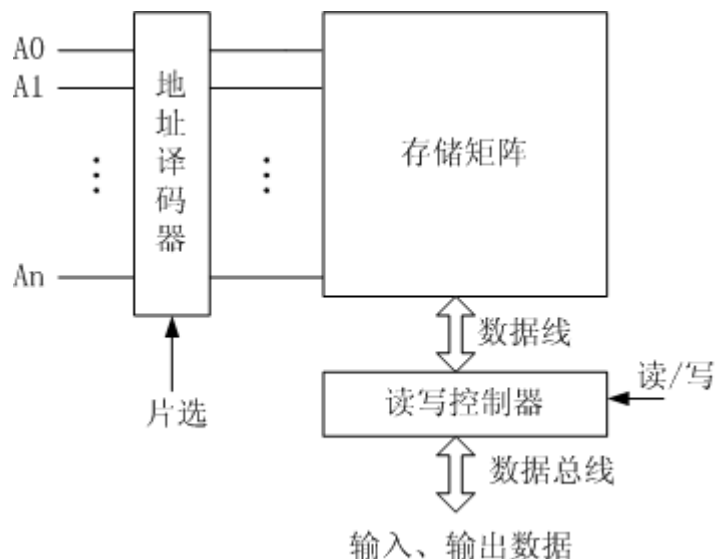


图 8-1: RAM 组成框图

其中，存储矩阵是 RAM 的核心部分，它由许多相同的存储元构成，并排列成矩阵结构形式。每一个存储元可以存放一位数据。若干个（如 8 个）存储元构成一个字，不同的字用不同的地址表示。该地址称为单元地址。

RAM 中的地址译码器用来对输入地址码进行译码，以确定需要访问的存储单元。读写控制器在读/写信号控制下控制数据的读出或写入。片选信号是在用若干个单片组成的存储体中，实现对各单片存储器工作的控制。输入输出（I/O）线是数据进出的通道，实际上是一线两用，它由读/写、片选信号控制。I/O 线的根数，取决于一个地址单元中包含的位数（叫字长）。例如容量为 1024 字×4 位，则每个地址单位有 4 个存储元（即字长为 4），故有 4 根 I/O 线。I/O 线一般具有三态输出结构。

常用的存储器芯片 2114 是一种 1024×4 位的静态随机存取存储器，采用 NMOS 工艺制作，各引出端功能如表 8-2 所示。

端名	功能
A9~A0	地址输入端
I/O4~I/O1	数据输入输出端
WR	写控制
CS	芯片选择

表 8-2:2114 芯片引出端功能

2114 的读/写访问:

- ① CS=0, WR=1 时读出当前输入地址指定存储器单元的内容, 读出不会破坏存储单元的内容。
- ② CS=0, WR=0 时将数据写入当前地址指定存储单元。

随机存取存储器是一种快速存取的存储器, 广泛应用于计算机或其他数字系统作主存储器使用, 通电后可以根据要求写入信息, 并在工作过程中能不断更改其存储内容。但一旦断电, 信息即全部消失。

## 2.2 RAM 的使用

在计算机中, RAM 是连接到总线上的, 各地址输入端所接的总线叫地址总线, 地址总线是单向传输的; 各数据输入输出端所接的总线叫数据总线, 数据总线是双向传输的, 通常用三态门选通; 各控制输入端所接的总线叫做控制总线。当某个设备要访问存储器时, 首先要申请到总线占用权; 然后把要访问单元的地址码, 通过地址寄存器送到地址总线; 再送片选信号和读/写信号, 同时打开挂在数据总线上的相应的三态门, 或向存储器存(写)数据, 或向存储器取(读)数据。

## 三、实验内容

请读者按照下面方法之一在本地创建一个项目, 用于完成本次任务:

### 方法一: 从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务, 从而在 CodeCode.net 平台上创建个人项目, 然后使用 Dream Logic 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

### 方法二: 不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台, 就需要使用 Dream Logic 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中, 实验模板的 URL

为 <https://www.codecode.net/engintime/Dream-Logic/Project-Template/Digital-Circuit/Lab008.git>

### 3.1 2114 存储器的读写

通过仿真, 对 2114 进行读写功能测试, 掌握 2114 的管脚功能和使用方法。实验步骤如下:

1. 打开项目下的原理图文件 2114.dlsche。在此电路中可以实现对 2114 的读写。由于 2114 的 D0~D3 既作为输入, 又作为输出, 所以需要通过三态门芯片 74LS244 完成对 2114 的读写控制, 避免发生冲突。具体的方法是在控制 2114 的 W 输入信号的同时, 也控制 74LS244 的哪一组三态门可以导通。
2. 启动仿真前, 将 2114 芯片的输入管脚 E 设为 0, W 设为 1, 输入地址设为 0, 准备读出 0 地址单元中存储的内容。
3. 启动仿真, 由于 2114 的初始化内存均为无效值, 所以从 0 地址单元读出的是一个无效值。

下面依次在 7~0 号存储单元中写入 7~0。步骤如下:

1. 将 74LS244 芯片的输入端 1A1~1A4 的值改为 07。
2. 将 2114 的地址端输入的值也改为 07。
3. 将 2114 的输入管脚 W 设为低电平。此时将十进制数 7 写入地址 7 对应的存储单元中。
4. 先将 2114 的地址减 1 (地址设为 6), 再将输入数据改为 6。此时将数据 6 写入地址为 6 的存储单元。
5. 仿照上面的步骤, 将 7~0 依次写入地址为 7~0 的 8 个存储单元。

6. 数据写完后，将 W 设为高电平，依次读出地址为 7~0 的 8 个存储单元。为了方便显示从存储器中读取到的值，读者可以将 74LS244 的输出端 2Y1~2Y4 连接一个 16 进制 7 段数码管。

### 3.2 使用存储器实现码组变换

在项目下新建一个原理图文件 transform.dlsche，在其中设计一个 4 位二进制自然码到格雷码的码组变换电路。格雷码如表 8-3 所示。

4 位二进制自然码	格雷码	十进制数
0000	0000	0
0001	0001	1
0010	0011	2
0011	0010	3
0100	0110	4
0101	1110	5
0110	1010	6
0111	1000	7
1000	1100	8
1001	1101	9

表 8-3：二进制自然码与格雷码对照表

**提示：**

将 4 位二进制码作为存储器的地址，而将其对应的格雷码保存在该地址单元中。这样，4 位二进制码就与存储器中对应位置的格雷码建立了映射关系，通过地址值访问存储器得到的内容就是格雷码。在测试电路时，首先需要将 4 位二进制码对应的格雷码依次写入地址为 0~9 的存储器单元中，然后尝试读出存储器中的值，验证读出的值是否是 4 位地址值对应的格雷码。

**提交作业**

在提交作业前，读者需要将码组变换电路 SVG 图形文件的链接添加到 README.md 文件中，这样，当使用浏览器查看提交后的线上项目时，就可以从 README.md 文件中看到电路了。方法如下：

1. 双击“项目管理器”中的 README.md 文件，使用一种合适的文本编辑器或者 Markdown 编辑器打开此文件。
2. 在 README.md 文件的末尾添加如下的文本：  

```
# 码组变换
![raw svg](transform.dlsche.svg)
```
3. 保存 README.md 文件。
4. 提交作业。

## 四、思考与练习

1. 自行查阅资料，了解计算机内部存储器的相关知识，掌握每种存储器的优缺点。
2. 对一块 2114，每单元地址的内容为一个字，字长是 4 位 (bit)。若要进行位扩展，字长变为 8 位，应如何连接和操作（使用多块 2114）。要想把单元数增加一倍，这叫字扩展（字数扩展），应如何连接。
3. 使用 2114 和 D 触发器设计一个格雷码加法计数器。自行查阅关于有限状态机的书籍，了解有限状态机的概念以及设计方法。格雷码加法计数器的工作原理是：在时钟脉冲的驱动下，一种 4 位格雷码转换为另一种 4 位格雷码。实质上是一个有限状态机。格雷码计数器状态转换图如图 8-4 所示。将 4 位格雷码依次写入存储器地址 0~15 单元中，使用 4 个 D 触发器输出作为存储器地址。而这四个 D 触发器的输入（也就是加计数以后所得下一个格雷码的地址）由当前存储器输出（格雷码）决定。

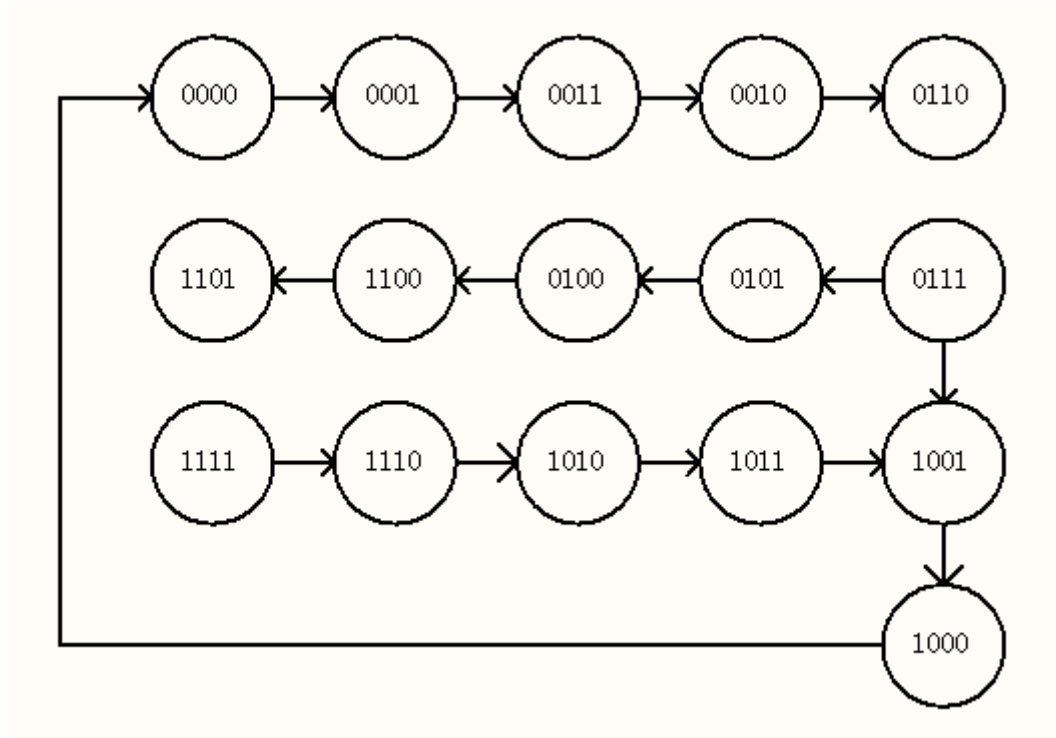


图 8-4: 格雷码计数器状态转换图



# 实验 9 密码电子锁的设计

**实验性质：**综合设计

**建议学时：**2 学时

**难易程度：**简单

## 一、实验目的

- 理解密码电子锁电路的基本原理。
- 进一步熟悉 D 触发器的基本功能。

## 二、预备知识

密码电子锁一般预先设定密码，用每个码位去控制触发器翻转，若码位按错则码位触发器不能翻转。在设计密码电子锁时，可以用密码依次控制各位 D 触发器的翻转，达到密码开锁的目的。

例如，可以使用 4 个 D 触发器串行连接，构成四位密码电路。S0~S9 代表键盘上的按键 0~9。当密码为 1469 时，S1、S4、S6、S9 分别是 1、4、6、9 四位密码的按钮端（可以使用单周期时钟控制），并分别接 4 个 D 触发器的时钟端。平时 4 个 D 触发器的 CLK 皆保持高电平状态，触发器保持原状态不变。当按下并抬起 S1 后，Q1=D1=1；再按下并抬起 S4 后，Q2=D2=Q1=1；同理，按下并抬起 S6 后，Q3=D3=Q2=1；按下并抬起 S9 后，Q4=D4=Q3=1，用此 Q4=1 去控制开锁机构即可（可以接一个指示灯，灯亮表示开锁）。

## 三、实验内容

请读者按照下面方法之一在本地创建一个项目，用于完成本次任务：

### 方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建个人项目，然后使用 Dream Logic 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

### 方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，就需要使用 Dream Logic 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的 URL

为 <https://www.codecode.net/engintime/Dream-Logic/Project-Template/blank.git>

参考预备知识中的内容，在原理图文件 top.dlsche 中设计一个 4 位密码电子锁。要求密码初始为“1469”，设计完成后仿真测试密码电子锁的功能。

### 改进设计

1. 成功开锁后，密码电子锁不应一直处于开锁状态，通常开锁信号触发开锁机构后，保持一段时间就会复位。设计一个自动复位电路，当密码电子锁成功开锁一段时间后，能够自动完成复位功能。
2. 使用四个 16 进制 7 段数码管依次显示用户输入的 4 个密码，复位后清零。
3. 想一想如何让用户可以修改预设的密码。

# 实验 10 七人表决器的设计

**实验性质：**综合设计

**建议学时：**2 学时

**难易程度：**简单

## 一、实验目的

- 熟悉组合电路的设计方法。

## 二、实验内容

多数表决器是对于一个行为由多个人投票，如果同意的票数多于反对的票数，则认为此行为有效通过。反之，则无效。要求读者使用四位二进制超前进位全加器 74LS283（参见附录 C），设计一个七人多数表决器（不允许弃权），并使用数码管将表决中赞成的票数显示出来。可以使用一位交互式数字信号作为表决按键，输入 0 表示反对，输入 1 表示赞成。按照下面的步骤完成电路设计：

请读者按照下面方法之一在本地创建一个项目，用于完成本次任务：

### 方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建个人项目，然后使用 Dream Logic 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

### 方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，就需要使用 Dream Logic 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的 URL

为 <https://www.codecode.net/engintime/Dream-Logic/Project-Template/Digital-Circuit/Lab010.git>

1. 打开项目下的原理图文件 74LS283.dlsche。该原理图文件中绘制了四位二进制超前进位全加器 74LS283 的内部原理图。
2. 启动仿真，根据表 10-1 验证 74LS283 的功能。
3. 在项目下新建一个原理图文件 7vote.dlsche，使用 3 个 74LS283 设计一个七人多数表决器。

输入端			输出端	
加数	被加数	进位输入	和	进位输出
A	B	Ci-1	S	Ci
0000	0000	0	0000	0
0000	0000	1	0001	0
0000	1111	0	1111	0
0000	1111	1	0000	1
1111	0000	0	1111	0
1111	0000	1	0000	1
1111	1111	0	1110	1
1111	1111	1	1111	1

表 10-1：74LS283 功能表

### 提交作业

在提交作业前，读者需要将七人多数表决器 SVG 图形文件的链接添加到 README.md 文件中，这样，当北京英真时代科技有限公司 <http://www.engintime.com>

使用浏览器查看提交后的线上项目时，就可以从 README.md 文件中看到电路了。方法如下：

1. 双击“项目管理器”中的 README.md 文件，使用一种合适的文本编辑器或者 Markdown 编辑器打开此文件。
2. 在 README.md 文件的末尾添加如下的文本：  
`# 七人多数表决器`  
`![raw svg](7vote.dlsche.svg)`
3. 保存 README.md 文件。
4. 提交作业。

# 实验 11 智力竞赛抢答器的设计

实验性质：综合设计

建议学时：2 学时

难易程度：一般

## 一、实验目的

- 了解一个数字系统的基本组成及它的控制电路的设计考虑。
- 熟悉数字系统模块化的设计方法。
- 熟悉集成芯片的综合应用。
- 学习如何用实验的方法来完善理论设计以及用实验的方法来验证电路模块的输入、输出逻辑。
- 熟悉数字系统逐级仿真调试的方法，从子模块到数字系统的功能验证过程，也是自底向上设计实现过程。

## 二、预备知识

抢答器在各类竞赛性质的场合得到了广泛的应用，它的出现消除了原来由于人眼的误差而未能正确判断最先抢答的人的情况。

抢答器的原理比较简单，首先必须设置一个抢答允许标志位，目的就是为了让允许或者禁止抢答者按下抢答按钮；如果抢答允许位有效，抢答器开始工作，4 个抢答者谁先按下抢答按钮，则谁抢答成功，同时记录按钮的序号，这样做的目的就是为了禁止后面再有人按下按钮。总的来说，抢答器的实现原理就是在抢答允许位有效后，第一个按下按钮的人将禁止再有按钮按下，同时显示抢答者的序号。

## 三、实验内容

### 3.1 学习绘制模块化的原理图

为了使原理图更加简洁、明了，本实验将引入模块的概念，Dream Logic 支持自定义模块的功能，可以将绘制的部分原理图封装成一个模块供使用，大大提高了原理图的识图效率，还可以提供模块的复用功能。

请读者使用Dream Logic提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的URL为 <https://www.codecode.net/engintime/Dream-Logic/Project-Template/blank.git>

1. 新建一个原理图文件 child.dlsche。此原理图将作为子模块的内部原理图。
2. 打开原理图文件 child.dlsche，放置一个非门。
3. 点击菜单栏中的“绘制->端口”，将两个端口分别和非门的输入端和输出端连接，放置时注意连接正确。如图 11-1 所示。注意，这里使用了端口锚点与器件管脚锚点直接连接的方法，读者也可以将端口放置在距离管脚一定距离的地方，然后使用网络完成连接。

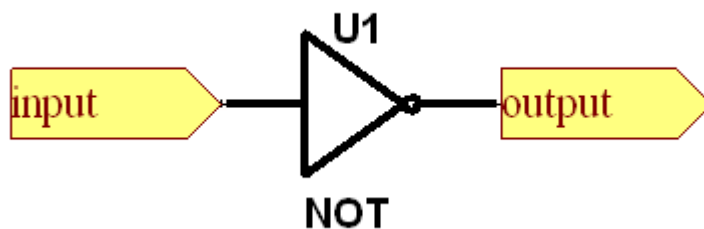


图 11-1：添加端口

4. 修改端口名称。选中端口，在属性窗口中将非门输入端口的名称修改为 input，输出端口的名称修改为 output。保存文件。
5. 打开原理图文件 top.dlsche，点击菜单栏中的“绘制->模块”，在原理图中拖放至所需大小。
6. 添加模块接口。点击菜单栏中的“绘制->模块接口”，根据 child.dlsche 中的输入端口和输出端口的数量在模块上放置相同数量的模块接口。通常输入接口放置在左侧，输出接口放置在右侧。
7. 修改模块接口名称。选中模块接口，将其名称修改为与原理图 child.dlsche 中的端口相对应的名称，这样就表示 top.dlsche 中的模块接口与 child.dlsche 中的端口相连接。也就是说相同名称的模块接口和子模块中的端口会连接在一起。
8. 为模块匹配原理图文件。在原理图文件 top.dlsche 中选中模块，在属性窗口中的“模块原理图文件”属性中选择 child.dlsche，这样就完成了模块与其封装电路的匹配。完成后的原理图文件 top.dlsche 如图 11-2 所示。

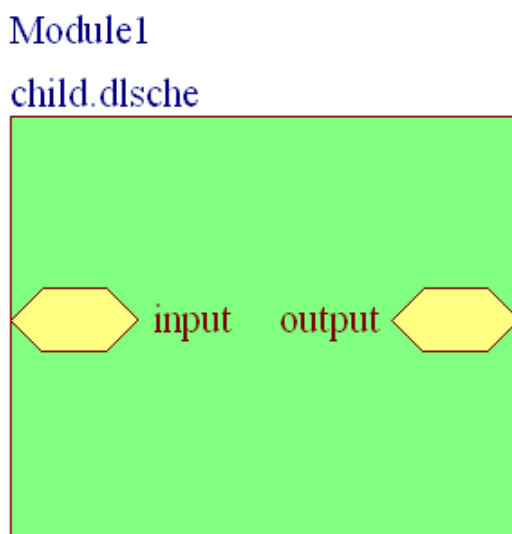


图 11-2：完成封装的模块

9. 完成封装后的模块即可添加元器件测试模块的功能了。由于模块内部只有一个非门，因此本模块的功能是完成高低电平的转换。测试电路如图 11-3 所示。

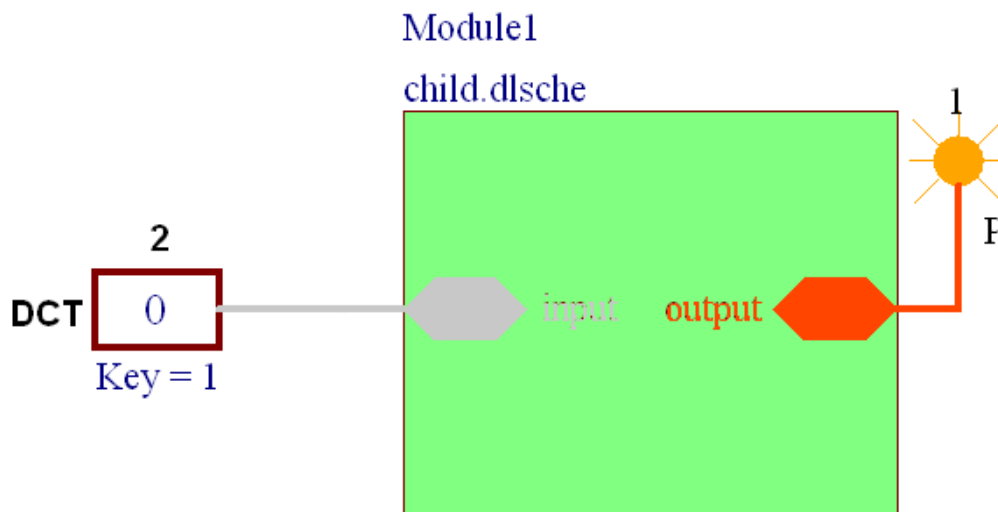


图 11-3：测试电路

10. 在仿真过程中如果想查看模块内部原理图 child.dlsche 中的状态，双击模块即可。若想从 child.dlsche 中返回到上层原理图 top.dlsche，在原理图 child.dlsche 的空白处双击即可。
11. 停止仿真，修改模块接口名称，将 input 修改为 input-err 后保存。

12. 重新启动仿真。此时仿真失败，在错误列表窗口中会显示出错误信息以及错误所在的原理图文件，双击错误信息，可直接定位到错误所在位置。这样就可以帮助用户找到那些由于模块接口和端口名称不对应而产生的错误。

13. 根据提示信息修改错误，将模块接口名称修改回 input，再次仿真后结果正确。

在后面的实验中为了使原理图更加简洁、清晰，读者可以尝试使用模块功能。以上内容仅是演示如何封装模块，若要添加更多的模块端口，应根据所需的数量灵活设置模块大小与模块接口的布局。

**注意**，内部网络名称可以与外部网络名称重名，但是并没有电气连接关系。内部网络只能通过端口与模块接口连接，从而与外部电路连接。

### 3.2 设计智力竞赛抢答器

请读者按照下面方法之一在本地创建一个项目，用于完成本次任务：

#### 方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建个人项目，然后使用 Dream Logic 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

#### 方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，就需要使用 Dream Logic 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的 URL

为 <https://www.codecode.net/engintime/Dream-Logic/Project-Template/blank.git>

### 设计要求

(1) 按下“启动”按钮后才允许抢答，30s 倒计时计时器开始工作，并在数码管上显示出倒计时时间（设计简单计时器的方法可以参考本书数字电路实验 12 中的相关内容）。

(2) 抢答按钮分为 8 组，即序号 1、2、3、4、5、6、7、8，抢答者按本组序号的抢答按钮进行抢答，抢中的组号立即锁存，并显示到数码管上，同时封锁其它组号的抢答功能，30s 倒计时计时器停止计时。

(3) 30 秒定时到，如果没有抢答者，本次抢答无效，系统自动复位。

(4) 提供手动复位功能，复位后抢答号码、抢答计时等清零。

### 提交作业

在提交作业前，读者需要将所有原理图 SVG 图形文件的链接添加到 README.md 文件中，这样，当使用浏览器查看提交后的线上项目时，就可以从 README.md 文件中看到电路了。

# 实验 12 路口交通灯管理系统的设计

实验性质：综合设计

建议学时：2 学时

难易程度：较难

## 一、实验目的

- 熟悉有限状态机的设计和调试。
- 熟练运用集成芯片设计电路。
- 进一步训练数字系统的设计和调试。

## 二、预备知识

### 2.1 同步时序电路

组合逻辑电路中是没有环路和竞争的。在组合逻辑电路中，输出总是根据输入，在传播延迟内稳定为一个正确的值。但是，将组合电路的输出直接反馈到输入，形成环路，就成为了时序电路，而不再是组合电路。包含环路的时序电路存在不良的竞争和不稳定的行为。分析这样的电路十分耗时，而且很容易犯错。

为了避免这些问题，设计师在环路中插入寄存器来断开环路。这样，就将环路转变为组合逻辑电路和寄存器的集合。寄存器包含系统的状态，这些寄存器的内容仅仅在时钟上升沿或下降沿到来时才发生改变，所以说状态同步于时钟信号。如果时钟足够慢，使得在下一个时钟沿到达前输入到寄存器的信号都可以稳定下来，那么所有的竞争将被消除。根据反馈环路上总是使用寄存器的原则，可以得到同步时序电路的一个形式化定义。非同步的时序电路称为异步电路。

通过电路的输入、输出、功能规范和时序规范可以定义一个电路。一个时序电路有一组有限的离散状态  $\{S_0, S_1, \dots, S_{k-1}\}$ 。同步时序电路有一个时钟输入，它的上升沿表示时序电路状态转变发生的时间。我们经常使用术语当前状态和下一个状态来区分目前系统状态和下一个时钟沿系统进入的状态。

同步时序电路要满足的条件：

- 每一个电路元件都是寄存器或者组合电路。
- 至少有一个电路元件是寄存器。
- 所有寄存器都接收同一个时钟信号。
- 每个环路至少包含一个寄存器。

两种常见的同步时序电路是有限状态机和流水线。

### 2.2 有限状态机

同步时序电路可以用图 12-1 中的形式来描述。这些形式称为有限状态机(Finite State Machine, FSM)。这个名字源于具有  $k$  位寄存器的电路可以处于  $2^k$  种状态（每个寄存器值表示一种状态）中的某一种唯一状态。一个有限状态机有  $M$  个输入、 $N$  个输出和  $k$  位状态。它还接收一个时钟信号和一个可选择的复位信号。有限状态机包含两个组合逻辑块，下一个状态逻辑和输出逻辑，以及一个存储这个状态的寄存器。在时钟沿到来时，有限状态机进入下一个状态，这个下一个状态是根据当前状态和输入计算出来的。根据有限状态机功能规范的描述，FSM 通常分为两类。在 Moore 型有限状态机中，输出仅仅取决于机器的当前状态。在 Mealy 型有限状态机中，输出取决于当前状态和当前输入。



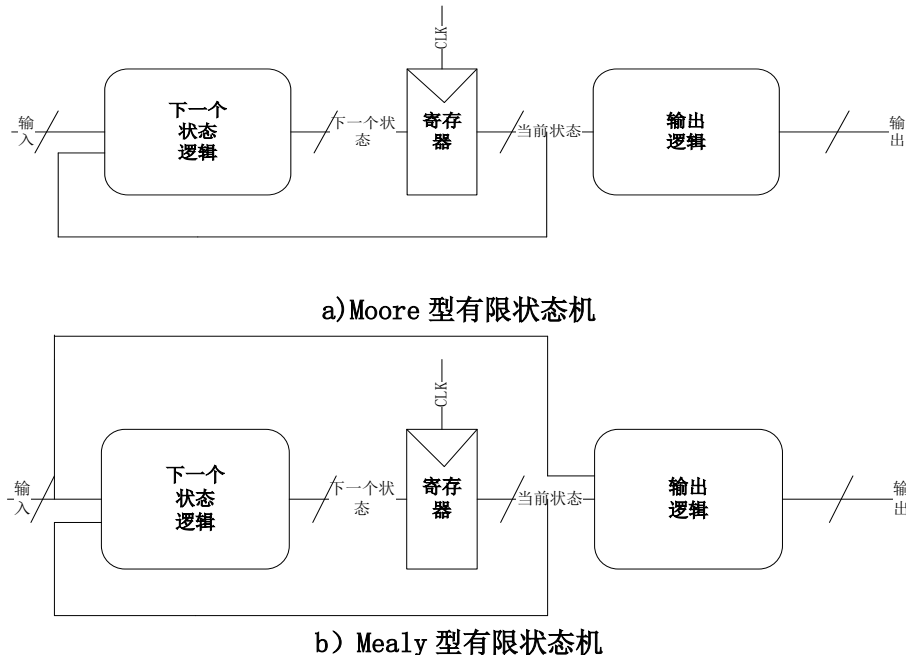


图 12-1：有限状态机

## 2.3 状态编码

### a) 二进制编码

一个二进制数代表一种状态，因为  $\log_2 K$  位可以表示  $K$  个不同的二进制数，所以有  $K$  个状态的系统只需要  $\log_2 K$  位编码。

### b) 独热编码

在独热编码中，状态的每一位表示一种状态。例如，一个有 3 个状态的有限状态机的独热编码为 001、010 和 100。状态的每一位存储在一个触发器中，所以独热编码比二进制编码需要更多的触发器。然而，使用独热编码，下一个状态和输出逻辑通常更简单，需要的门电路也更少。最佳编码方式取决于具体的有限状态机。

## 2.4 设计有限状态机的步骤

1. 确定输入和输出。
2. 画出状态转换图。
3. 写出状态转换表。
4. 选择状态编码。
5. 为下一个状态和输出逻辑写出布尔表达式。
6. 根据布尔表达式画出有限状态机的电路图。

# 三、 实验内容

在设计复杂的数字系统之前，读者应该掌握使用 Dream Logic 绘制模块化电路的方法，这部分内容请参考本书数字电路实验 11 中的相关内容。

## 3.1 设计简单的计时器

请读者使用 Dream Logic 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的 URL

为 <https://www.codecode.net/engintime/Dream-Logic/Project-Template/Digital-Circuit/Lab012.git>

1. 打开项目下的原理图文件 74LS190.dlsche。该原理图文件中绘制了异步置数十进制加/减计数器 74LS190 的内部原理图。
2. 启动仿真，根据表 12-1 完成 74LS190 的功能验证。

CLK	S	LD	U/D	工作状态
X	1	1	X	保持
X	X	0	X	预置数
↑	0	1	0	加法计数
↑	0	1	1	减法计数

表 12-1: 74LS190 功能表

3. 在项目下新建一个原理图文件 clock.dlsche。在其中使用 74LS190 和触发器等器件设计一个倒计时计时器。倒计时计数器从 35 开始减计数，当计数到 0 时，下个时钟沿又使计时器从 35 开始新一轮的倒计时。需要为计时器提供一个复位信号，任何时刻，复位信号变为 0 都会使计时器重新从 35 开始倒计时。仿真验证时，可以先使用手动的单周期时钟驱动计时器，仿真成功后，可以尝试使用自动时钟驱动计时器，设置自动时钟频率的方法可以参考下面的注意事项。

**注意**，需要灵活设置自动时钟的频率以及原理图的仿真步长与仿真时间。

- 选中用于驱动计时器的自动时钟器件，在属性窗口中将时钟频率设置为 1，再启动仿真后，可以观察到计时器上的时间与仿真时间保持一致。仿真时间显示在 Dream Logic 软件底部的状态栏中。
- 点击将要进行仿真的原理图的空白处，在属性窗口中会显示原理图的仿真参数。设置仿真步长为 100ms，可以使仿真进行的速度比较适中。设置仿真时间为 5000s，可以保证仿真在一段足够长的时间内进行，方便观察仿真结果。

### 3.2 设计路口交通灯管理系统

请读者按照下面方法之一在本地创建一个项目，用于完成本次任务：

#### 方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建个人项目，然后使用 Dream Logic 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

#### 方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，就需要使用 Dream Logic 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的 URL

为 <https://www.codecode.net/engintime/Dream-Logic/Project-Template/blank.git>

#### 1. 要求如下：

- 1) 实现东西、南北两个方向上的交通灯控制。使用型号库“数字信号显示”中提供的交通灯显示灯光变化。
- 2) 灯光变化的规律为：红灯亮→绿灯亮→黄灯亮→红灯亮……，如此依次循环。
- 3) 使用两个数码管显示一个方向上的倒计时，红灯持续时间为 7 秒、绿灯持续时间为 5 秒、黄灯持续时间为 2 秒。
- 4) 当任何一个方向出现警车、消防车、救护车等优先通行类车辆时，则东西、南北两个方向上的红灯全亮，其他灯灭，时钟停止计时，其他车辆禁止通行。当这种特殊状态结束后，恢复原来的状态，继续运行。
- 5) 使用单周期时钟器件产生的负脉冲（低电平）表示有紧急事件发生，优先类车辆通行；负脉冲结束表示紧急事件结束。
- 6) 使用有限状态机的方法设计交通灯控制器。

#### 2. 交通灯设计提示。

- (1) 确定输入和输出。

输入：

- 时钟 CLK，倒计时时钟源。
- 复位按钮 RESET，按下复位按钮后，南北方向为红灯（从 7 秒开始倒计时），东西方向为绿灯（从 5 秒开始倒计时）。
- 紧急事件按钮 Emergency。按下变为低电平，两个方向均为红灯。松开变为高电平，红绿灯恢复为初始状态继续工作。

输出：

- 南北方向的红、绿、黄灯控制信号 LRsn、LGsn、LYsn，高电平对应灯亮，以及倒计时 sn0、sn1、sn2，显示倒计时 0~7 只需 3 位二进制码。
- 东西方向的红、绿、黄灯控制信号 LRew、LGew、LYew，高电平对应灯亮，以及倒计时 ew0、ew1、ew2。

(2) 画出状态转换图，如图 12-2 所示，从图中可以看出，交通灯共有 14 个不同的状态。

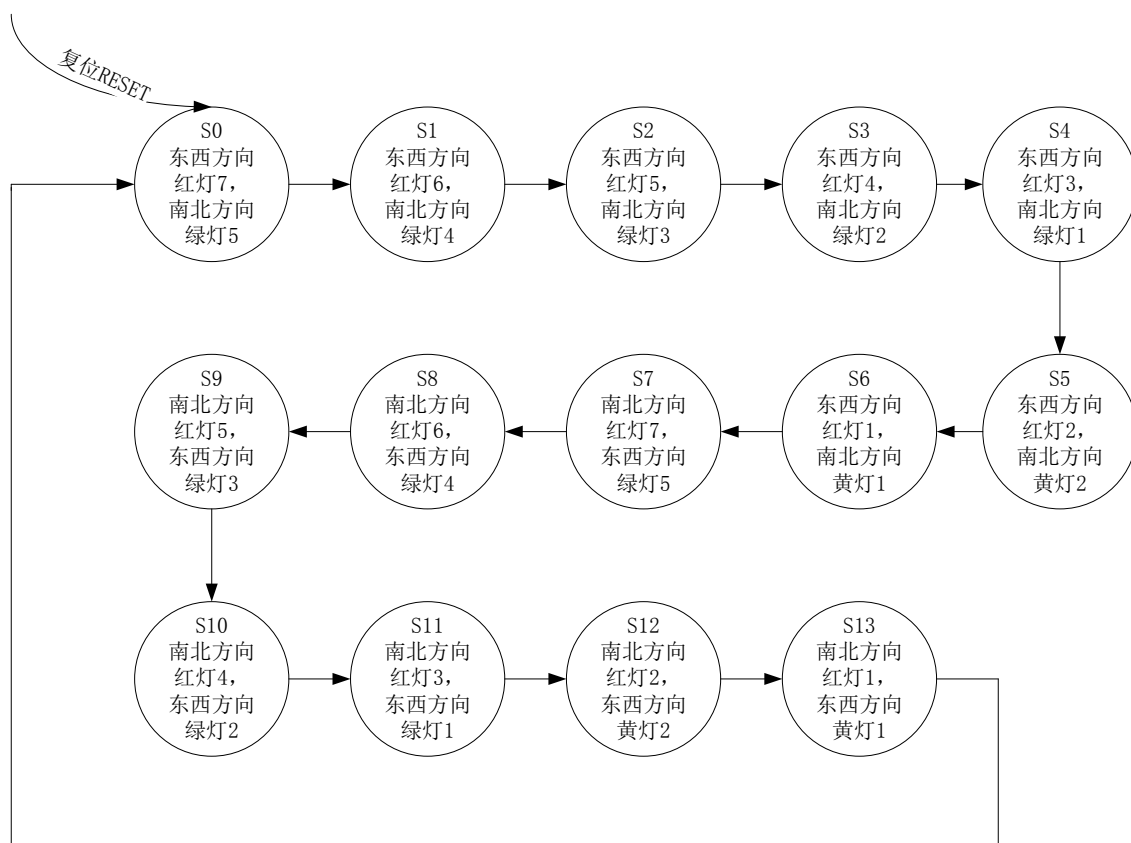


图 12-2：交通灯状态转换图

(3) 状态编码。

使用二进制编码，14 个状态需要至少 4 位二进制编码。使用  $T_3$ 、 $T_2$ 、 $T_1$ 、 $T_0$  四位编码状态。

(4) 写出状态转换表，如表 12-2 所示。

当前状态	当前状态编码				下一个状态	下一个状态编码			
	$T_3T_2T_1T_0$					$T_3'T_2'T_1'T_0'$			
S0	0	0	0	0	S1	0	0	0	1
S1	0	0	0	1	S2	0	0	1	0
S2	0	0	1	0	S3	0	0	1	1
S3	0	0	1	1	S4	0	1	0	0
S4	0	1	0	0	S5	0	1	0	1
S5	0	1	0	1	S6	0	1	1	0
S6	0	1	1	0	S7	0	1	1	1
S7	0	1	1	1	S8	1	0	0	0
S8	1	0	0	0	S9	1	0	0	1
S9	1	0	0	1	S10	1	0	1	0
S10	1	0	1	0	S11	1	0	1	1
S11	1	0	1	1	S12	1	1	0	0
S12	1	1	0	0	S13	1	1	0	1
S13	1	1	0	1	S0	0	0	0	0

表 12-2: 交通灯状态转换表

(5) 为下一个状态逻辑写出布尔表达式

写下一个状态逻辑的布尔表达式时, 不需要考虑时钟, 只需要考虑当前状态和输入对下一个状态的影响。在交通灯控制器中, 对下一个状态有影响的外部输入是复位信号 RESET 和紧急事件 Emergency, 无论当前处于何种状态, 一旦有复位信号 RESET, 下一个状态一定是 S0 (初始状态)。然后分析紧急事件信号的影响, 紧急事件结束后, 仍然恢复为原来的状态, 也就说明, 紧急事件不会影响下一个状态, 它是通过使交通灯控制器的当前状态保持不变, 停止状态转换, 而仅仅使交通灯暂时维持红灯而已。所以, 下一个状态的布尔表达式为以下样式:

$T_3' = \text{RESET} (T_3, T_2, T_1, T_0 \text{ 的逻辑表达式})$

$T_2' = \text{RESET} (T_3, T_2, T_1, T_0 \text{ 的逻辑表达式})$

$T_1' = \text{RESET} (T_3, T_2, T_1, T_0 \text{ 的逻辑表达式})$

$T_0' = \text{RESET} (T_3, T_2, T_1, T_0 \text{ 的逻辑表达式})$

请读者根据状态转化表完成下一个状态的布尔表达式。

提示: 从交通灯控制器的状态转换图可以看出, 交通灯的状态从 S0 开始, 依次转换为 S1, S2, S3, ..., S13, 然后又转变为 S0。状态每经过一秒转换一次, 正好是一个带异步置数端的 14 进制计数器。计数器的 4 位输出就是当前状态码  $T_3, T_2, T_1, T_0$ , 根据当前状态码, 就可以得到当前输出。

(6) 画出电路图

交通灯的逻辑设计过程已经完成, 请读者自行完成交通灯电路, 通过仿真验证。

### 提交作业

在提交作业前, 读者需要将所有原理图 SVG 图形文件的链接添加到 README.md 文件中, 这样, 当使用浏览器查看提交后的线上项目时, 就可以从 README.md 文件中看到电路了。

# 实验 13 电梯控制器的设计

**实验性质：**综合设计

**建议学时：**2 学时

**难易程度：**较难

## 一、实验目的

- 进一步熟悉有限状态机的设计方法。
- 进一步熟悉一个数字系统的设计和仿真调试方法。

## 二、实验内容

在设计复杂的数字系统之前，读者应该掌握使用 Dream Logic 绘制模块化电路的方法，这部分内容请参考本书数字电路实验 11 中的相关内容。此外本书数字电路实验 12 中介绍了同步时序电路、有限状态机的概念，还提供了计时器的设计方法。

请读者按照下面方法之一在本地创建一个项目，用于完成本次任务：

### 方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建个人项目，然后使用 Dream Logic 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

### 方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，就需要使用 Dream Logic 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的 URL

为 <https://www.codecode.net/engintime/Dream-Logic/Project-Template/blank.git>

## 设计要求

设计一个三层楼房全自动电梯控制电路，该电路具有如下功能：

1. 每层电梯入口设有上下请求开关各一个（最底层只有向上请求开关，最高层只有向下请求开关），电梯内设有乘客到达层数的停站要求开关。
2. 电梯所处层数指示装置和电梯上下行状态指示装置。
3. 电梯每 3 秒升（降）一层，到达某一层时，数码管显示该层层数，并一直保存到电梯到达新一层为止。
4. 电梯到达有停站要求的层后，经过 1 秒，电梯门自动打开（开门指示灯亮），经过 5 秒后，电梯门自动关闭（开门指示灯灭）。
5. 能保证响应电梯内外的所有请求信号，并按照电梯运行规则依次响应，每个请求信号保留至执行后撤除。
6. 电梯运行规则：电梯处于上升模式时，只响应比所在位置高的楼层的上楼请求信息，由上而下逐个执行直到最后一个请求执行完毕。如更高层有下楼请求，则直接升到有下楼请求的最高层接客，然后转入下降模式。电梯处于下降模式时与之相反，仅响应比电梯所在位置低的下楼请求，由上到下逐个解决，直到最后一个请求被处理完毕。如果最底层有上楼请求时，则降至该层楼，并转入上升模式。电梯执行完所有的请求后，应停在最后所在位置不变，等待新的请求。
7. 开机时，电梯应停在一楼，而各种上下请求均被清除。

## 系统设计方案

1. 三层电梯控制面板如图 13-1 所示。在设计中，可以灵活使用单周期时钟、数码管、指示灯等器件代

替控制面板中的按钮。

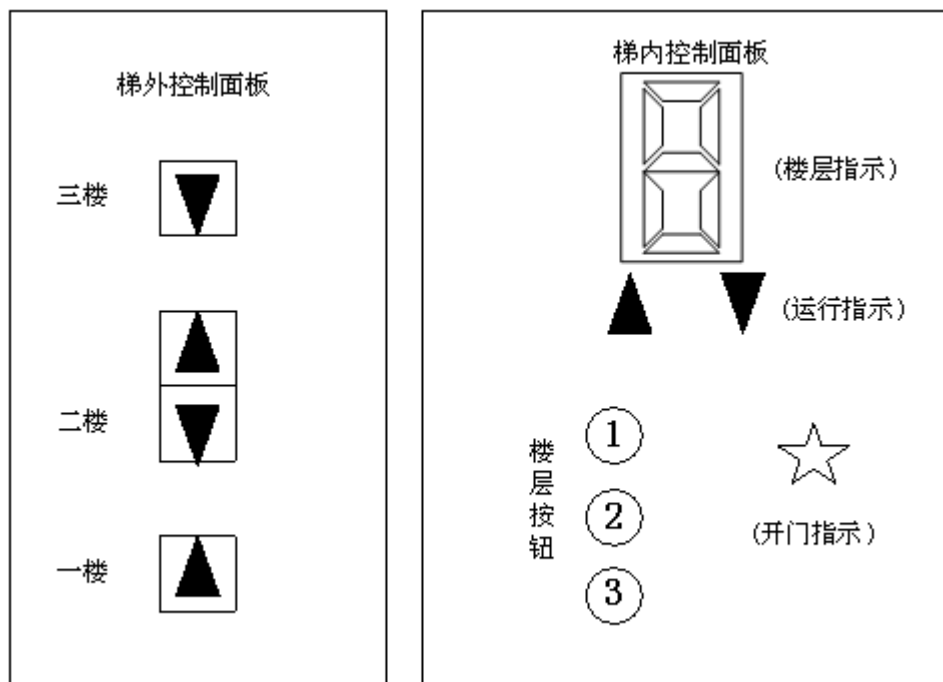


图 13-1：电梯控制面板

2. 设计框图如图 13-2 所示。

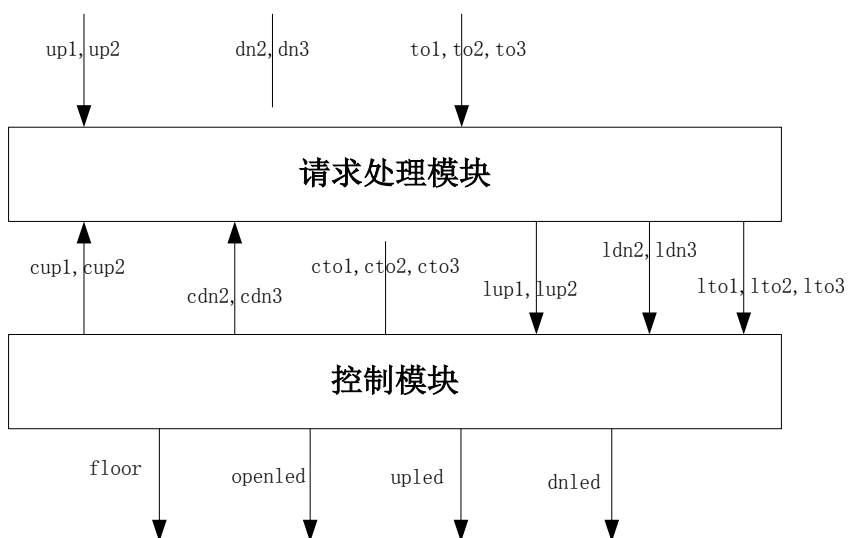


图 13-2：设计框图

3. 主要模块设计

根据上面的设计方案，设计中应具有一些信号和模块。

1) 信号说明

up1~up2: 分别为电梯外 1、2 楼用户上楼请求信号。

dn2~dn3: 分别为电梯外 2、3 楼用户下楼请求信号。

to1~to3: 分别为电梯内用户到 1、2、3 楼的请求信号。

lup1~lup2: 分别为电梯外 1、2 楼用户上楼请求指示。

ldn2~ldn3: 分别为电梯外 2、3 楼用户下楼请求指示。

lto1~lto3: 分别为电梯内用户到 1、2、3 楼的请求指示。

cup1~cup2: 分别用于清除 1、2 楼用户的上楼请求。

cdn2~cdn3: 分别用于清除 2、3 楼用户的下楼请求。

cto1~cto3: 分别用于清除电梯内用户到 1、2、3 楼的请求。

floor: 楼层显示。

Openled: 开门指示。

Uplcd: 上升指示。

Dnled: 下降指示。

**注意:** 当请求信号被满足时, 请求指示清除, 否则, 请求指示一直有效。

## 2) 模块说明

**请求处理模块:** 存储用户的请求以及当请求被处理后请求指示的清除。

**控制模块:** 电梯到达有停站要求的楼层后, 经过 1 秒, 电梯门自动打开 (开门指示灯亮), 经过 5 秒后, 电梯门自动关闭 (开门指示灯灭), 电梯继续运行; 能保证响应电梯内外的所有请求信号, 并按照电梯运行规则依次响应, 每个请求信号保留至执行后撤除; 开机时, 电梯应停在一楼, 而各种上下请求均被清除。

## 4. 电梯有限状态机设计参考

以 1 秒为电梯状态转换的时间单位, 也就是每经过 1 秒, 电梯由当前状态进入下一个状态。以此为依据, 将电梯状态划分如下:

- 1) 电梯起始状态 stop: 关停, 电梯门关闭, 电梯停止升降。
- 2) 电梯开门等待 1 秒 open1。
- 3) 电梯开门等待 2 秒 open2。
- 4) 电梯开门等待 3 秒 open3。
- 5) 电梯开门等待 4 秒 open4。
- 6) 电梯开门等待 5 秒 open5。
- 7) 电梯上升 1 秒 up1: 电梯每上升一层需要 3 秒, 将上升过程中的每一秒作为一个状态。
- 8) 电梯上升 2 秒 up2。
- 9) 电梯上升 3 秒 up3。
- 10) 电梯下降 1 秒 dn1, 与电梯上升同理, 划分为 3 个不同状态。
- 11) 电梯下降 2 秒 dn2。
- 12) 电梯下降 3 秒 dn3。

**注意:** 以上列出是电梯的主要状态, 每一个主要状态依据当前所在楼层可以细分为多个状态, 比如 stop 状态, 可以分为电梯停在 1 楼 stop1, 电梯停在 2 楼 stop2, 电梯停在 3 楼 stop3 共 3 个不同的状态。

## 5. 参考状态转换图 13-3。



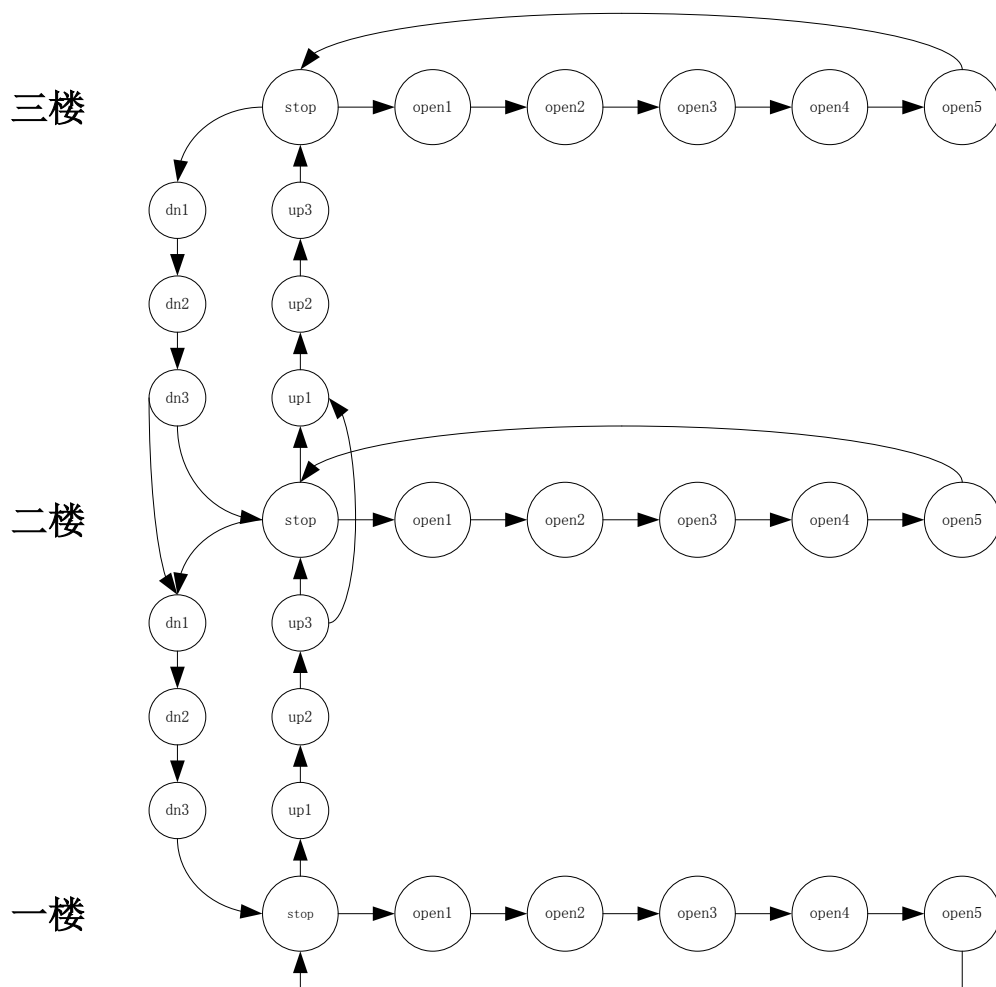


图 13-3：电梯状态转换图

注意：图中的每一个圆圈表示一个独立的状态，例如一楼的 stop 与二楼的 stop 表示两个不同的状态。此外，电梯控制器是一个 mealy 型有限状态机，下一个状态由当前状态以及当前的输入决定，请读者结合电梯运行规则，自行标出状态转换条件，也就是各个楼层的电梯请求。

6. 状态转换表（使用 5 位二进制编码所有状态）如表 13-1 所示。说明如下

- Lup1、lup2、ldn2、ldn3、lto1、lto2、lto3：1 或 /1 均表示有请求，0 表示无请求，空表示任意值。
- 当前状态与所有有效输入均为“与”的关系，当一行中有多个“/1”时，表示只要其中一个请求有效即可，例如第一行的状态转换逻辑可以描述为： $slopel = stopl(lup1 \mid lto1)$ ，含义是，当电梯处于停在 1 楼的状态时，若 1 楼有上楼请求或者有到 1 楼请求，则电梯开门并进入开门第一秒的状态。而

S1up3		0		/1		0	/1		0	1	1	1	1	S2up1
-------	--	---	--	----	--	---	----	--	---	---	---	---	---	-------

表示由当前状态 s1up3 进入下一个状态 s2up1。 $s2up1 = s1up3(\sim lup2)(\sim lto2)(ldn3 \mid lto3)$ ，含义是，当电梯处于 1 楼升 2 楼的第 3 秒状态时，若 2 楼没有上楼请求、没有到 2 楼的请求、3 楼有下楼请求或有到 3 楼的请求，则电梯到达 2 楼后不停止，继续上升，进入 2 楼上 3 楼的第一秒状态。

当前状态	Lup1	Lup2	Ldn2	Ldn3	Lto1	Lto2	Lto3	Up/dn	S4	S3	S2	S1	S0	下一状态
Stop1	/1				/1				0	0	0	0	1	S1open1
S1open1									0	0	0	1	0	S1open2
S1open2									0	0	0	1	1	S1open3
S1open3									0	0	1	0	0	S1open4
S1open4									0	0	1	0	1	S1open5
S1open5									0	0	0	0	0	Stop1
Stop1	0	/1	/1	/1	0	/1	/1		0	0	1	1	0	S1up1
S1up1									0	0	1	1	1	S1up2
S1up2									0	1	0	0	0	S1up3
S1up3		/1				/1			0	1	0	0	1	Stop2
S1up3		0		/1		0	/1		0	1	1	1	1	S2up1
Stop2						1			0	1	0	1	0	S2open1
Stop2		1						1	0	1	0	1	0	S2open1
Stop2			1					0	0	1	0	1	0	S2open1
S2open1									0	1	0	1	1	S2open2
S2open2									0	1	1	0	0	S2open3
S2open3									0	1	1	0	1	S2open4
S2open4									0	1	1	1	0	S2open5
S2open5									0	1	0	0	1	Stop2
Stop2		0		/1		0	/1	1	0	1	1	1	1	S2up1
S2up1									1	0	0	0	0	S2up2
S2up2									1	0	0	0	1	S2up3
S2up3									1	0	0	1	0	Stop3
Stop3				/1			/1		1	0	0	1	1	S3open1
S3open1									1	0	1	0	0	S3open2
S3open2									1	0	1	0	1	S3open3
S3open3									1	0	1	1	0	S3open4
S3open4									1	0	1	1	1	S3open5
S3open5									1	0	0	1	0	Stop3
Stop3	/1	/1	/1	0	/1	/1	0		1	1	0	0	0	S3dn1
S3dn1									1	1	0	0	1	S3dn2
S3dn2									1	1	0	1	0	S3dn3
S3dn3			/1			/1			0	1	0	0	1	Stop2
S3dn3	/1		0		/1	0			1	1	0	1	1	S2dn1
Stop2	/1		0		/1	0		0	1	1	0	1	1	S2dn1
Stop2	/1		0	0	/1	0	0	1	1	1	0	1	1	S2dn1
S2dn1									1	1	1	0	0	S2dn2
S2dn2									1	1	1	0	1	S2dn3
S2dn3									0	0	0	0	0	Stop1

表 13-1：电梯状态转换表

## 提交作业

在提交作业前，读者需要将所有原理图 SVG 图形文件的链接添加到 README.md 文件中，这样，当使用浏览器查看提交后的线上项目时，就可以从 README.md 文件中看到电路了。

# 实验 14 数字钟的设计

实验性质：综合设计

建议学时：2 学时

难易程度：一般

## 一、实验目的

- 学习数字电路系统的设计方法、及数字钟功能扩展电路的设计。
- 从电路角度出发，建立系统的概念，培养综合运用数字电路的能力。

## 二、预备知识

### 2.1 数字钟电路的组成

数字钟的原理如图 14-1 所示。

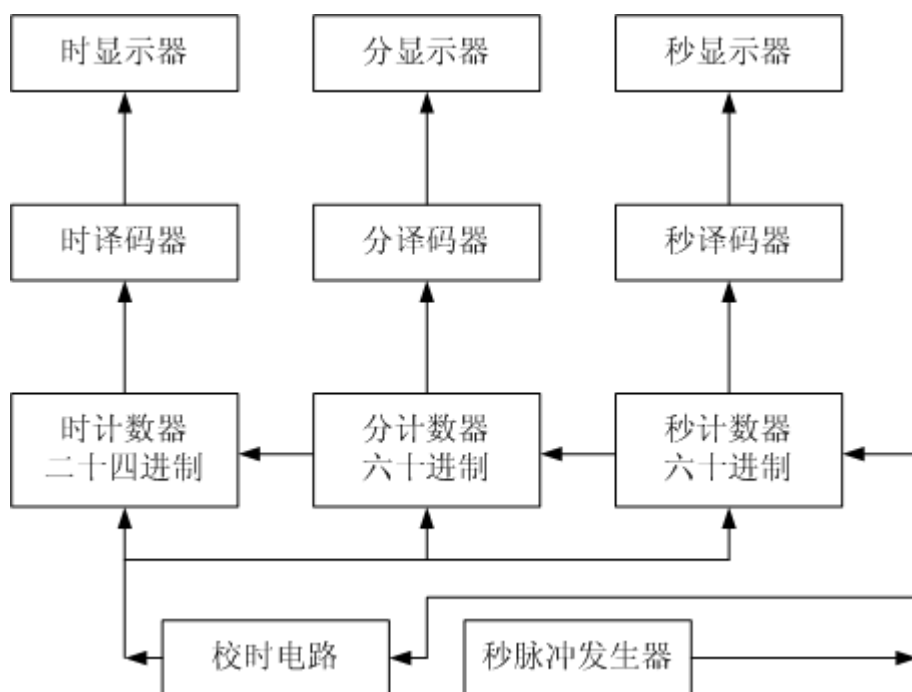


图 14-1：数字钟原理框图

在图 14-1 中，秒脉冲发生器是数字钟的核心，其工作频率应该为 1Hz，它产生的秒信号输入秒计数器进行计数，当达到 59 时产生进位信号给分计数器，秒计数器复位到 0 的同时，分计数器开始计时。当分计数器达到 59 时产生进位信号给时计数器，分计数器复位到 0 的同时，时计数器开始计数。

### 2.2 数字钟主体电路的设计

#### (1) 秒脉冲发生器。

使用型号库“数字信号源”中的自动时钟作为秒脉冲发生器，通过将时钟的频率设置为 1Hz，使其每隔一秒输出一个时钟上升沿，从而触发秒计数器计数。

#### (2) 计数器电路。

可用于计数的芯片很多，比如 74LS161，74LS90 等。

#### (3) 译码显示电路。

可以使用型号库“数字信号显示”中的 16 进制 7 段数码管或者七段数码显示器来显示时、分、秒的数值。

(4) 校时电路。

手动校时，可以手动设置时、分、秒的值。

### 三、 实验内容

在设计复杂的数字系统之前，读者应该掌握使用 Dream Logic 绘制模块化电路的方法，这部分内容请参考本书数字电路实验 11 中的相关内容。此外本书数字电路实验 12 中介绍了同步时序电路、有限状态机的概念，还提供了计时器的设计方法。

请读者按照下面方法之一在本地创建一个项目，用于完成本次任务：

#### 方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建个人项目，然后使用 Dream Logic 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

#### 方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，就需要使用 Dream Logic 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的 URL

为 <https://www.codecode.net/engintime/Dream-Logic/Project-Template/blank.git>

#### 设计要求

根据数字钟原理框图 14-1，设计一个数字钟。要求如下：

- 1) 小时按 24 进制计数，分和秒按 60 进制计数。
- 2) 可以通过校时电路手动设置时、分、秒。

#### 改进一

改进数字钟电路，实现倒计时功能。要求通过按键设置数字钟计时模式：

- a) 当数字钟处于正常计时（加计数）模式时，按下此按键，数字钟转为倒计时（减计数）模式。
- b) 当数字钟处于倒计时（减计数）模式时，按下此按键，数字钟转为正常计时（加计数）模式。

#### 改进二

实现秒表功能。要求通过按键控制秒表复位（清零），并提供启动/暂停按键功能：

- a) 复位按键按下后，设置数字钟为 00:00:00（0 时 0 分 0 秒）。
- b) 当秒表正在计时，按下启动/暂停按键，秒表停止计时，再次按下此按键，秒表继续计时。

#### 改进三

实现定时闹钟功能。要求可以设置闹钟响铃时刻，当数字钟的计时达到所设定的时刻后，点亮指示灯。

## 第二部分 计算机组成原理实验

计算机组成原理实验是在八位模型机 DM1000 上开展的。可将组成原理实验划分为五个阶段：

**第一阶段**包含了第一个实验《实验环境的使用》。在该实验中，读者首先要掌握 Dream Logic 软件的基本使用方法，并从整体上了解一下 DM1000 八位模型机的特点，同时也需要学习 DM1000 汇编程序编译、加载到运行的整个过程。

**第二阶段**包含《寄存器实验》、《运算器实验》、《程序计数器实验》、《存储器实验》、《控制器实验》，每一个实验对应于 DM1000 中的一个子模块。完成这些实验后，读者就可以掌握 DM1000 每个子模块的功能和内部原理。并且通过仿真运行与各个子模块紧密相关的指令，读者将进一步理解各个子模块在整机中的分工与协作。

**第三阶段**包含《数据通路实验》，通过该实验，读者可从数字电路的角度观察数据传送的通道，学习数据通路是如何由控制器模块输出的各个控制信号建立起来的。通过搭建起的数据或地址信息通路，数据或地址信息在各个模块之间完成运算或转移。在实验的最后，希望读者能够理解微指令的本质就是数据通路，并能根据一条微指令的功能设计其数据通路，并得到微指令的机器码。

**第四阶段**包含《中断实验》，中断是计算机发展过程中的一个里程碑，其率先将并行工作的理念引入计算机，大大提高了 CPU 的效率。在本实验中，读者首先通过跟踪软中断指令的执行和中断返回过程，理解软中断的实现机制。然后通过 DM1000 中加入一个中断控制器子模块，从而实现 8 级中断请求以及中断嵌套，通过单步运行指令，读者可看到硬中断从触发到响应，再到中断服务程序的运行以及中断返回的整个过程，从而加深对中断机制以及中断处理概念的理解。

**第五阶段**包含《指令和微指令设计实验》、《阵列乘法器实验》，读者将有机会从整个系统的角度重新审视 DM1000 模型机。在这些实验中，读者需要以 DM1000 模型机已有的指令为基础，为扩展的新指令设计数据通路、构造微指令的机器码，并编写指令对应的微程序，从而实现新指令的功能。通过这些实验，读者将加深对微指令和指令的理解，明白不同的数据通路（微指令）通过组合来完成不同的功能（指令），设计新的指令就是将指令功能分解为微指令的过程。

# 实验 1 实验环境的使用

实验性质：验证

建议学时：2 学时

## 一、实验目的

- 掌握 Dream Logic 软件的基本使用方法。
- 了解 DM1000 模型机汇编程序的编写方法，并学习将汇编源程序编译为机器指令的方法。
- 了解 DM1000 模型机的微指令，并学习将微指令程序编译为微程序的方法。
- 掌握 DM1000 模型机加载指令程序和微指令程序的方法。
- 熟练掌握原理图中使用网络、端口、总线等在器件、模块间建立连接的方法，掌握模块化电路的设计方法。
- 熟练掌握 DM1000 模型机进行单周期仿真的方法，并了解各个工具窗口以及它们的作用。

## 二、预备知识

### 2.1 机器语言

机器语言是机器指令的集合。机器指令展开来讲就是一台机器可以正确执行的命令。电子计算机的机器指令是一列二进制数字。计算机将之转变为一列高低电平，以使计算机的电子器件受到驱动，进行运算。

每一种微处理器，由于硬件设计和内部结构的不同，就需要不同的电平脉冲来控制，使它工作。所以每一种微处理器都有自己的机器指令集，也就是机器语言。

早期的程序设计均使用机器语言。程序员将用 0、1 数字编成的程序代码打在纸带或卡片上，1 打孔，0 不打孔，再将程序通过纸带机或卡片输入计算机。

例如，如果使用 Intel 8086 CPU 完成运算  $s = 768 + 12288 - 1280$ ，机器码如下：

```
10110000000000000000000011
0000010100000000000110000
00101101000000000000000101
```

### 2.2 汇编语言

由于机器语言难于辨别和记忆，于是汇编语言产生了。汇编语言的主体是汇编指令。汇编指令和机器指令的差别在于指令的表示方法上。汇编指令是机器指令便于记忆的书写格式。

例如：Intel 8086 CPU 的机器指令 1000100111011000 表示把寄存器 BX 的内容送到 AX 中（寄存器，简单地讲是 CPU 中可以存储数据的器件，一个 CPU 中可能有多个寄存器。AX 是其中一个寄存器的代号，BX 是另一个寄存器的代号。）。汇编指令则写成 `mov ax, bx`。显然，这样的写法更便于阅读和记忆。

操作：寄存器 BX 的内容送到 AX 中

机器指令：1000100111011000

汇编指令：`mov ax, bx`

计算机能读懂的只是机器指令，那么如何让计算机执行用汇编指令编写的程序呢？这时，就需要一个能够将汇编指令转换成机器指令的翻译程序，这样的程序我们称其为汇编器或编译器。程序员用汇编语言写出源程序，再用汇编编译器将其译为机器码，由计算机最终执行。图 1-1 中描述了这个过程。

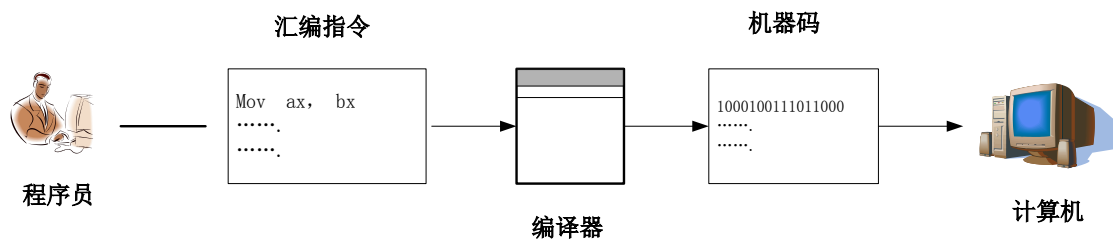


图 1-1：用汇编语言编写程序的工作过程

### 2.3 DM1000 八位模型机

Dream Logic 为读者提供了一个具有完整功能的八位模型机 DM1000。可将 DM1000 理解为一个具有 8 位地址线宽、8 位数据线宽的微处理器。DM1000 模型机原理图如图 1-2 所示：

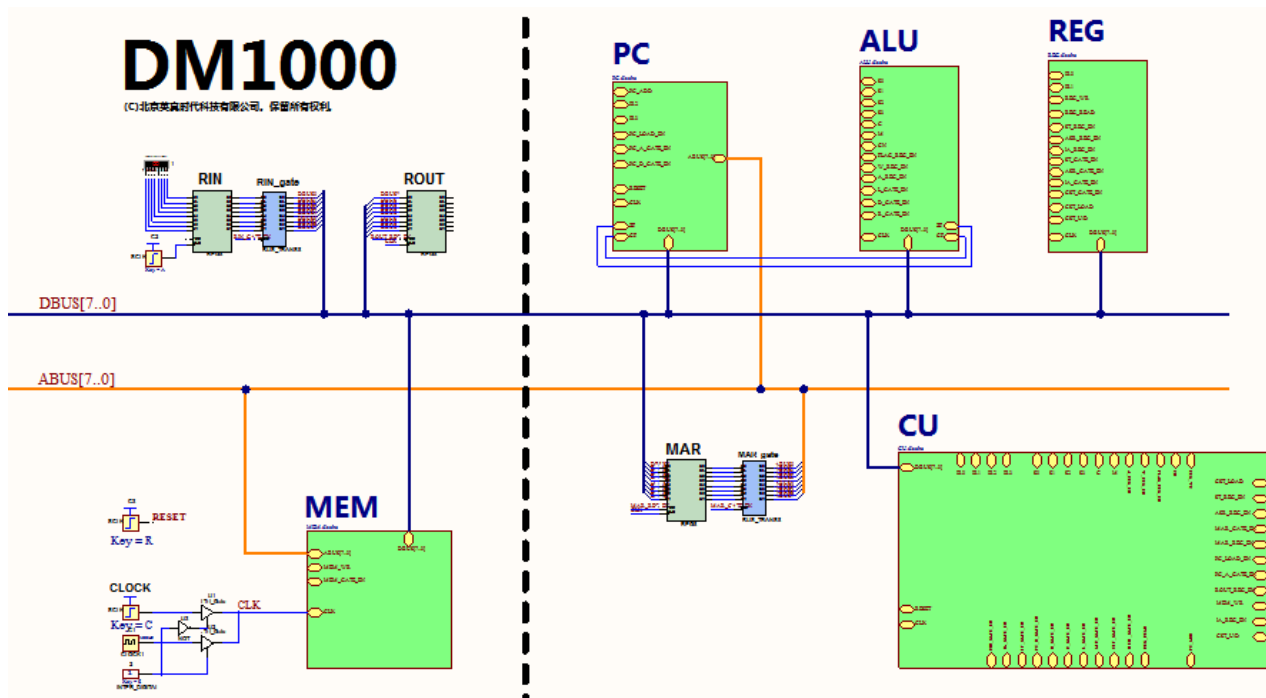


图 1-2：DM1000 八位模型机原理图

DM1000 主要由以下几部分组成：

- 数据总线 DBUS 和地址总线 ABUS。总线宽度均为 8 位。由于地址总线只有 8 位，所以最多只能访问 256 字节的物理地址。
- 控制信号。控制信号并没有设计成控制总线的方式，只是一组控制信号线的集合。
- 程序计数器 PC。
- 运算器 ALU。用于处理 8 位数据的算术运算和逻辑运算。
- 寄存器 REG。所有的寄存器都可以存储 8 位数据（一个字节）。
- 主存储器 MEM。其中提供了 256 个字节的存储容量。
- 控制器 CU。
- 输入寄存器 RIN
- 输出寄存器 ROUT
- 地址寄存器 MAR

### 2.4 DM1000 的指令集

DM1000 有一套完整的指令集，详情请查看附录 A 中的表 A-4。DM1000 的指令集与 Intel x86 处理器提



供的指令十分类似，所使用的汇编语言的语法也类似，如果读者之前系统的学习过 Intel x86 汇编语言的相关课程，就可以很轻松的掌握 DM1000 的指令集，并可以开始使用这些指令为 DM1000 编写汇编程序。

下面显示的是一段使用 DM1000 汇编指令编写的汇编源程序，程序分为代码段和数据段。代码段的开始位置用关键字 .text 标出，而数据段的开始位置用关键字 .data 标出。

```
.text          ; 代码段
mov r0, 16    ; 将立即数 16 赋值给通用寄存器 r0
mov a, num    ; 将标号 num 指定的主存储器单元内容复制到累加器 a 中
add a, r0     ; 将累加器 a 的值与通用寄存器 r0 的值相加，结果写回累加器 a 中

.data         ; 数据段
num: -1
```

## 2.5 DM1000 的微指令

DM1000 在执行一条指令时，会将其分为多个步骤即多条微指令，一个时钟周期执行一条微指令。

以指令“add a, r0”为例，可以分为以下步骤：

1. 取指。取指是所有指令执行的第一步，该步骤将程序计数器 PC 在存储器中指向的一个字节取出，并将其写入指令寄存器 IR 中。与此同时，CU 模块将指令机器码中的高 6 位扩展为 16 位，作为微程序计数器 uPC 的值，使 uPC 转去执行该指令对应的微程序。
2. 将寄存器 r0 的值送到工作寄存器 w 中。
3. ALU 将累加器 a 与工作寄存器 W 的值相加，并将结果写回累加器 a 中。
4. 指令执行完毕，程序计数器 PC 加 1，指向下一条指令。

其对应的微指令如下所示：

```
path [pc], ir    ; 取指。注意，所有指令共用此条取指微指令。
.....
path rx, w       ; 将通用寄存器 rx 的值传送到工作寄存器 W 中。
path alu_add, a  ; 将累加器 A 与寄存器 W 的值相加，结果写回 A 中。
inc pc          ; 程序计数器 PC+1，指向下一条指令。
reset upc       ; 将微程序计数器复位，指向共用的取指微指令，准备取出下一条指令。
```

## 三、实验内容

请读者按照下面的步骤完成实验内容，着重理解 DM1000 八位模型机执行指令的基本过程，并掌握 Dream Logic 软件和 CodeCode.net 平台的基本使用方法，为完成后续实验做好准备。

### 3.1 从 CodeCode.net 平台领取任务

CodeCode.net 平台是用于教师在线布置实验任务，并统一管理学生提交的作业的平台。如果没有使用到 CodeCode 平台，可以跳过此部分，直接从 3.2 节继续进行实验。

1. 读者通过浏览器访问<https://www.codecode.net>，可以打开 CodeCode.net 平台的登录页面。
2. 在登录页面中，读者输入帐号和密码，点击“登录”按钮，可以登录 CodeCode.net 平台。
3. 登录成功后，在“课程”列表页面中，可以找到计算机组成原理相关的实验课程。点击此课程的链接，可以进入该课程的详细信息页面。
4. 在课程的详细信息页面中，可以查看课程描述信息，该信息对于完成实验十分重要，建议读者认真阅读。点击左侧导航中的“任务”链接，可以打开任务列表页面。
5. 在任务列表中找到本次实验对应的任务，点击右侧的“领取任务”按钮，可以进入“领取任务”页面。
6. 在“领取任务”页面中，查看任务信息后，填写“新建项目名称”和“新建项目路径”，然后选择“项目所在的群组”，点击“领取任务”按钮后，可以创建个人项目用于完成本次实验，并自

动跳转到该项目所在的页面。

**提示：**在“领取任务”页面中，“新建项目名称”和“新建项目路径”这两项的内容可以使用默认值，如果选择的群组中已经包含了名称或者路径相同的项目，就会导致领取任务失败，此时需要修改新建项目名称和新建项目路径的内容。

7. 在个人项目页面中，包括左侧的导航栏、项目信息、文件列表、readme.md 文件内容等，如图 1-3 所示。
8. 在个人项目页面的图 1-3 中，点击红色方框中的按钮，可以复制当前项目的 URL，在本实验后面的练习中会使用这个 URL 将此项目克隆到读者计算机的本地磁盘中，从而供读者进行修改。

**提示：**领取任务的本质是：教师在新建任务时填写了实验模板的 URL，然后读者在领取任务时，根据任务中记录的 URL 来派生（Fork）出一个新的项目，供读者在其中进行修改。

**建议教师按照下面的步骤在 CodeCode.net 平台上布置此次实验任务：**

1. 在计算机组成原理实验课程中，新建一个实验任务。
2. 在“新建任务”页面中，教师需要添写“任务名称”，推荐使用“实验 1 实验环境的使用”；还需要填写“模板项目URL”，应该使用  
“<https://www.codecode.net/engintime/Dream-Logic/Project-Template/DM1000/DM1000.git>”。
3. 点击“新建任务”按钮，完成新建任务操作。

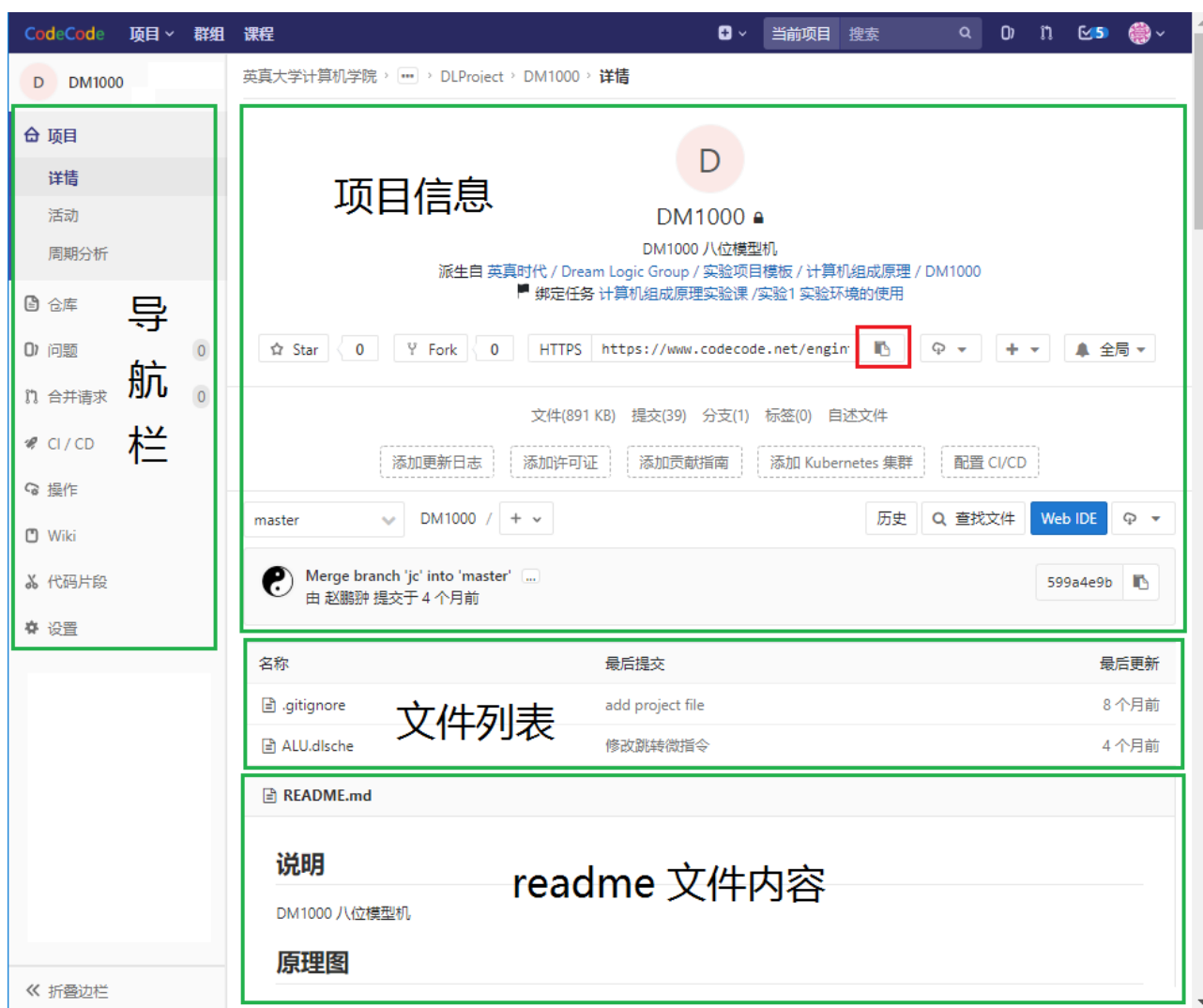


图 1-3：个人项目页面

### 3.2 启动 Dream Logic

在安装有 Dream Logic 的计算机上, 可以使用两种方法来启动 Dream Logic:

- 在桌面上双击“Engintime Dream Logic”图标。
- 或者
- 点击“开始”菜单, 在“所有程序”中的“Engintime Dream Logic”中选择“Engintime Dream Logic”即可启动程序。

启动 Dream Logic 后会首先打开“登录”窗口, 可以使用两种方法完成登录。

- 如果读者在 CodeCode.net 平台上注册了帐号, 在连接互联网的情况下, 可以使用 CodeCode.net 平台中已注册的用户名和密码进行登录。
- 如果读者还没有 CodeCode.net 平台上的帐号, 可以点击左下角“从加密锁获取授权”按钮, 获取授权之后, 完成登录。

### 3.3 将 CodeCode.net 平台上的 DM1000 八位模型机项目克隆到本地

如果读者从 CodeCode.net 平台领取了任务, 可以在领取任务后将读者的个人项目克隆到本地磁盘中; 如果读者没有 CodeCode.net 平台账号无法领取任务, 可以直接将实验模板项目克隆到本地磁盘中。

#### 将领取任务新建的个人项目克隆到本地

1. 选择 Dream Logic 菜单“文件->新建->从 Git 远程库新建项目”, 打开“从 Git 远程库新建项目”对话框。
2. 在“Git 远程库 URL (G)”中填入读者个人项目的 Git 远程库的 URL 地址 (在图 1-3 中, 点击红色方框中的按钮, 可以复制当前项目的 URL)。
3. “项目文件夹名称”填入“DM1000”。
4. “项目位置”选择新建项目需要保存到的磁盘目录。
5. 点击“确定”按钮后会弹出一个 Windows 控制台窗口, 并开始运行 Git 克隆命令将 Git 远程库克隆到本地, 一定要等克隆成功后再关闭该窗口。
6. 克隆项目成功后, 在对话框中选择“克隆成功, 打开新建项目”就可以打开新建的项目。

#### 直接将实验模板项目克隆到本地

由于 CodeCode.net 平台上提供的实验模板是开放的, 所有的人都可以访问。所以, Dream Logic 可以直接将 DM1000 八位模型机实验模板克隆到本地磁盘。步骤如下:

1. 选择 Dream Logic 菜单“文件->新建->从 Git 远程库新建项目”, 打开“从 Git 远程库新建项目”对话框。
2. 在“Git 远程库 URL (G)”中填入 DM1000 项目模板 Git 远程库的 URL 地址: <https://www.codecode.net/engintime/Dream-Logic/Project-Template/DM1000/DM1000.git>
3. “项目文件夹名称”填入“DM1000”。
4. “项目位置”选择新建项目需要保存到的磁盘目录。
5. 点击“确定”按钮后会弹出一个 Windows 控制台窗口, 并开始运行 Git 克隆命令将 Git 远程库克隆到本地, 一定要等克隆成功后再关闭该窗口。
6. 克隆项目成功后, 在对话框中选择“克隆成功, 打开新建项目”就可以打开新建的项目。

#### 注意:

1. 为了能正常使用从 Git 远程库新建项目功能, 用户需要在本地安装 Git 客户端程序, 并且在安装 Git 的过程中会有一个步骤询问是否设置 Windows 环境变量, 此时一定要设置上 Windows 环境变量。
2. 文件路径中不允许包含中文, 请读者在使用之初就养成使用英文命名项目或文件的习惯。

### 3.4 DM1000 八位模型机项目

打开新建的 DM1000 模型机项目后, 在左侧的“项目管理器”窗口中会显示 DM1000 项目中包含的所有文件。请读者根据表 1-1 了解每个文件的基本情况。

文件夹或文件名称	说明
子模块	文件夹
ALU.dlsche	算术逻辑运算模块原理图文件
PC.dlsche	程序计数器模块原理图文件
REG.dlsche	寄存器模块原理图文件
CU.dlsche	控制模块原理图文件
MEM.dlsche	主存储器模块原理图文件
uPC_NEXT.dlsche	微程序计数器模块原理图文件
汇编源程序	文件夹
ram.asm	此文件中默认包含了一个简单的汇编语言源程序,作为示例程序提供给读者,读者可以修改此文件中的源代码,根据需要编写自己的汇编程序。
ram.bat	这是一个 Windows 批处理文件,运行此文件可以执行其中的命令让 dmasm.exe 编译器将 ram.asm 中的汇编代码编译为机器码,并写入存储器映射文件 ram.rxm 中,同时还会生成列表文件 ram.lst 和调试信息文件 ram.dbg。
微指令源程序	文件夹
rom.masm	此文件中包含了 DM1000 的全部微指令源程序。读者可以根据需要修改此文件,添加新的微指令。
rom.bat	这是一个 Windows 批处理文件,运行此文件可以执行其中的命令让 microasm.exe 编译器将 rom.masm 中的微指令源程序编译为机器码,并写入存储器映射文件 rom.rxm 中,同时还会生成列表文件 rom.lst 和调试信息文件 rom.dbg。
存储器映射文件	文件夹
ram.rxm	主存储器映射文件,其中包含了汇编程序生成的机器码。当 DM1000 开始运行时,此文件中的内容会被加载到 MEM 子模块的 RAM 中,从而允许 DM1000 执行其中的机器码。
rom.rxm	微指令存储器映射文件,其中包含了微指令源程序生成的机器码。当 DM1000 开始运行时,此文件中的内容会被加载到 CU 子模块的 ROM 中,从而允许 DM1000 将指令的机器码翻译为微指令的机器码并执行。
DM1000.dlsche	DM1000 八位模型机的总图,由各个子模块、时钟和复位控制通过总线或网络连接,组成可运行程序的计算机系统。

表 1-1: DM1000 项目中的文件

除了以上可以从项目管理器中看到的文件之外,在项目文件夹中还有一些其它文件需要进行说明。读者可以在项目管理器窗口中右键点击项目节点(根节点),选择右键菜单中的“打开所在的文件夹”,Dream Logic 会使用 Windows 资源管理器打开项目所在的文件夹,需要说明的文件参见表 1-2。

文件名称	说明
DM1000.dlproj	项目文件
dmasm.exe	DM1000 汇编程序的编译器。
dmasm.c	dmasm.exe 程序的 C 源代码。
microasm.exe	DM1000 微指令程序的编译器。
microasm.c	microasm.exe 程序的 C 源代码。
ram.dbg	DM1000 汇编程序的调试信息文件。在使用 dmasm.exe 编译 ram.asm 文件时,会生成此文件。

ram.lst	DM1000 汇编程序的列表文件。在使用 dmasm.exe 编译 ram.asm 文件时, 会生成此文件。
rom.dbg	DM1000 微指令程序的调试信息文件。在使用 microasm.exe 编译 rom.masm 文件时, 会生成此文件。
rom.lst	DM1000 微指令程序的列表文件。在使用 microasm.exe 编译 rom.masm 文件时, 会生成此文件。

表 1-2: DM1000 项目文件夹中的其它文件

这里还需要对 DM1000 的原理图 DM1000.dlsche 进行一些说明:

1. 此原理图是 DM1000 的顶层原理图, 其通过总线将各个模块连接起来, 这样大大简化了顶层原理图的复杂度, 可以使读者很清晰的掌握各个模块间的连接关系, 快速掌握 DM1000 的整体结构。
2. 在此原理图中遵守左面输入, 右面输出的约定。模块左侧的总线表示输入; 模块右侧的总线表示输出; 模块顶部或底部的总线, 表示模块既能从总线接收数据, 又能向总线发送数据。
3. 在此原理图中提供了时钟和复位两个控制信号。其中, 复位信号可以使用键盘上的 R 键控制, 每次复位都能使 DM1000 模型机从第一条指令开始执行程序。时钟提供了单步时钟和自动时钟, 可以使用键盘上的 S 键进行切换。单步时钟可以使用键盘上的 C 键控制, 默认是高电平, 也就是说每次按下 C 键, 都会让单步时钟变为低电平, 并产生一个下降沿, 抬起 C 键会让单步时钟恢复为高电平, 并产生一个上升沿。
4. 除了使用总线连接各个模块的接口外, 各个模块的接口还通过使用相同的名称表示它们具有连接关系。为了确保原理图简洁清晰, PC/ALU/REG 等模块的输入信号都是通过这种方式与 CU 模块输出的控制信号进行连接的。
5. 总图中的模块是通过子模块文件的名称与之对应的, 例如读者使用鼠标左键点击选中 REG 模块后, 在右侧的“属性”窗口中就会显示此模块的所有属性, 其中“模块原理图文件”的属性值为“REG.dlsche”, 说明此模块是与原理图文件 REG.dlsche 对应的。

**提示:** 用鼠标左键双击总图中的模块, 就会自动跳转到对应的子模块原理图文件, 在子模块原理图文件中的空白处双击鼠标左键, 就会返回到总图中, 这样操作起来十分方便。

**提示:** 如果读者是第一次使用 Dream Logic, 请务必在本书第一部分的实验一中找到原理图常用操作的介绍内容, 并自行练习, 为后面的实验内容做好准备。

### 3.5 练习使用 DM1000 运行一个简单的汇编程序

接下来, 本实验会带领读者使用 DM1000 运行一个简单的汇编程序, 为后续的实验打下一个坚实的基础。

首先请读者学习一下如何为 DM1000 编写汇编程序:

1. 打开项目下的汇编源程序文件 ram.asm, 结合附录 A 中表 A-4 的内容, 熟悉每一条指令的功能。
2. 读者在后面的实验过程中, 可以在 ram.asm 文件中根据需要编写新的汇编程序。
3. 汇编程序编写完毕后, 可以在“项目管理器”中的 ram.bat 文件上点击右键, 选择菜单中的“运行批处理文件”即可完成对 ram.asm 文件的编译。如果在弹出的 Windows 控制台窗口中, 没有报告任何语法错误信息, 就说明编译成功了, 并且会在最后显示成功生成的文件名称, 包括存储器映射文件 ram.rxm, 列表文件 ram.lst 和调试信息文件 ram.dbg。

读者可以按照下面的步骤练习如何修改汇编程序中的语法错误:

1. 首先将 ram.asm 中汇编指令 mov r0, 16 中的寄存器 r0 删除后保存文件。
2. 接着按照之前介绍的方法对汇编程序文件进行编译, 在 Windows 控制台窗口中会提示汇编文件中的 mov 指令缺少操作数, 并提示出存在语法错误的文件名称和行号。
3. 根据语法错误提示信息, 修改汇编程序后保存。

#### 4. 重新编译汇编程序文件，确保可以成功。

DM1000 提供的微指令程序 rom.masm 文件的编译方法与汇编程序文件的类似，只是换为用 rom.bat 批处理文件进行编译，请读者自行练习，并注意查看微指令程序在编译过程中生成了哪些文件。接下来，读者可以按照下面的步骤了解一下汇编程序生成的文件，以及微指令程序生成的文件是如何在 DM1000 中使用的：

1. 在 DM1000.dlsche 原理图中，在主存储器单元 MEM 模块上双击可以打开此模块对应的原理图 MEM.dlsche 文件。
2. MEM.dlsche 文件中的 RAM 是主存储器，用于存储 DM1000 在运行过程中所需的指令和数据。用鼠标左键选中 RAM，可以在右侧的属性窗口中查看此器件的所有属性。
3. 在 RAM 属性中的“调试”节点下，读者可以看到“存储器映射文件”属性已经默认设置为 ram.rxm。这样，当 DM1000 启动仿真时，就会将 ram.rxm 文件中的内容加载到 RAM 中，作为指令和数据。读者可以选中此属性后，点击属性值右侧的按钮，在弹出的对话框中为 RAM 指定其他的存储器映射文件。当然，在一般情况下，读者不需要修改此属性值。
4. 在 RAM 属性中的“调试”节点下，读者可以看到“调试信息文件”属性和“程序计数器名称”属性已经默认设置为 ram.dbg 和 PC。当 DM1000 启动仿真时，会使用 ram.dbg 文件提供的调试信息和 PC 寄存器的值，在“源代码”窗口中为读者实时显示 DM1000 将要执行的指令。
5. 在 MEM.dlsche 原理图文件的空白处双击鼠标左键，返回 DM1000 的总图中。
6. 在 DM1000.dlsche 原理图中，双击控制单元 CU 模块打开 CU.dlsche 原理图。其中的 ROM 是微程序存储器，读者可以选中 ROM 查看其属性，并与之前观察的 RAM 中的属性进行对照。
7. 在 CU.dlsche 原理图文件的空白处双击鼠标左键，返回 DM1000 的总图中。

现在，读者应该已经对 DM1000 有了一些初步的了解，但是这些了解都是静态的。接下来，读者可以按照下面的步骤对 DM1000 进行一次完整仿真，通过让 DM1000 运行一个简单的汇编程序，从而对 DM1000 有一个动态的认识：

1. 首先，确保当前打开的是 DM1000.dlsche 原理图，然后选择菜单“仿真”中的“启动仿真”（快捷键 F5）。

**注意：**“启动仿真”功能总是对当前打开的原理图进行仿真，因此，启动仿真前一定要先打开需要仿真的原理图。

2. 在 DM1000 刚刚启动仿真时，按下 R 键复位，将 PC 和 uPC 寄存器的值初始化为 0，这就会让 DM1000 从汇编程序的第一条指令开始执行，同时也是从微指令的第一条指令开始执行。
3. 选择“视图”菜单中的“寄存器”弹出“寄存器”窗口，在寄存器窗口中找到 PC 和 uPC 的值，确认其为 0，其中 PC 是 8 位计数器，uPC 是 16 位寄存器。在寄存器窗口中灰色的项目表示对应的寄存器没有使能，时钟上升沿到来时不会写入数据，而黑色的表示寄存器使能，时钟上升沿到来时，寄存器的输入传送到输出端（写入）。红色表示上个时钟上升沿到来时发生写入操作的寄存器。
4. 在寄存器窗口中双击 PC 计数器对应的项目，可以在原理图中迅速定位 PC 寄存器的位置，请读者确认 PC 计数器的 EN 管脚为低电平，表示 PC 计数器不使能，禁止计数；同时，PC 计数器的 8 位输出数据线都为低电平，表示 PC 计数器的值为 0。用相同的方法定位 uPC 寄存器，并观察其各个管脚的电平。
5. 选择“视图”菜单中的“存储器”弹出“存储器”窗口，选择此窗口工具栏下拉列表中的 RAM，可以查看 RAM 存储器中的数据。RAM 存储器的地址线和数据线都是 8 位的，窗口中的每一行表示一个字节，其中冒号左边的是 8 位地址值，冒号右边的是 8 位数据值。RAM 存储器加载的数据应



该是汇编程序生成的机器码，所以其中的数据应该与 ram.rxm 存储器映射文件中的数据相同，读者可以选择“视图”菜单中的“项目管理器”弹出“项目管理器”窗口，然后双击 ram.rxm 文件，使用二进制编辑器打开此文件，确认 ram.rxm 文件中的数据与 RAM 中的数据相同。

- 在“存储器”窗口中具有灰色底色的行，表示存储器当前输入地址线以及输出数据线上的值。读者可以选择“存储器”窗口工具栏上的定位按钮，或者在窗口中单击右键，在弹出的右键菜单中点击“定位存储器”在原理图中迅速定位 RAM 存储器，并确认其地址线上的值为 0（此时，RAM 地址线上的值是由 PC 寄存器提供的），输出数据线上的值是 0x8C。由于此时正在从 RAM 中读取数据，所以其 WR 管脚为高电平。
- 请读者按照前面步骤 5 和步骤 6 的方法，在“存储器”窗口中查看 ROM 存储器的值，并与微指令源程序生成的 rom.rxm 文件中的内容进行比较，确认其一致。需要注意的是，ROM 是微程序存储器，它存储的是地址和内容均固定的微指令，只能读出，不能写入。ROM 地址线上的值是由微程序寄存器 uPC 提供的，ROM 输出的就是一条 32 位的微指令，微指令中的每一位都有特定的含义，主要作为控制信号。例如 ROM 的 Q25 管脚输出的信号 MEM\_WR 接到 RAM 的 WR 输入端，用于控制 RAM 的读写操作。由于 ROM 的地址线是 12 根，数据线是 32 根，所以在“存储器”窗口中，ROM 的一行数据包括 4 个字节。

在仿真过程中通过“寄存器”窗口、“存储器”窗口和原理图窗口可以观察到 DM1000 在运行过程中的各种状态，为了方便读者更好的理解 DM1000 的运行机制，Dream Logic 还提供了“源代码”窗口帮助读者从汇编指令和微指令的角度观察 DM1000 的运行过程，从而将汇编指令与处理器的行为更加紧密的联系起来，使读者可以在仿真的过程中同时从汇编指令和微指令的角度、从系统的角度（寄存器、存储器）、从数字电路的角度（时序逻辑、组合逻辑）加深对计算机组成原理的理解。

请读者按照下面的步骤学习“源代码”窗口的用法：

- 在 DM1000 启动仿真后，默认会在左侧显示两个源代码窗口“源代码 1”和“源代码 2”。其中源代码 1 窗口默认显示 RAM 存储器对应的汇编代码，也就是汇编程序 ram.asm 编译生成的列表文件 ram.lst 中的内容；同样的，源代码 2 窗口默认显示 ROM 存储器对应的微指令源代码，也就是微指令源程序 rom.masm 编译生成的列表文件 rom.lst 中的内容。

- 先来看源代码 1 窗口中的内容，其一行源代码的内容可以分为四个部分：

```
0005    00    8C 10    mov    r0, 16
```

- 最左侧的 0005，表示此条指令在源代码文件 ram.asm 中的第 5 行。
- 第二部分 00，表示此条指令的机器码在存储器映射文件中的偏移地址为 0。
- 第三部分 8C 10，表示此条指令的机器码，包含了两个字节（十六进制表示）。DM1000 8 位模型机指令的机器码可以是单字节或双字节。
- 第四部分 mov r0, 16 是此条指令的汇编代码。此条指令会将立即数 16 存入 r0 寄存器中。

源代码 1 窗口左侧的蓝色箭头表示 PC 计数器在存储器中指向的位置，由于 DM1000 启动并复位后 PC 计数器的值为 0，所以蓝色箭头指向偏移为 0 的代码行。请读者将源代码 1 窗口中指令的机器码与存储器窗口中 RAM 的内存进行比较，确认其内容一致。

- 源代码 2 窗口中的格式：

```
0005    00    EF 3F F9 FF    path    [pc], ir
```

- 最左侧的 0005，表示此条微指令在微程序源文件 rom.asm 中的第 5 行。
- 第二部分 00，表示此条微指令在 ROM 存储器中的偏移地址为 0。由于 DM1000 启动并复位后 uPC 寄存器的值为 0，所以蓝色箭头指向偏移为 0 的微指令。
- 第三部分 EF 3F F9 FF 是此条微指令的机器码（十六进制表示），共 4 个字节。DM1000 8 位模型机微指令的机器码都是 4 字节。
- 第四部分 path [pc], ir 是此条微指令的代码。

这是一条取指微指令，它的功能是将当前 PC 指向的指令从主存储器 RAM 中取出放到数据总

线 DBUS 上, 在下个时钟上升沿到来时写入指令寄存器 `ir` 中暂存。任何一条指令执行的第一条微指令都是取指微指令, 取指微指令的工作方式如下:

- PC 计数器向地址总线的输出三态门 `PC_gate` 使能, 从而将 PC 计数器的值传送到地址总线 ABUS 上。
- ABUS 上的值作为 RAM 存储器的地址, RAM 存储器的读写控制端输入高电平, 于是将 RAM 当前地址指向存储单元的内容 (一个字节指令机器码) 读出。
- RAM 存储器输出三态门 `MEM_gate` 使能, 从而将读出的指令字节码输出到数据总线 DBUS 上。

微指令实质上是 32 位控制信号的编码, 将取指微指令 “`path [pc], ir`” 对应的十六进制编码 “`EF 3F F9 FF`” 转换为 32 位二进制编码得 “`1110 1111 0111 1111 1111 1001 1111 1111`”。这 32 位编码依次对应于微程序存储器 ROM 输出的 32 位控制信号, 32 位输出信号中绝大多数是低电平有效, 而一条微指令中往往只有少数信号是有效的, 因此在分析 ROM 输出的信号时, 通常只需考虑某几位低电平输出信号。

DM1000 的控制单元 CU 接收并识别数据总线 DBUS 上的指令机器码。经过解析, 就得到了该指令对应的微程序入口地址。该入口地址在下个时钟上升沿到来时会写入微程序计数寄存器 `uPC`, 于是转去执行微程序。上升沿到来的同时, 也将数据总线 DBUS 上的指令字节码写入了指令寄存器 `ir` 中。该过程需要读者通过仿真, 结合电路去加以理解, 以此为基础, 后续的其他微指令执行过程就容易理解了。

4. 在发送时钟上升沿执行指令对应微程序的入口微指令之前, 请读者再次从工具窗口和原理图窗口中确认以下内容:
  - PC 寄存器的值 0 已经传送到地址总线 ABUS 上;
  - RAM 存储器 0 地址处的值 `0x8C` 已经通过了总线收发器 `MEM_gate` (可以在 “总线收发器” 窗口中查看总线收发器的值和使能状态, 双击可迅速在原理图中定位总线收发器的位置) 传送到数据总线 DBUS 上;
  - 数据总线上的数据已经到达了 IR 寄存器输入端, 且 IR 寄存器使能端 `EN` 为低电平, 写使能, 当下个时钟上升沿到来时, IR 寄存器输入端的指令将被写入 IR 寄存器中。
  - 指令字节 `0x8C` 的高 6 位扩展为 `0x460` (该指令对应微程序的入口地址, 指向入口第一条微指令) 并传送到 `uPC` 寄存器的输入端, `uPC` 寄存器使能 (`EN` 低电平表示使能)。当下一个时钟上升沿到来时, `0x460` 将写入微程序寄存器 `uPC` 中, 这样微程序就转移到 `0x460` 处执行。
5. 现在, 读者可以使用单步时钟发送一个上升沿来触发 DM1000 执行下一条微指令了! 方法是: 在原理图窗口中按下单步时钟对应的键盘上的 C 键再抬起, 该步操作产生的时钟上升沿使得微程序寄存器 `uPC` 将输入端的地址写入, 微指令存储器 ROM 于是输出新的微指令, 控制处理器电路执行相应的操作。
6. 观察 DM1000 在第一个时钟上升沿触发后进入的状态: IR 寄存器的值已经变为 `0x8C` (在寄存器窗口中由于 IR 寄存器刚刚发生写操作, 所以会用红色高亮显示); `uPC` 寄存器的值变为 `0x0460`, 在源代码 2 窗口中蓝色箭头跳转指向偏移为 `0x460` 处的微指令, 从源代码 2 窗口中可以看到, 蓝色箭头指向的微指令正是 `mov rx, immediate` 的第一条微指令 `inc pc`, 它的功能是将 PC 加 1, 为读出第二个指令字节码 (立即数) 准备地址。后续每输入一个时钟上升沿, 执行一条微指令, 也就是单步运行的实质。

接下来读者就可以通过发送单步时钟的方式依次执行指令 `mov r0, 16` 对应的 4 条微指令了, 由于在后面的实验中会对 DM1000 的各个模块有更加深入的分析, 所以这里就只是对这 4 条微指令做一个简单的说明:

```
0283    460    FF FF FF FF    inc    pc
```



0284	464	EF 7F B9 FF	path	[pc], rx
0285	468	FF FF FF FF	inc	pc
0286	46C	CF FF FF FF	reset	upc

- 第一条微指令是将 PC 寄存器的值加 1，从而让 PC 指向指令在 RAM 存储器中的第二个机器码，也就是立即数 16；
- 第二条微指令是将 PC 指向的机器码 16 放入 r0 寄存器中，到这里此条指令的功能就完成了，接下来要执行下一条指令。
- 第三条微指令将 PC 寄存器的值加 1，则 PC 就指向了下一条指令的第一个机器码；
- 第四条微指令将 uPC 的值重置为 0，则 uPC 又指向了 ROM 的 0 地址处，即第一条取指微指令。

读者可以按照之前的步骤依次执行此汇编程序中的所有指令，直到在累加寄存器 A 中得到 16 和-1 的和。在仿真的过程中注意观察各个工具窗口和原理图窗口的变化，熟练掌握使用单步时钟仿真 DM1000 的过程，为后面的实验打下坚实的基础。

#### 注意：

1. 在仿真过程中，将鼠标放置到源代码窗口中的寄存器上，会显示出当前寄存器的值。
2. 在仿真过程中，无论是在总图中，还是在子模块中，都可以使用键盘上的 C 按键发送单步时钟。
3. 在仿真过程中，无论是双击原理图中的子模块，还是双击子模块原理图的空白处，都是在同一个窗口中变换仿真层级，不会打开模块对应的原理图文件。

### 3.6 简单修改原理图

目前，时钟信号 CLK 和复位信号 RESET 主要是通过网络标签与各个模块中的对应接口进行连接的，请读者使用“绘制”菜单中的“网络”功能，使用网络将 CLK 信号与各个模块中的对应接口连接起来，将 RESET 信号也与各个模块中的对应接口连接起来。

通常，网络标签用于将原理图中距离比较远的、比较分散的多个网络连接在一起，比使用网络画线连接要方便很多。但是，使用网络画线连接比较直观，容易辨认连接关系。所以，在绘制原理图时，应该综合使用网络标签和网络，充分发挥各自的优点，才能绘制出可读性好、便于修改的原理图。

### 3.7 提交作业

如果读者是通过从 CodeCode.net 平台领取任务创建的个人项目，并将个人项目克隆到本地进行实验，实验结束后可以将本地已更改的项目再推送到 CodeCode.net 平台的个人项目中，方便教师通过 CodeCode.net 平台查看读者提交的作业。步骤如下：

1. 在项目管理器窗口中，右键点击项目节点，在弹出的菜单中，选择“Git”中的“推送当前分支到 Git 远程库”菜单项，弹出“推送当前分支到 Git 远程库”的对话框。
2. 在“推送当前分支到 Git 远程库”对话框中，输入本次项目更改的提示信息。
3. 点击“推送”按钮，可以将当前项目推送到 CodeCode.net 平台的个人项目中。

可以按照下面的步骤查看推送后的项目：

1. 使用浏览器访问<https://www.codecode.net>，使用已注册的帐号登录 CodeCode.net 平台。
2. 在“课程”列表中选择计算机组成原理实验课程，打开计算机组成原理实验课程的详细信息页面，点击左侧导航栏的“任务”链接，打开任务列表页面。
3. 在任务列表页面打开本次实验对应的任务，在任务详情中点击“查看项目详情”按钮，可以跳转到读者的个人项目页面。
4. 在个人项目页面中，可以查看当前项目的全部内容。
5. 在项目左侧的导航栏中点击“仓库”中的“提交”链接，可以打开提交列表页面。
6. 在提交列表页面中，点击最后一次提交，可以查看此次提交发生变更的文件。

**技巧：**在 Dream Logic 的项目管理器窗口中，选中项目节点，点击鼠标右键，在弹出的右键菜单中，选择“Git”中的“使用浏览器访问 Git Origin”菜单项，可以自动打开本地项目在 CodeCode.net 平台上对应的个人项目。

## 四、 思考与练习

1. 本次实验，通过使用 DM1000 运行了一段完整的程序。在程序的执行过程中，哪些微指令的执行使程序计数器 PC 发生了改变？微程序计数器 uPC 又有何变化规律。

**提示：**一条汇编指令对应一段微指令程序，一个时钟周期，执行一条微指令，因此，一条汇编指令的执行需要多个时钟周期。在源代码窗口中，蓝色箭头指向的第二列十六进制数就是 PC 或 uPC 的值。也可以在寄存器窗口中查看 PC 和 uPC 的值。当 uPC 等于 0 时，此时指向取指微指令，执行完取指操作后，uPC 依次加 4，逐条执行汇编指令对应微程序中的微指令。执行汇编指令对应的最后一条微指令后，uPC 复位，又指向取指微指令，取出下一条汇编指令并执行。

2. 在运行程序指令时，每条指令执行的第一步是取出该条指令。取指微指令为“path [pc], ir”，其对应的 16 进制机器码为 0xFF93FEF (小端模式)，将其与 CU 模块中 ROM 输出的控制信号进行比较确认是否一致。

**提示：**可以将取指时 CU 中 ROM 输出的 32 位控制信号按从高位到低位（Q31 表示最高位，Q0 表示最低位）的顺序编码，高电平表示为 1，低电平表示为 0。然后将其转换为 16 进制数，再与取指微指令的机器码比较。

3. DM1000 采用何种编码方式表示带符号的整数，是原码、反码还是补码？-1 的编码是多少？
4. 请读者按照本书第一部分数字电路实验 1 第二节中的内容，练习绘制简单的原理图，学习 DreamLogic 的基本使用方法。
5. 请读者参考本书第一部分数字电路实验 11 中的内容学习 Dream Logic 中模块和子模块的使用方法。

## 实验 2 寄存器实验

实验性质：验证

建议学时：2 学时

### 一、实验目的

- 掌握寄存器模块 REG 内部电路原理。
- 掌握寄存器三态输出门的工作原理。
- 掌握寄存器模块 REG 在微处理器中承担的功能。

### 二、预备知识

DM1000 中含有 13 个寄存器，寄存器的名称和功能如表 2-1 所示：

分类	名称	功能
输入寄存器	RIN	通过 IN 指令输入 8 位数据到累加器 A，模拟 DM1000 的输入设备
输出寄存器	ROUT	通过 OUT 指令将累加器 A 的数据输出，模拟 DM1000 的输出设备
地址寄存器	MAR	连接数据总线 and 地址总线，是数据总线 DBUS 上的数据作为地址传送到地址总线 ABUS 上的唯一通道
指令寄存器	IR	存放指令的第一个字节码，该字节码含有指令类型 and 寄存器操作数等信息
累加寄存器	A	存放算术逻辑运算的第一个操作数
工作寄存器	W	存放算术逻辑运算的第二个操作数
标志寄存器	FLAG	存放算术逻辑运算结果的零标志 and 进位标志，用于条件转移指令
通用寄存器	R0~R3	暂存程序运行过程中的数据或地址
栈顶指针寄存器	SP	存放栈顶地址，入栈时，SP-1，栈向低地址空间生长；出栈时，SP+1，栈向高地址空间生长
辅助寄存器	ASR	暂存指令执行过程中的临时数据或地址

表 2-1：DM1000 内部寄存器

DM1000 中的寄存器都是 8 位的，只可以存放一个字节，很明显是一个受限的系统。因此，在编写程序的时候，要避免立即数越界的问题。

例如：

```
mov a, 256
```

由于 256 的二进制码为 1 0000 0000，超过了 8 位，超出的位在编译时将自动舍弃。这条指令放入 a 寄存器中的值是 0，而不是 256。

#### 2.1 写寄存器

寄存器主要用于存储程序运行过程中的数据或地址信息。DM1000 中的常用寄存器如图 2-1 所示，当管脚 EN 为低电平时，寄存器写使能，即允许将管脚 D0~D7 输入数据传送到输出端 Q0~Q7。所以，当 EN=0 时，若管脚 CLK 输入时钟上升沿，D0~D7 的值就保存到寄存器中，并从 Q0~Q7 输出。

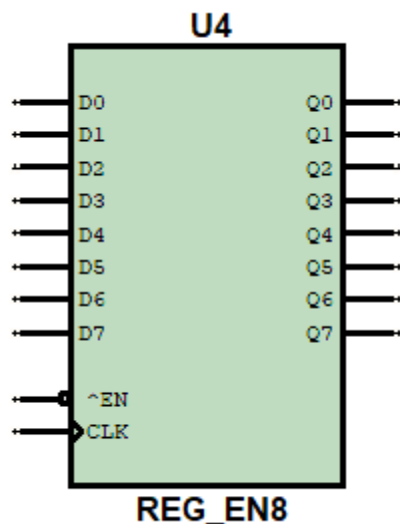


图 2-1: 低电平写使能寄存器

## 2.2 DM1000 中的单向总线收发器

当多个部件与总线相连时，如果出现两个或两个以上部件同时向总线发送信息，会导致信号冲突。

在 DM1000 中，为有效防止总线冲突，挂载到总线上的输入部件（向总线发送信息的部件）必须通过 8 位单向总线收发器（8 位单向三态门）与总线相连。

挂载在同一总线上的多个单向总线收发器，当且仅当其中一个使能时，使能的收发器才能向总线发送信息，其余的将被禁止，这样保证只有一个部件向总线发送信息，避免了不同信号之间的冲突。如图 2-2 所示。

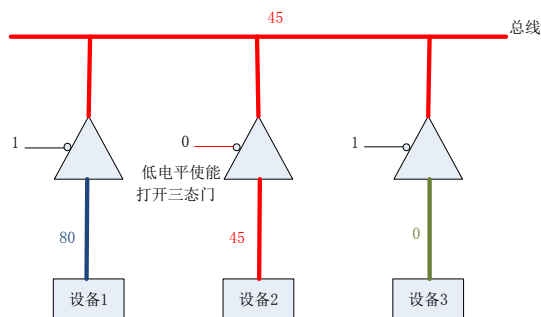


图 2-2: 多个设备通过单向总线收发器向总线传送数据示意图

## 三、实验内容

### 3.1 任务（一）掌握寄存器的写操作

本次实验涉及到“单周期时钟”和“一位交互式数字信号”两种器件的使用，读者可以参考第一部分中的数字电路实验 1，详细了解它们的使用方法。

请读者按照下面方法之一在本地创建一个项目，用于完成本次任务：

#### 方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建个人项目，然后使用 Dream Logic 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

#### 方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，就需要使用 Dream Logic 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的 URL

为 <https://www.codecode.net/engintime/Dream-Logic/Project-Template/DM1000/DM1000.git>。

建议教师按照下面的步骤在 CodeCode.net 平台上布置此次实验任务：

- (1) 在计算机组成原理实验课程中，新建一个实验任务。
- (2) 在“新建任务”页面中，教师需要添写“任务名称”，推荐使用“实验 2 寄存器实验-任务（一）（二）”；还需要填写“模板项目URL”，应该使用  
“<https://www.codecode.net/engintime/Dream-Logic/Project-Template/DM1000/DM1000.git>”。
- (3) 点击“新建任务”按钮，完成新建任务操作。

请读者按照下面的步骤完成本次实验：

1. 在原理图 DM1000.dlsche 中，找到输入寄存器 RIN。
2. 在打开的 DM1000.dlsche 原理图中，选择菜单“仿真”下的“启动仿真”（快捷键 F5）。
3. 启动仿真后，在底部的寄存器窗口中，找到 RIN 寄存器，如图 2-3 所示。打开寄存器窗口的方法是，选择菜单“视图”中的“寄存器”即可。
4. 在寄存器窗口中可以看到，RIN 使能，说明 RIN 允许写入数据；但是它的值为空，说明此时 RIN 的输出端数据无效，即至少存在一位输出是无效的（蓝色网络表示无效电位）。



名称	值	说明	使能	全名
R2		R2通用寄存器	否	REG.R2
R3		R3通用寄存器	否	REG.R3
RIN		输入寄存器	是	RIN
ROUT		输出寄存器	否	ROUT
ST		栈寄存器	否	REG.ST
uPC	0x0000	微程序计数器	是	CU.uPC
W		工作寄存器	否	ALU.W

图 2-3：寄存器窗口中 RIN 的当前状态

5. 在寄存器窗口中选 RIN，双击鼠标左键，可快速定位到寄存器在原理图中的位置。可以查看 RIN 此时的状态，如图 2-4 所示。RIN 的输入均为低电平，显示为灰色，输出无效，显示为蓝色。

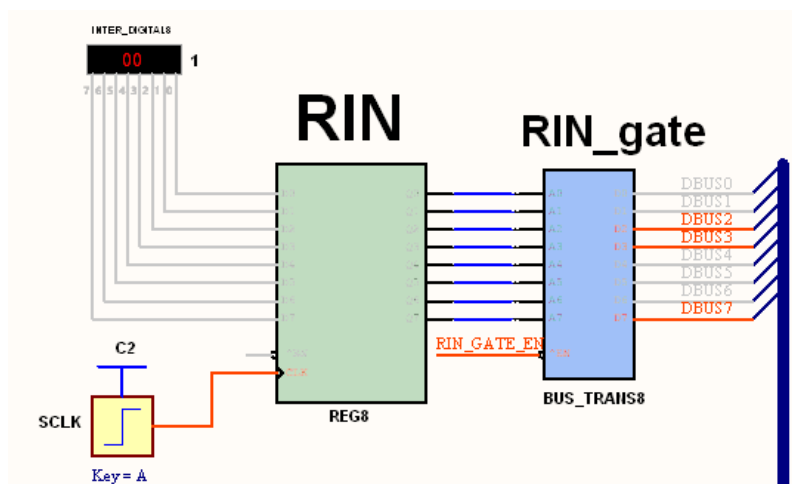


图 2-4：启动仿真后的寄存器 RIN

6. 确保鼠标焦点在原理图 DM1000.dlsche 中，然后通过按键“A”，向 RIN 寄存器管脚 CLK 端输入一个上升沿，将 0 写入 RIN 寄存器。
7. 观察：RIN 的输出均显示为灰色，表示 0。
8. 观察：寄存器窗口中 RIN 的十六进制值为 0x00，并且显示为红色，表示 RIN 寄存器发生了写操作。

9. 选中 RIN 左侧的八位交互式数字信号，在属性栏中将数字信号值修改为 0f，如下图 2-5 所示。

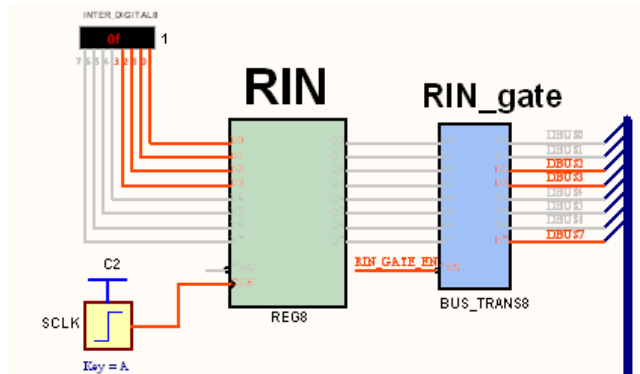


图 2-5: RIN 输出的值为 0

10. 按下按键“A”然后抬起，就会将 0f 写入 RIN 寄存器，观察写入后 RIN 寄存器的输出。
11. 将 0f 写入 RIN 寄存器后，请读者再将八位交互式数字信号输出的值修改为 ff，观察在没有时钟上升沿触发的情况下，ff 有写入 RIN 寄存器吗？
12. 选择菜单“仿真”中的“结束仿真”。

通过以上步骤可知，在寄存器使能的情况下，改变寄存器的输入并不会影响其输出，若想将数据写入寄存器中，还需要时钟上升沿的触发。

请读者在 DM1000 原理图中，添加一个“一位交互式数字信号”，与 RIN 的管脚 EN 连接，用来控制寄存器 RIN 的写使能。启动仿真后设置 RIN 寄存器的 EN 为高电平，通过按键 A 发送的时钟上升沿对 RIN 进行写操作，数据能写入 RIN 寄存器吗？

### 3.2 任务（二）通用寄存器 R0~R3 写入数据和读取数据

请读者继续使用完成任务（一）时在本地创建的项目，并按照下面的步骤完成本次实验：

#### 修改寄存器模块 REG 的电路

1. 打开项目子模块中的寄存器模块原理图文件 REG.dlsche。
2. 由于寄存器模块的输入数据和输出数据都是连接的数据总线 DBUS[7..0]，为了防止输入、输出信号相冲突，读者需要删除寄存器模块原理图中的所有名称为 DBUS0~DBUS7 的网络标签，还需要删除左右两个名称为 DBUS[7..0] 的端口。这样就使寄存器的输入信号网络与输出信号网络断开连接了。
3. 在原理图中添加一个“恒定高电平”信号源，将除了寄存器 R0~R3 之外的所有寄存器的写使能管脚 EN 与该“恒定高电平”信号源连接，并且将这些寄存器右侧的与之一对应的总线收发器的控制管脚 EN 也与该“恒定高电平”信号源连接，这样可以防止无关寄存器的写入和读取数据操作。
4. 为原理图中的输入端口 IR0、IR1、REG\_WR、REG\_READ 分别连接一个“一位交互式数字信号源”，并自定义每个信号的控制按键，以便仿真时进行手动控制。
5. 在原理图中添加一个“单周期时钟”源器件，将其与端口 CLK 连接，为各寄存器提供时钟。
6. 在原理图中添加一个“八位数字信号”源器件，将其用于输出数据的 8 个管脚与左侧的用于向寄存器写入数据的 8 条信号线进行连接，用于控制所有寄存器的输入数据。
7. 在原理图中添加两个十六进制七段数码管，将这两个器件的总共 8 个输入管脚与右侧用于向数据总线输出数据的 8 条信号线进行连接（注意连接顺序），用于显示通过总线收发器输出到总线上的值。

#### 向通用寄存器 R0~R3 写入数据

通过上述步骤，完成寄存器模块测试电路。下面，按步骤完成寄存器数据的写入：

1. 设置输入端信号 IR0=0，IR1=0，REG\_WR=0，REG\_READ=1，设置“八位数字信号”源器件的值为 1，然后单独对寄存器模块进行仿真。可看到此时只有寄存器 R0 的写使能 EN=0，手动输入时钟，将

数据 1 写入 R0 寄存器中。

$\sim 1G$	B	A	$Y_0$	$Y_1$	$Y_2$	$Y_3$
1	x	x	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0

表 2-2: 2-4 译码器 74LS139D 的真值表

2. 结合 2-4 译码器的真值表 2-2, 仿照寄存器 r0 的写入方法, 通过设置控制信号 IR1、IR0、REG\_WR, 以及“八位数字信号”源器件的值, 分别向寄存器 r1, r2, r3 中写入数据 0x11, -1 (使用补码表示为 0xff), 0x80。IR1, IR0 是指令的最后两位编码, 它们编码产生 4 个控制信号, 用来选择通用寄存器 r0~r3。

### 读出通用寄存器 R0~R3 的数据

通过控制信号 IR1、IR0、REG\_READ、REG\_WR, 依次读出寄存器 R0~R3 的数据, 读出的数据传送到输出数据总线上, 记录表格如下:

要求	REG_WR	REG_READ	IR1	IR0	输出到总线上的数据
将 R0 的值输出到总线上					
将 R1 的值输出到总线上					
将 R2 的值输出到总线上					
将 R3 的值输出到总线上					

表 2-3: 读寄存器 R0~R3 的控制信号

总结上述仿真实验, 可以得出以下结论:

- ① 将数据总线上的数据写入通用寄存器 R0~R3 时, 需要设置 REG\_WR 信号为低电平, 使能用于选择寄存器的译码器 U9, 并通过 IR0、IR1 进行译码从而使能一个通用寄存器, 当时钟上升沿触发时, 完成数据的写入。
- ② 将寄存器 R0~R3 的数据读出到数据总线上时, 需要设置 REG\_READ 信号为低电平, 使能用于选择输出门的译码器 U10, 并通过 IR0、IR1 进行译码从而使能一个输出门, 此时就会立即将输出门对应寄存器的值输出到数据总线上, 不需要时钟上升沿触发。
- ③ 当多个通用寄存器都有值时, 必须确保只有一个输出门被使能, 其余的输出门关闭, 从而实现将一个指定通用寄存器的值输出到数据总线的目的。
- ④ 打开控制单元 CU, 可以看到指令寄存器 IR 存放的是指令的 8 位编码, 仅使用其中的两位 IR0 和 IR1 编码四个通用寄存器 R0~R3, 这样就节省了指令编码的位数, 可以让指令编码中更多的位做别的事情。这种使用尽可能少的位数编码同一类的多个寄存器的方法在复杂指令集和精简指令集体系结构中均有使用。例如, Intel 8086 处理器的单操作数指令格式中就使用 3 位来编码 8 个十六位通用寄存器, 如表 2-4 所示。

编码	译码后使能的寄存器
000	AX
001	CX
010	DX
011	BX
100	SP
101	BP
110	SI
111	DI



## 提交作业

在提交作业前，读者需要将 REG 模块 SVG 图形文件的链接添加到 README.md 文件中，这样，当使用浏览器查看提交后的线上项目时，就可以从 README.md 文件中看到修改后的 REG 模块了。方法如下：

1. 双击“项目管理器”中的 README.md 文件，使用一种合适的文本编辑器或者 Markdown 编辑器打开此文件。
2. 在 README.md 文件的末尾添加如下的文本：  
# 寄存器模块 REG  
![raw svg](REG.dlsche.svg)
3. 保存 README.md 文件。
4. 提交作业。

### 3.3 任务（三）

前面的实验任务是对寄存器模块 REG 内部电路进行单独测试，重点是让读者学习 REG 模块的电路功能和实现原理。本次任务将通过为 DM1000 编写汇编程序，让 DM1000 运行寄存器读写指令的方式，完成对寄存器的读/写操作，使读者了解寄存器模块 REG 在整机系统中是如何配合其他功能模块完成功能的。

请读者按照下面方法之一在本地创建一个项目，用于完成本次任务：

#### 方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建个人项目，然后使用 Dream Logic 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

#### 方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，就需要使用 Dream Logic 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的 URL

为 <https://www.codecode.net/engintime/Dream-Logic/Project-Template/DM1000/DM1000.git>。

**建议教师按照下面的步骤在 CodeCode.net 平台上布置此次实验任务：**

- (1) 在计算机组成原理实验课程中，新建一个实验任务。
- (2) 在“新建任务”页面中，教师需要添写“任务名称”，推荐使用“实验 2 寄存器实验-任务（三）”；还需要填写“模板项目 URL”，应该使用  
“<https://www.codecode.net/engintime/Dream-Logic/Project-Template/DM1000/DM1000.git>”。
- (3) 点击“新建任务”按钮，完成新建任务操作。

请读者按照下面的步骤完成本次实验：

1. 打开项目下的汇编源程序文件 ram.asm，使用下面的源代码替换此文件中的原有内容。

```
.text
```

```
mov r0, 0 ; 将立即数 0 写入 r0 寄存器
```

```
mov r1, 1
```

```
mov r2, 2
```

```
mov r3, 3
```

```
mov a, 4 ; 将立即数 4 写入累加器 a，累加器 a 存在于算术逻辑运算模块 ALU 内部
```

```
add a, r2 ; 将累加器 a 与寄存器 r2 相加，结果写回 a 中
```

2. 在“项目管理器”窗口中右键点击批处理文件 ram.bat，在弹出菜单中选择“运行批处理文件”，将汇编源代码文件 ram.asm 编译为机器码文件 ram.rxm，列表文件 ram.lst，以及调试信息文件 ram.dbg。启动仿真后，机器码文件 ram.rxm 会被加载到 MEM 模块内的存储器 RAM 中。
3. 打开原理图文件 DM1000.dlsche，启动仿真（快捷键为 F5）。
4. 启动仿真后，如果正在使用自动时钟进行仿真，可以使用 S 键切换为单步时钟仿真。
5. 通过按键 R 完成复位。DM1000 利用 R 按下后输出的低电平程序计数器 PC 和微程序计数器 uPC 清零，从而完成复位操作。复位后，读者在源代码窗口 1 中可以看到蓝色箭头指向第一条指令，在



源代码窗口 2 中可看到蓝色箭头指向第一条微指令。

### 源代码窗口 1

1. 在左上角源代码窗口 1 中显示的是 ram.asm 编译后生成的列表文件 ram.lst。如图 2-6 所示：

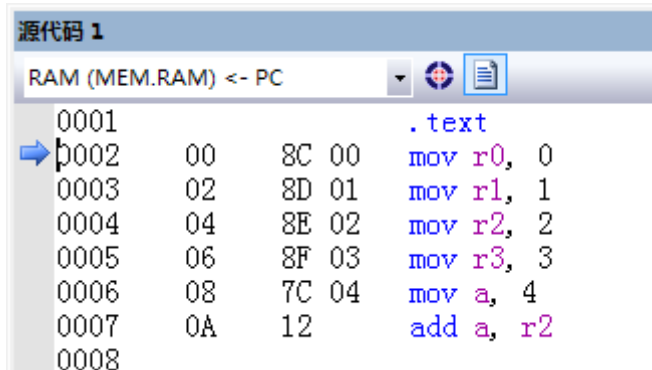


图 2-6：源代码窗口 1

2. 在窗口中可看到“movr0, 0”指令地址为 00，对应的机器码是两个字节“8C 00”，其中字节“00”就表示立即数 0。依次类推，后面几条 mov 指令中的立即数分别为 01、02、03、04。
3. 此外，我们注意到，指令“movr0, 0”到指令“movr3, 3”的第一个从 8C 依次加 1 直到 8F，这和指令中的寄存器有什么关系呢？结论将在后面揭晓。
4. 最后一条指令是单字节指令，它的机器码为 12。

### 取指微指令

下面，以第一条指令 mov r0, 0 为例，进行单步时钟仿真，从而分析该指令是如何执行的。

1. 启动仿真后，在源代码窗口 2 中，可看到蓝色箭头指向第一条微指令“path [pc], ir”，即取指微指令。
2. 双击打开 CU 模块，可看到 uPC 寄存器输出 0，ROM 输出取指微指令编码的 32 位控制信号。
3. 在 ROM 右侧的输出端口中，只有三个控制信号为低电平，其中有效的是 PC\_A\_GATE\_EN，MEM\_GATE\_EN。
4. PC\_A\_GATE\_EN=0，是因为取指令是将 PC 作为地址访问指令存储器 RAM，只有 PC 向地址总线输出的三态门打开，才能将 PC 值传送到地址总线 ABUS 上。双击当前视图中的空白处，返回顶层电路，然后双击 PC 模块进入内部，可以看到当前 PC=0，地址输出门 PC\_gate 使能 EN= PC\_A\_GATE\_EN = 0，PC 值通过输出门传送到地址总线 ABUS 上。
5. MEM\_GATE\_EN=0，是因为从指令存储器 RAM 读出指令后，只有打开存储器 RAM 向数据总线的输出门，才能将指令传送到数据总线 DBUS 上。返回顶层电路，然后进入 MEM 模块内部，可看到 RAM 的输入地址为 0 (PC=0)，W\R=1，读出地址 0 指向的存储单元内容 8C (指令)，这正是指令“movr0, 0”的第一个字节码。存储器输出门 MEM\_gate 使能打开，指令 8C 传送到数据总线 DBUS 上。到此，取指微指令的使命还没有结束。

通过前面的分析可知，取指微指令通过地址输出门和存储器输出门控制信号，将指令的第一个字节从存储器 RAM 读出并且输出到数据总线 DBUS 上。那么接下来 CU 模块如何使用 DBUS 上的指令呢？

1. 返回顶层电路并双击打开 CU 模块。
2. 找到指令寄存器 IR，可以看到，IR 写使能，且输入指令码为 8C，当下个时钟上升沿到来时，8C 将被写入 IR 中。
3. 将指令机器码写入指令寄存器只是 CU 模块完成的一个任务。CU 模块的另一个任务是根据 DBUS 上的指令机器码，计算出该指令对应的微程序入口地址。从 CU 视图可以看到，指令机器码 8C 的高 6 位扩展为 12 位地址 0x460 传送到 uPC 的输入端，当下个时钟到来时，uPC 将写入 0x460，从而执行该处

的微指令。

4. 手动输入单步时钟，可看到 IR 寄存器写入 8C，写入后 IR0 和 IR1 均为 0，这两位正是寄存器 r0 的编码。而 uPC 写入 0x460，在源代码窗口 2 中可以看到，蓝色箭头指向 460，这正是指令“movr0, 0”对应的第一条微指令。

通过上述仿真分析可知，CU 模块将指令保存到指令寄存器 IR 的同时，使得微程序计数器 uPC 转去执行指令对应的微程序，至此，取指微指令的使命才真正结束。

### 执行指令

取指完成后，接下来就该执行取出的指令了。在 DM1000 中，执行指令的本质是顺序运行该指令对应的固定微程序，当微程序运行完后，指令的功能也就实现了。

我们逐条运行指令“mov r0, 0”对应的微程序。

1. inc pc, 该条微指令将 PC 加 1，从而指向下一个字节。
  - ① 将鼠标移动到该条微指令的 PC 上可以看到当前 PC=0。
  - ② 打开 CU 模块，可看到当前 ROM 输出均为高电平，其中只有 PC\_ADD 信号有效，PC\_ADD 控制 PC 计数器的加计数使能，打开 PC 模块，可看到 PC 的计数使能端 EN=PC\_ADD 为高电平，说明允许 PC 加计数。
  - ③ 通过按键 C 输入一个时钟，可看到 PC 输出为 1。打开存储器窗口，PC=1，正好指向“movr0, 0”指令中的立即数 0 所在的存储单元。
2. path [pc], rx, 该条微指令将 PC 指向的存储单元内容读出并传送到通用寄存器中，微指令中的 rx 表示 R0~R3 中的某一个寄存器，具体是哪一个，则由指令寄存器 IR 中的 IR0 和 IR1 决定。
  - ① 打开 CU 模块，可看到当前 ROM 左端输出的控制信号中，有效的低电平信号为 REG\_WR, PC\_A\_GATE\_EN, MEM\_GATE\_EN。
  - ② 其中 PC\_A\_GATE\_EN, MEM\_GATE\_EN 两个门控信号配合，读出地址单元 1 的内容，即立即数 0，打开 MEM 模块进行观察验证。
  - ③ 立即数 0 读出后通过总线传送到目的寄存器的输入端，准备写入目的寄存器。打开寄存器模块 REG，可看到数据总线上的立即数为 0，IR0、IR1 和 REG\_WR 经过译码器输出唯一一个低电平信号，该信号使能 r0 寄存器。
  - ④ 通过按键 C 输入时钟，可看到寄存器 r0 写入了立即数 0。

经过以上两条微指令的执行，立即数 0 已经写入寄存器 r0 中，指令“mov r0, 0”的使命已经完成了，后面两条微指令分别实现 PC 加 1 和 uPC 复位，为下一条指令取指做准备。

至此，我们完成了指令“movr0, 0”的全过程分析，请读者完成后续的指令执行过程的分析。最后一条指令是加法指令，需要将寄存器 r2 中的值读取出来，请读者仿照将立即数写入寄存器的仿真过程分析读取寄存器的过程。

## 四、思考与练习

1. 各个寄存器对应的单向总线收发器使能信号是由控制单元 CU 发出的，任何时刻，最多只能有一个总线收发器向同一总线发送数据，避免信号冲突。DM1000 中挂接到数据总线 DBUS 上的总线收发器有哪些，哪些是接收总线数据的？哪些是向总线发送数据的？控制单元 CU 是如何控制总线收发器与数据总线 DBUS 间的数据传输的？

**提示：**控制单元 CU 发出的控制信号 X3、X2、X1、X0 四位二进制数可以编码 16 个总线收发器的使能。

在执行微指令的过程中，如果希望某个总线收发器向数据总线 DBUS 发送数据，就可以通过编码 X3、X2、X1、X0 决定该总线收发器使能，而其他的收发器不使能，这样数据总线 DBUS 在一个时钟周期中

总是只接收一个总线收发器发送的数据。请结合附录 A 中表 A-2 加以理解。

2. 在 DM1000 中查找所有与地址总线 ABUS 相连的总线收发器。分析 CU 是如何控制它们与地址总线 ABUS 传输地址的？
3. 请读者尝试说明为什么 DM1000 不支持类似 `mov r0, r1` 这样的指令。

## 实验 3 运算器实验

实验性质：验证+设计

建议学时：2 学时

### 一、实验目的

- 通过 ALU 模块电路的单独仿真，测试 ALU 的算术逻辑运算功能，掌握 ALU 内部电路原理。
- 通过编程调用 ALU 模块进行运算，掌握控制单元 CU 是如何控制 ALU 进行不同运算的。

### 二、预备知识

算术逻辑运算单元（ALU）74LS181 是一种功能较强的组合逻辑电路，它能进行多种算术运算和逻辑运算。它的基本逻辑结构是超前进位加法器。

$S_3 \sim S_0$ ：工作方式选择。

$M$ ：当  $M=1$  时，进行逻辑运算； $M=0$  时，进行算术运算。

$A=A_3 \sim A_0$ ,  $B=B_3 \sim B_0$ ：参加运算的两个数（注脚 3 表示最高位）。

$F_3 \sim F_0$ ：运算结果。

$CN$ ：当做加运算时，表示最低位进位输入， $CN=1$  时，无进位输入； $CN=0$  时，有进位输入。当做减法时，表示最低位借位， $CN=1$ ，有借位； $CN=0$ ，无借位。

$CN4$ ：最高位进位输出，低电平有效。

$AEQB$ ：当  $F_3 \sim F_0$  全为高电平时为 1，否则为 0。

$G$  称为进位发生输出， $P$  称为进位传送输出。它们是为了便于实现多芯片 ALU 之间的超前进位的。

方式	$M=1$ 逻辑运算	$M=0$ 算术运算	
$S_3 S_2 S_1 S_0$	逻辑运算	$CN=1$ (无进位)	$CN=0$ (有进位)
0 0 0 0	$F=A$ (非)	$F=A$ (直传)	$F=A$ 加 1
0 0 0 1	$F=/(A+B)$	$F=A+B$	$F=(A+B)$ 加 1
0 0 1 0	$F=(/A)B$	$F=A+/B$	$F=(A+/B)$ 加 1
0 0 1 1	$F=0$	$F=$ 负 1	$F=0$
0 1 0 0	$F=/(AB)$	$F=A$ 加 $A(/B)$	$F=A$ 加 $A/B$ 加 1
0 1 0 1	$F=/B$	$F=(A+B)$ 加 $A/B$	$F=(A+B)$ 加 $A/B$ 加 1
0 1 1 0	$F=A \oplus B$ (异或)	$F=A$ 减 $B$ 减 1	$F=A$ 减 $B$
0 1 1 1	$F=A/B$	$F=A(/B)$ 减 1	$F=A(/B)$
1 0 0 0	$F=/A+B$	$F=A$ 加 $AB$	$F=A$ 加 $AB$ 加 1
1 0 0 1	$F=/(A \oplus B)$	$F=A$ 加 $B$	$F=A$ 加 $B$ 加 1
1 0 1 0	$F=B$	$F=(A+/B)$ 加 $AB$	$F=(A+/B)$ 加 $AB$ 加 1
1 0 1 1	$F=AB$ (与)	$F=AB$ 减 1	$F=AB$
1 1 0 0	$F=1$	$F=A$ 加 $A$	$F=A$ 加 $A$ 加 1
1 1 0 1	$F=A+/B$	$F=(A+B)$ 加 $A$	$F=(A+B)$ 加 $A$ 加 1
1 1 1 0	$F=A+B$ (或)	$F=(A+/B)$ 加 $A$	$F=(A+/B)$ 加 $A$ 加 1
1 1 1 1	$F=A$ (直传)	$F=A$ 减 1	$F=A$ (直传)

表 3-1：74LS181 真值表

提示：1—高电平，0—低电平。中文的“加”和“减”表示算数运算。其他的符号表示逻辑运算，例

如 “/” 表示逻辑取反，“+” 表示逻辑或。

## 三、 实验内容

### 3.1 任务（一）

对 DM1000 中的 ALU 模块进行单独仿真测试，检验 ALU 的算术逻辑运算功能。

请读者按照下面方法之一在本地创建一个项目，用于完成本次任务：

#### 方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建个人项目，然后使用 Dream Logic 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

#### 方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，就需要使用 Dream Logic 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的 URL

为<https://www.codecode.net/engintime/Dream-Logic/Project-Template/DM1000/DM1000.git>。

**建议教师按照下面的步骤在 CodeCode.net 平台上布置此次实验任务：**

- (1) 在计算机组成原理实验课程中，新建一个实验任务。
- (2) 在“新建任务”页面中，教师需要添写“任务名称”，推荐使用“实验 3 运算器实验-任务（一）”；还需要填写“模板项目 URL”，应该使用  
“<https://www.codecode.net/engintime/Dream-Logic/Project-Template/DM1000/DM1000.git>”。
- (3) 点击“新建任务”按钮，完成新建任务操作。

请读者按照下面的步骤完成本次实验：

1. 打开项目下的原理图文件 ALU.dlsche，ALU 中的主要器件如下：
  - (1) 两片 74LS181 串行进位扩展成 8 位算术逻辑运算器。
  - (2) 累加器 A 和工作寄存器 W 的输入端直接与数据总线相连，在使能信号和时钟控制下，可将数据总线上的数据写入寄存器。
  - (3) 累加器 A 和工作寄存器 W 的输出分别作为运算器 ALU 的两个 8 位运算数。
  - (4) ALU 在控制信号的作用下进行不同的运算并通过直通门 D\_gate 输出运算结果。
2. 删除电路图中的网络标签 DBUS0~DBUS7，使输入数据网络与输出数据网络断开连接，避免输入、输出信号相冲突。
3. 在原理图中，使用型号库“数字信号源”中的元器件，为各个输入端口添加控制信号以及单周期时钟，使用 8 位交互式数字信号源作为寄存器 A 和 W 的数据输入。使用型号库“数字信号显示”中的 16 进制 7 段数码管显示 8 位 ALU 的运算结果，如图 3-1 所示：

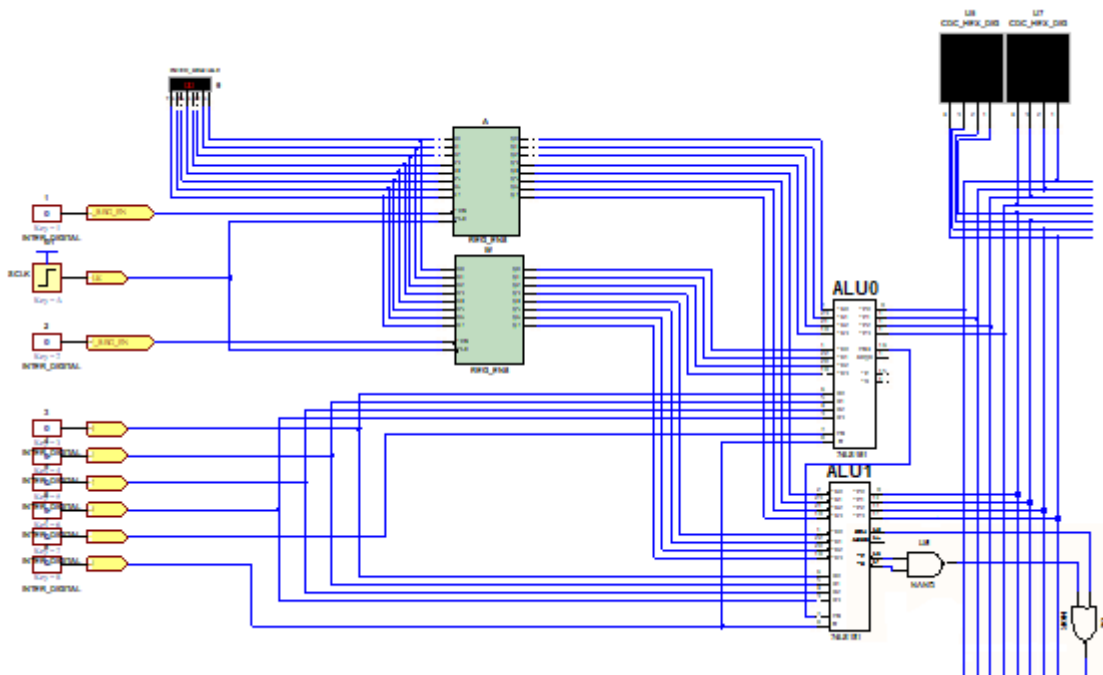


图 3-1：在 ALU 中添加控制信号以及单周期时钟

- 保存电路并单独仿真 ALU 模块。
- 参照表 3-1，分别向寄存器 A 和 W 中写入数据，A 作为第一运算数，W 作为第二运算数。然后通过各个控制信号手动控制 ALU 的运算方式，完成表 3-2。
- 完成表格后，结束仿真。

表达式	M	S <sub>3</sub> S <sub>2</sub> S <sub>1</sub> S <sub>0</sub>	C=1(无进位)			C=0(有进位)		
			计算结果	为 0	进位	计算结果	为 0	进位
7+2	0	1 0 0 1	0x09	否	否	0x0A	否	否
0x0F -5								
3& 0x0B								
4 8								
~7								
8 + (-1)								
1 + (-1)								
0x80 + 0x80								
6-9								

表 3-2：ALU 仿真记录表

**提示：**负数使用补码表示，因此“-1”表示为“0xff”。

通过以上实验，不难看出，要使 ALU 完成运算任务，首先需要将运算数分别写入累加器 A 和工作寄存器 W 中，与此同时，还需要通过控制信号指定 ALU 做何种运算。

由此可知，当 DM1000 执行运算指令时，首先需要将指令提供的操作数分别写入累加器 A 和 W 中，然后由指令类型输出对应的控制信号，控制 ALU 做指定的运算，最终将结果写回目的操作数寄存器中。

### 提交作业

在提交作业前，读者需要将 ALU 模块 SVG 图形文件的链接添加到 README.md 文件中，这样，当使用浏览器查看提交后的线上项目时，就可以从 README.md 文件中看到修改后的 ALU 模块了。方法如下：

- 双击“项目管理器”中的 README.md 文件，使用一种合适的文本编辑器或者 Markdown 编辑器打

开此文件。

2. 在 README.md 文件的末尾添加如下的文本：

```
# 运算器模块 ALU
![raw svg](ALU.dlsche.svg)
```

3. 保存 README.md 文件。
4. 提交作业。

### 3.2 任务（二）

使用 DM1000 完成一个减法运算，观察减法指令的执行过程。

请读者按照下面方法之一在本地创建一个项目，用于完成本次任务：

#### 方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建个人项目，然后使用 Dream Logic 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

#### 方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，就需要使用 Dream Logic 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的 URL

为 <https://www.codecode.net/engintime/Dream-Logic/Project-Template/DM1000/DM1000.git>。

**建议教师按照下面的步骤在 CodeCode.net 平台上布置此次实验任务：**

- (1) 在计算机组成原理实验课程中，新建一个实验任务。
- (2) 在“新建任务”页面中，教师需要添写“任务名称”，推荐使用“实验 3 运算器实验-任务（二）”；还需要填写“模板项目 URL”，应该使用  
“<https://www.codecode.net/engintime/Dream-Logic/Project-Template/DM1000/DM1000.git>”。
- (3) 点击“新建任务”按钮，完成新建任务操作。

请读者按照下面的步骤完成本次实验：

1. 打开项目下的汇编源程序文件 ram.asm，使用以下程序替换原程序：

```
.text      ;代码段
mov r0, -1 ;将立即数-1 写入寄存器 r0 中
mov a, 0   ;将立即数 0 写入累加寄存器 a 中
sub a, r0  ;将累加器 a 的值减去寄存器 r0 的值，结果写回 a 中
mov r0, a  ;将累加器 a 的值放入寄存器 r0，直通
```

2. 选择项目下的文件 ram.bat，单击 右键弹出菜单，选择“运行批处理文件”，对汇编源程序 ram.asm 进行编译。
3. 打开原理图 DM1000.slsche，启动仿真。
4. 通过按键 R 复位。
5. 通过按键 S，选择单步运行程序。
6. 通过按键 C 单步运行指令，当运行到减法指令时，停止单步时钟输入。打开寄存器窗口，选择十进制显示寄存器值，可看到寄存器 r0 的值为-1，双击定位到 r0，可看到 r0 的输出管脚均为高电平，在寄存器窗口中选择十六进制显示，可看到 r0 的值为 0xff，这正是-1 的 8 位补码表示。累加器 a 的值为 0，双击定位累加器 a，可看到累加器 a 的输出管脚均为低电平。

通过前面的步骤，我们将-1 写入了寄存器 r0 中，而将 0 写入了累加器 a 中，接下来是减法指令执行的详细过程。

1. 在仿真窗口中打开 ALU 模块，可看到此时工作寄存器 W 的输出值无效。由指令 suba, r0 可知，



需要将 r0 写入工作寄存器 W 后，ALU 才能进行运算。

2. 通过按键 C 输入时钟，执行取指微指令后，源代码窗口 2 中的蓝色箭头指向如图 3-2 所示：

0082					; sub a, rx
0083	180	EF FF FA 7F			path rx, w
0084	184	EF FF E4 86			path alu_sub, a
0085	188	FF FF FF FF			inc pc
0086	18C	CF FF FF FF			reset upc
0087					
0088	190	FF FF FF FF			dup 4, null
0089					

图 3-2：减法指令对应的微程序

3. 微指令“pathrx, w”是将源寄存器 rx (rx 表示 r0~r3 中的某一个寄存器) 的值传送到工作寄存器 w 中。打开控制单元 CU，可看到 ROM 输出的有效低电平控制信号为：
- ① REG\_READ，通用寄存器读使能信号。打开寄存器模块 REG，可看到当前读出到总线上的数据正是寄存器 r0 的值-1。
  - ② W\_REG\_EN，工作寄存器写使能信号。打开 ALU 模块，可看到 r0 寄存器的值通过总线传送到工作寄存器 W 的数据输入端，且 W 寄存器写使能，下个时钟上升沿到来时，-1 将被写入 W 中。
  - ③ 通过按键 C 输入单步时钟，寄存器 W 写入-1。
4. 在源代码窗口 2 中，可看到蓝色箭头指向微指令“pathalu\_sub, a”，该条微指令控制 alu 做减法，并使能累加器 a，为运算结果写回 a 中做准备。打开控制单元 CU，可看到 ROM 输出的有效控制信号为：
- ① S0S1S2S3=0110，C 和 M 均输出低电平，查询表 3-1 可知，此时 ALU 做减法运算“A 减 B”。
  - ② A\_REG\_EN 为电平，累加器 A 写使能。
  - ③ D\_GATE\_EN 为低电平，ALU 的运算结果直接输出到总线 DBUS 上。
  - ④ 打开 ALU 模块，可看到在上述信号控制下，ALU 进行减法运算“0- (-1)”，运算结果为 1，通过直通门 D\_gate 输出到总线 DBUS 上，且累加器 A 写使能。
  - ⑤ 通过按键 C 输入单步时钟，可看到累加器 a 写入了运算结果 1。到此，减法指令 suba, r0 的使命结束了。
5. 继续单步运行，后面两条微指令完成 PC 加 1 和复位 uPC 的任务，为下一条指令取指做准备。
6. 接下来是将累加器 a 的值放入寄存器 r0 的指令，请读者自行仿真，重点理解运算器直通的工作方式。
7. 最后，由于 PC 加 1 后指向的存储单元没有指令，而是初始化后的 0，因此，取指时使得 uPC 始终为 0，DM1000 进入重复取指状态。此时可结束仿真。

通过分析减法指令的执行过程可知，取指后，通过两步完成了减法运算，第一步将操作数传送到工作寄存器 W 中，第二步控制 ALU 进行减法运算，然后利用下个时钟上升沿将运算结果写回累加器 a 中。

减法指令的每一步都由微指令进行控制，在这里，我们可以更清晰的看到，微指令的实质是若干有效控制信号的组合，这些有效信号控制 DM1000 完成一个特定的最小操作，通过这些微操作的顺序进行，最终实现指令的功能。

修改汇编程序文件 ram.asm，编写一段简单的汇编程序用于计算下面的运算表达式，要求将最后的计算结果保存到 R0 寄存器中。

**表达式：**

$$7+6*2-(4|2)$$

仿真运行编写的汇编程序，确保其可以正常运行。记录移位运算时运算器模块输入的控制信号，并描述运算器模块中的电路是如何进行移位运算的。记录逻辑或运算时运算器模块输入的控制信号。



请读者按照下面的提示编写汇编程序：

1. 可能用到的指令包括加法指令 ADD，减法指令 SUB，逻辑或指令 OR，指令格式请参考附录 A 中的表 A-4。
2. 可使用逻辑左移一位指令 SHL 计算乘 2，指令格式请参考附录 A 中的表 A-4。

## 四、思考与练习

1. 参看附录 A 的内容，在 DM1000 指令集中找出算术逻辑运算类指令。比如 add 是算术运算指令，not 是逻辑运算指令，试着使用这些指令编写程序。然后运行程序，观察指令执行过程中何时 ALU 进行了算术逻辑运算，此时对应的微指令是什么？在 ALU 模块中查看输入的控制信号，此时控制信号使 ALU 做何种运算。
2. 在程序的执行过程中，观察 ALU 模块中哪些输入信号是由控制单元 CU 发出的，并结合附录 A 中的控制信号列表，理解各个控制信号的作用。
3. DM1000 的通用寄存器是八位的，但是仍然可以进行十六位的算术运算，方法是先使用 add 指令计算低八位，再使用 adc 指令（该指令会考虑进位或借位）计算高八位。例如，下面的程序就可以用来计算两个数 0x017b 与 0x3a24 的和，将结果低八位放入 R0 寄存器，将高八位放入 R1 寄存器：

```
.text
mov r0, 0x7b
mov r1, 0x01
mov r2, 0x24
mov r3, 0x3a
mov a, r0
add a, r2
mov r0, a
mov a, r1
adc a, r3
mov r1, a
```

请读者尝试运行上面的程序，看看计算的结果是否正确。如果结果错误，请在 ALU 模块中查找原因，并修正电路图。然后确保可以同时运行上面的程序，还可以运行下面的程序求 0x0180 与 0x3acd 的和：

```
.text
mov r0, 0x80
mov r1, 0x01
mov r2, 0xcd
mov r3, 0x3a
mov a, r0
add a, r2
mov r0, a
mov a, r1
adc a, r3
mov r1, a
```

4. DM1000 目前实现的 CF 进位标志是通过 CN4、P 和 G 管脚做组合逻辑实现的，只能正确处理 add 加法指令产生的进位标志，还无法处理 sub 减法指令产生的借位标志。请读者继续改进 CF 标志位的设计，当进行无符号减法运算时，若减数大于被减数，此时有借位，则 CF 值为 1，否则 CF 值为 0。如果读者改进了 CF 标志位的设计，请参考思考与练习 3 的方式，编写一段汇编代码，确保 sbb 指令可以正常处理有借位的减法。

5. 目前 DM1000 只是通过网络连接时的移位方式实现了移位指令，所以只能移动一位。一般在运算器中是通过桶形移位器实现移位功能的，在 Drean Logic 的型号库“桶形移位器和总线收发器”中提供了八位桶形移位器“SHIFT8”。请读者自行查阅资料，学习桶形移位器的工作方式，并尝试在运算器模块中放置一个八位桶形移位器，从而实现功能更加强大的移位指令。

## 实验 4 程序计数器实验

实验性质：验证+设计

建议学时：2 学时

### 一、实验目的

- 掌握程序计数器 PC 的内部电路原理。
- 掌握程序计数器 PC 在程序运行过程中的作用。

### 二、预备知识

程序计数器 PC 在计算机内部存放现行指令的地址，通常具有计数功能，可实现顺序寻址和跳跃寻址。其 PC+1 的操作可完成顺序寻址功能，跳跃寻址则通过转移类指令修改 PC 的值来实现。

程序计数器模块 PC 中的核心器件是 8 位同步置数/异步清零二进制计数器 PC，它的功能如表 4-1 所示。

CLR	$\bar{\text{LOAD}}$	EN	CLK	Q	功能
0	X	X	X	0	PC 复位
1	0	X	$\uparrow$ （上升沿）	Q=D	同步置数，PC 跳转
1	1	0	X	保持	保持
1	1	1	$\uparrow$ （上升沿）	Q=Q+1	加计数

表 4-1：8 位同步置数/异步清零二进制计数器真值表

程序计数器 PC 是计算机中非常重要的器件，PC 的基本功能：

- 复位。PC 清零，从第一条指令开始执行。
- 地址预置。程序跳转时转移地址的置入。
- 自动加 1。保证程序的顺序执行。
- 地址输出。提供取指令地址。

在 DM1000 中，对于双字节指令，第一个字节作为指令取出后写入指令寄存器 IR 中，并由该字节高 6 位得到对应微程序的入口地址；而第二个字节是指令的操作数。

主存储器 RAM 一次只能输出一个字节，所以，PC 不总是指向指令，也可能指向双字节指令的第二个字节。例如指令“mov r0, 16”是双字节指令，指令的第二个字节表示立即数 16，读取立即数 16 时，PC 指向该立即数。

程序计数器 PC 模块的输入控制信号如下：

- ① 当 RESET=0 时计数器被清零，输出 0 地址；
- ② 当 LOAD=0 时，在时钟上升沿到来时，输入端数据 D0~D7 写入 PC 计数器；
- ③ 当 PC\_ADD=1 时，计数使能，在时钟上升沿到来时，PC+1；
- ④ 当 PC\_A\_GATE\_EN=0 时，PC 地址输出门打开，PC 输出到地址总线上；
- ⑤ 当 PC\_D\_GATE\_EN=0 时，PC 向数据总线输出门打开，PC 输出到数据总线上；
- ⑥ 当 PC\_LOAD\_EN=1 时，不允许 PC 被预置；
- ⑦ 当 PC\_LOAD\_EN=0 时，PC 计数器 LOAD 端由 IR<sub>3</sub>, IR<sub>2</sub>, CF, ZF 决定。IR<sub>3</sub>, IR<sub>2</sub> 分别是指令的第四位和第三位。IR<sub>3</sub>IR<sub>2</sub> 组合决定了不同的跳转方式：
  - IR<sub>3</sub>IR<sub>2</sub> = 11 或 10 时，LOAD=0，PC 被预置，对应无条件跳转 JMP xx（转移地址）
  - IR<sub>3</sub>IR<sub>2</sub> = 00 时，LOAD= $\bar{\text{CF}}$ ，若 CF=1，LOAD=0，PC 被预置，对应有进位则转移的情况 JC xx

- $IR_3IR_2 = 01$  时,  $LOAD = \sim ZF$ , 若  $ZF=1$ ,  $LOAD=0$ , PC 被预置, 对应结果为 0 则转移的情况 JZ xx

### 三、实验内容

#### 3.1 任务（一）

通过仿真测试程序计数器 PC 模块各输入控制信号的作用, 使读者对 PC 模块功能以及电路原理有深入的理解。

请读者按照下面方法之一在本地创建一个项目, 用于完成本次任务:

##### 方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务, 从而在 CodeCode.net 平台上创建个人项目, 然后使用 Dream Logic 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

##### 方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台, 就需要使用 Dream Logic 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中, 实验模板的 URL

为 <https://www.codecode.net/engintime/Dream-Logic/Project-Template/DM1000/DM1000.git>。

**建议教师按照下面的步骤在 CodeCode.net 平台上布置此次实验任务:**

- (1) 在计算机组成原理实验课程中, 新建一个实验任务。
- (2) 在“新建任务”页面中, 教师需要添写“任务名称”, 推荐使用“实验 4 程序计数器实验-任务（一）”; 还需要填写“模板项目 URL”, 应该使用  
“<https://www.codecode.net/engintime/Dream-Logic/Project-Template/DM1000/DM1000.git>”。
- (3) 点击“新建任务”按钮, 完成新建任务操作。

请读者按照下面的步骤改造 PC 模块的原理图:

1. 打开原理图 PC.dlsche。
2. 删除原理图中的网络标签 DBUS0~DBUS7, 断开输入信号网络与输出信号网络的连接, 防止输入、输出信号相冲突。
3. 在原理图中添加一个“8 位交互式数字信号”源器件, 将其输出管脚 0~7 分别与 PC 的输入管脚 D0~D7 通过网络连接, 作为 PC 的数据输入。
4. 将 PC 的输出管脚与两个 16 进制 7 段数码管连接, 用于显示 PC 的输出。
5. 在原理图中, 为 RESET 和 CLK 信号分别接一个单周期时钟器件, 为其它控制信号分别连接一个“一位交互式数字信号”器件, 这样就可以通过这些数字信号源器件来控制 PC 模块的工作方式了。

请读者按照下面的步骤对 PC 模块进行仿真:

1. 启动仿真。
2. 使用单周期时钟器件将复位信号 RESET 置为低电平再恢复为高电平, 这样当 RESET 为低电平时就会将 PC 计数器清零, 恢复为高电平时 PC 清零信号无效, 就可以继续进行后续的工作了。
3. 设置 PC\_LOAD\_EN 为高电平, PC\_ADD 为高电平, 允许 PC 加计数。
4. 输入单周期时钟, 使 PC 加计数, 直到 PC 值为 5。
5. 设置 RESET 为低电平, 可看到 PC 被清零, 这就是为什么程序运行过程中, 一旦复位, 立即从第一条指令开始重新运行程序。

##### 无条件跳转测试

1. 将 RESET, PC\_LOAD\_EN, PC\_ADD 设置为高电平, 然后连续输入单周期时钟, 直到 PC=4。
2. 设置 PC 的输入 D0~D7 为 0x15, 准备跳转地址。
3. 设置 IR2、IR3 为高电平, 对应无条件跳转指令。

4. 设置 PC\_LOAD\_EN 为低电平，可看到 PC 的 LOAD 输入低电平，在输入一个单周期时钟后，输入端的跳转地址 0x15 写入了计数器 PC。
5. 设置 PC\_LOAD 为高电平，PC 置数端 LAOD 为高电平，置数信号无效。
6. 输入单周期时钟，PC 加 1。
7. 继续输入单周期时钟，直到 PC=0x18。

### 有进位则跳转

1. 设置 PC 的输入 D0~D7 为 0xf0，准备进位跳转地址。
2. 设置 IR2、IR3 为低电平，对应有进位则转移指令。
3. 设置 CF 为高电平，表示有进位。
4. 设置 PC\_LOAD\_EN 为低电平，可看到 PC 的 LOAD 输入低电平，在输入一个单周期时钟后，输入端的跳转地址 0xf0 写入了计数器 PC。
5. 将 CF 置为低电平，表示无进位。可看到此时 PC 的 LOAD 输入高电平，置数信号无效。
6. 将 PC 的输入改为 0x10，在输入一个单周期时钟后，PC 没有写入新的值 0x10，而是进行了加一操作。这说明，在没有进位标志时，不满足跳转的条件，转移地址将不会被加载到 PC，程序不会跳转，而是继续执行下一条指令。

### 结果为 0 则跳转

仿照有进位则跳转的测试方法，设计实验步骤，测试结果为 0 则 PC 跳转的功能。

## 提交作业

在提交作业前，读者需要将 PC 模块 SVG 图形文件的链接添加到 README.md 文件中，这样，当使用浏览器查看提交后的线上项目时，就可以从 README.md 文件中看到修改后的 PC 模块了。方法如下：

1. 双击“项目管理器”中的 README.md 文件，使用一种合适的文本编辑器或者 Markdown 编辑器打开此文件。
2. 在 README.md 文件的末尾添加如下的文本：  
# 程序计数器模块 PC  
![raw svg](PC.dlsche.svg)
3. 保存 README.md 文件。
4. 提交作业。

## 3.2 任务（二）

通过前面的实验，读者对 PC 模块功能有了初步的了解。本任务将通过让 DM1000 运行一段汇编程序，考察 PC 模块是如何发挥其作用的。

请读者按照下面方法之一在本地创建一个项目，用于完成本次任务：

### 方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建个人项目，然后使用 Dream Logic 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

### 方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，就需要使用 Dream Logic 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的 URL

为<https://www.codecode.net/engintime/Dream-Logic/Project-Template/DM1000/DM1000.git>。

**建议教师按照下面的步骤在 CodeCode.net 平台上布置此次实验任务：**

- (1) 在计算机组成原理实验课程中，新建一个实验任务。
- (2) 在“新建任务”页面中，教师需要添写“任务名称”，推荐使用“实验 4 程序计数器实验-任务（二）”；

还需要填写“模板项目URL”，应该使用

“<https://www.codecode.net/engintime/Dream-Logic/Project-Template/DM1000/DM1000.git>”。

(3) 点击“新建任务”按钮，完成新建任务操作。

请读者按照下面的步骤完成本次实验：

1. 打开项目下的源程序文件 ram.asm，输入以下程序并保存。其中“end:”是一个标号，它表示接下来那行指令的偏移地址。标号通常作为跳转指令的操作数，表示要跳转执行的位置。

```
.text      ;代码段
mov r0, 4   ;将立即数 4 移到通用寄存器 r0 中
mov a, 8    ;将立即数 8 移到累加寄存器 a 中
jmp end     ;程序跳转到 end 处执行
and a, r0   ;将累加寄存器 a 的值与寄存器 r0 的值进行与运算，结果写回累加器 a 中
```

```
end:
add a, r0   ;将累加寄存器 a 的值与寄存器 r0 的值相加，结果写回累加器 a 中
```

2. 运行批处理文件 ram.bat，编译源程序。
3. 打开原理图文件 DM1000.dlsche，启动仿真。
4. 通过按键 S 切换为单步运行程序。

观察第一条指令的执行过程：

1. 在源代码窗口 1 中，可看到当前执行指令 movr0, 4。
2. 在源代码窗口 2 中，可看到当前执行取指微指令。
3. 打开 PC 模块，PC 的置数以及清零端均为高电平，无效。
4. PC 的计数使能端 EN 为低电平，禁止计数。
5. 打开控制单元 CU，从 ROM 输出的控制信号中找出 PC\_ADD 和 PC\_LOAD\_EN。
6. 输入单周期时钟，PC 保持不变，说明取指微指令不会改变 PC。此时 PC 使能输入端 EN 为高电平，允许加计数，在源代码窗口 2 中可以看到，当前对应的是微指令 incpc。
7. 输入单周期时钟，PC 加 1，指向指令 movr0, 4 的第二个字节。
8. 输入单周期时钟，立即数 4 写入 r0 中。
9. 输入单周期时钟，PC 加 1，指向下一条指令。

通过上述仿真可知，DM1000 通过运行 inc pc 微指令实现 PC 加计数，从而顺序执行指令。继续单步运行，直到取出指令 jmpend，下面分析无条件跳转指令是如何实现跳转的。

1. 取出 jmpend 指令后，源代码窗口 2 中的蓝色箭头指向微指令 incpc。
2. 打开 CU 模块，可看到指令寄存器 IR 中已经写入了 jmpend 指令的第一个字节 AC，且 IR2、IR3 均为高电平，正是无条件跳转的编码，已经为跳转做好准备。
3. 输入单周期时钟，PC 加 1，此时 uPC 指向微指令 path [pc], pc，如图 4-1 所示。

0329				; jmp symbol
0330	560	FF FF FF FF		inc pc
0331	564	EF 6F F9 FF		path [pc], pc
0332	568	CF FF FF FF		reset upc
0333				
0334	56C	FF FF FF FF		dup 5, null
0335				

图 4-1：无条件跳转指令

4. 打开 RAM 存储器窗口，可以看到，当前 PC 指向的存储单元保存的正是无条件跳转地址 07。

5. 打开 PC 模块, 可以看到 PC 的同步置数端 LOAD 为低电平, 置数信号有效。置数信号之所以有效, 是因为 PC\_LOAD\_EN 为低电平, IR2, IR3 均为高电平, 8 选 1 数据选择器选择 D3 输出到 Y 端, 输出 Y 取反后输出到 W 端, 8 选 1 的 D3 输入高电平, 因此 LOAD 为低电平。
6. 可看到 PC 的输入端数据 D0~D7 正是跳转地址, 打开存储器模块 MEM, 可以看到该跳转地址是从存储器 RAM 的 5 号地址单元读出的。
7. 打开 PC 模块, 输入单步时钟, PC 加载跳转地址 07, 于是跳转到指定位置运行。
8. 单步运行, 直到程序执行完毕。
9. 结束仿真。

通过上述仿真实验, 我们分析了程序运行过程中顺序取指与无条件跳转的具体实现过程, 简单地说, 顺序取指是通过执行微指令 incpc 实现的, 而程序跳转是通过执行微指令 path [pc], pc 实现的, 具体属于哪一种跳转, 则是由指令编码的 IR2 和 IR3 决定的。

### 使用 JZ 指令替换 JMP 指令

改造源程序 ram.asm, 使用为零则跳转指令 jz 替换 jmp 指令, 通过仿真, 分析 jz 指令是如何实现跳转的。

#### 提示:

- (1) 为零则跳转指令 jz 是条件转移指令, 发生跳转的条件是标志寄存器 FLAG 中的零标志位 ZF 有效, 即 ZF 为高电平。
- (2) 为了使 ZF 为高电平, 那么就需要在 jz 指令之前通过执行一条运算指令, 并使运算结果为 0。

### 使用 JC 指令替换 JMP 指令

改造源程序 ram.asm, 使用有进位则跳转指令 jc 替换 jmp 指令, 通过仿真, 分析 jc 指令是如何实现跳转的。

#### 提示:

- (1) 有进位则跳转指令 jc 是条件转移指令, 发生跳转的条件是标志寄存器 FLAG 中的进位标志 CF 有效, 即 CF 为高电平。
- (2) 为了使 CF 为高电平, 可在 jc 指令之前执行一条加法运算指令, 并使运算结果产生进位。

## 四、思考与练习

1. 指令机器码中的 IR2 和 IR3 这两个位, 既用于在 ROM 中定位微指令的偏移位置, 又用于控制跳转方式, 所以就注定了 JC 指令的第一条微指令的偏移地址中的 5、6 位 (从 0 开始) 是 00, JZ 指令的是 01, 而 JMP 指令的是 11, 所以在 JMP 指令和 JZ 指令之间又填充了 32 个字节。读者可以在启动仿真后, 在微指令的源代码窗口中找到这 3 条跳转指令, 并检验他们的第一条微指令的偏移地址。再请读者考虑一个问题, 如果让指令机器码中的 IR2 和 IR3 这两个位只用于控制跳转方式, 而不用在 ROM 中定位微指令的偏移位置的话, 那么指令中还剩下多少位用于在 ROM 中定位微指令的偏移位置? 它们能寻址多少条指令的微指令呢? 还够用吗?
2. 目前, jmp、jz、jc 指令都是跳转到绝对地址, 也就是说在这些指令的第二个字节记录的都是绝对地址。首先, 请读者考虑一下这种跳转到绝对地址的弊端, 例如指令在内存中的位置被移动后 (在一个受限的计算机系统内经常会这样做!), 这些跳转指令还能正常执行吗? 然后, 请读者修改 PC 模块的电路, 当发生跳转时, 不再将数据总线上的数据直接给 PC 寄存器, 而是将数据总线上的数据和当前 PC 的值相加后, 再给 PC 寄存器, 从而使这些跳转指令可以跳转到相对地址。注意, 除了修改 PC 模块的电路外, 读者还需要修改汇编器的 C 源代码, 在这些跳转指令的第二个字节中保存相对地址, 而不再是绝对地址, 即将 dmasm.c 文件中第 1884 行替换为  
`“machine_code[reallocate_table[i].address] = (BYTE)symbol_table[j].address -  
reallocate_table[i].address;”。`

# 实验 5 存储器实验

实验性质：验证

建议学时：2 学时

## 一、实验目的

- 掌握主存储器模块 MEM 的读写访问。
- 掌握主存储器模块 MEM 在程序运行过程中的作用。

## 二、预备知识

根据在计算机系统中的作用，**存储器**可分为主存储器、辅助存储器、缓冲存储器。

主存储器（主存）的主要特点是它可以和 CPU 直接交换信息。辅助存储器（辅存）是主存储器的后援存储器，用来存放当前暂时不用的程序和数据，它不能与 CPU 直接交换信息。主存拥有速度快、容量小、每位价格高的特点。缓冲存储器（缓存）用在两个速度不同的部件之中，例如 CPU 和主存之间可以设置一个快速缓存起到缓冲作用。

在 DM1000 中，存储器模块 MEM 扮演主存储器的角色，它用来存储指令和数据，可以读或写。

## 三、实验内容

### 3.1 任务（一）

掌握主存储器模块 MEM 的工作原理，具体地说，MEM 模块在 DM1000 中承担两个功能：

- （1） 读。从存储器读取指令或数据。
- （2） 写。将程序执行过程中的数据或地址信息写入存储器。

本任务通过对 MEM 模块内部电路的仿真，使读者掌握存储器的读、写原理。

请读者按照下面方法之一在本地创建一个项目，用于完成本次任务：

#### 方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建个人项目，然后使用 Dream Logic 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

#### 方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，就需要使用 Dream Logic 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的 URL

为<https://www.codecode.net/engintime/Dream-Logic/Project-Template/DM1000/DM1000.git>。

**建议教师按照下面的步骤在 CodeCode.net 平台上布置此次实验任务：**

- （1） 在计算机组成原理实验课程中，新建一个实验任务。
- （2） 在“新建任务”页面中，教师需要添写“任务名称”，推荐使用“实验 5 存储器实验-任务（一）”；还需要填写“模板项目 URL”，应该使用  
“<https://www.codecode.net/engintime/Dream-Logic/Project-Template/DM1000/DM1000.git>”。
- （3） 点击“新建任务”按钮，完成新建任务操作。

请读者按照下面的步骤完成本次实验：

1. 打开项目下的原理图文件 MEM.dlsche。



2. 在原理图中分别找出 MEM 的各个输入输出端口，结合以下说明，熟悉电路：

- (1) 端口 ABUS[7..0]，单向输入地址总线，为访问存储器提供地址。
- (2) 端口 DBUS[7..0]，双向输入/输出数据总线。写入时，外部数据通过该总线传送到存储器数据输入端；读出时，从存储器读出的指令或数据通过该总线传送到其他部件。
- (3) 端口 MEM\_WR，单向输入控制信号，控制存储器的读写。当 MEM\_WR 为低电平时，在输入时钟上升沿的触发下，可将数据写入存储器。当 MEM\_WR 为高电平时，RAM 总是输出地址指定单元的内容。
- (4) 端口 MEM\_GATE\_EN，存储器输出门控制信号。当 MEM\_GATE\_EN 为低电平时，输出门 MEM\_gate 打开，从存储器读出的指令或数据就可以通过输出门传送到数据总线上。当 MEM\_GATE\_EN 为高电平时，输出门关闭，存储器 RAM 的输出与总线隔离。
- (5) 端口 CLK，时钟信号。当存储器写使能，也就是 MEM\_WR 为低电平时，输入数据不能立即写入存储器，还需要等待时钟上升沿到来时才能写入。
- (6) 从原理图中不难看出，存储单元 MEM 的输入数据总线与输出数据总线是同一条。也就是存储器写入的数据是通过外部总线传送过来的，同时，从存储器读出的指令或数据也是通过外部总线传送到其它部件的。

读者对 MEM 的各个输入输出端口有了基本的了解后，就可以在原理图中添加相应的输入信号，通过仿真模拟存储器的读写。

1. 删除图中的网络标签 DBUS0~DBUS7，断开输入信号和输出信号的连接，避免信号冲突。
2. 分别使用 8 位交互式数字信号源作为存储器的地址和数据输入，使用一位交互式数字信号源作为存储器的读写控制信号，使用单周期时钟作为 RAM 的 CLK 输入。
3. 启动仿真。
4. 将 RAM 的读写控制信号 MEM\_WR 设置为高电平，读存储器。
5. 修改存储器 RAM 的输入地址，使地址从 0 递增到 8，依次读出每一个存储单元的内容，并通过存储器窗口显示的存储器内容进行比较验证，完成表格 5-1。

地址（两位十六进制表示）	存储器 RAM 输出值（使用十六进制表示）
00	
01	
02	
03	
04	
05	
06	
07	
08	

表 5-1：存储器 RAM 的读出

以上步骤完成了对存储器 RAM 的读操作，下面对存储器 RAM 进行写操作。

1. 将存储器写信号置为有效，即 MEM\_WR 设为低电平。
2. 将输入地址设置为 8。
3. 将输入数据设置为 8。
4. 输入一个时钟上升沿，在存储器窗口中可以看到，地址为 08 的单元显示内容为 08，文本为红色，说明数据 08 已经写入存储器 08 单元。
5. 仿照以上方法，将地址和数据递减，逐个写入。
6. 写入完成后，将 MEM\_WR 设为高电平，依次读出地址 00~08 指向存储单元的数据，验证写入数据的正确性。

## 7. 结束仿真。

**提交作业**

在提交作业前，读者需要将 MEM 模块 SVG 图形文件的链接添加到 README.md 文件中，这样，当使用浏览器查看提交后的线上项目时，就可以从 README.md 文件中看到修改后的 MEM 模块了。方法如下：

1. 双击“项目管理器”中的 README.md 文件，使用一种合适的文本编辑器或者 Markdown 编辑器打开此文件。
2. 在 README.md 文件的末尾添加如下的文本：  
# 存储器模块 MEM  
![raw svg](MEM.dlsche.svg)
3. 保存 README.md 文件。
4. 提交作业。

**3.2 任务（二）**

使用 DM1000 运行汇编程序，观察从存储器取指令和数据，以及将数据写入存储器的过程。

请读者按照下面方法之一在本地创建一个项目，用于完成本次任务：

**方法一：从 CodeCode.net 平台领取任务**

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建个人项目，然后使用 Dream Logic 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

**方法二：不从 CodeCode.net 平台领取任务**

如果读者不使用 CodeCode.net 平台，就需要使用 Dream Logic 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的 URL

为<https://www.codecode.net/engintime/Dream-Logic/Project-Template/DM1000/DM1000.git>。

**建议教师按照下面的步骤在 CodeCode.net 平台上布置此次实验任务：**

- (1) 在计算机组成原理实验课程中，新建一个实验任务。
- (2) 在“新建任务”页面中，教师需要添写“任务名称”，推荐使用“实验 5 存储器实验-任务（二）”；还需要填写“模板项目 URL”，应该使用  
“<https://www.codecode.net/engintime/Dream-Logic/Project-Template/DM1000/DM1000.git>”。
- (3) 点击“新建任务”按钮，完成新建任务操作。

请读者按照下面的步骤完成本次实验：

1. 打开项目下的汇编源程序文件 ram.asm，复制下列程序到 ram.asm 中并保存：

```
.text
mov a, 7      ;将立即数 7 移到累加器 a 中
mov r2, 0xd0  ;将立即数 0xd0 移到通用寄存器 r2 中
mov [r2], a   ;将累加器 a 的值保存到 r2 指定的存储器单元(0xd0)中
```

2. 运行批处理文件 ram.bat，编译源程序。
3. 打开项目下的原理图文件 DM1000.dlsche，启动仿真。
4. 通过按键 S 切换为单步运行。

接下来，我们以第一条指令 mov a, 7 为例，分析从存储器取指令和数据的过程。

1. 通过源程序窗口（源代码窗口 1）可以看到，当前正在执行指令 mov a, 7。
2. 通过微程序窗口（源代码窗口 2）可以看到，当前正在执行取指微指令。按下列说明进行观察，分析取指令过程：

① 打开存储器模块 MEM，可看到 MEM\_WR 为高电平，说明取指微指令对存储器 RAM 进行读操作；

- ② 当前输入地址为 0，这个地址是由 PC 提供的，打开 PC 模块，可看到程序计数器 PC 的值通过输出门 PC\_gate 输出到了地址总线 ABUS 上。
  - ③ 打开 MEM 模块，从 0 地址读出的指令 0x7C 通过存储器输出门传送到了数据总线上。说明指令已经读出，此时所有与数据总线连接的模块都能从数据总线上获取指令。
  - ④ 打开真正使用指令的模块 CU，可看到指令寄存器 IR 使能，当前指令高 6 位扩展后得到的对应微程序入口地址传送到 uPC 输入端。
  - ⑤ 输入单步时钟，可看到指令 0x7C 写入指令寄存器 IR，与此同时 uPC 写入指令对应微程序的入口地址 0x3E0。
3. 取出指令后，首先执行的是指令“mov a, 7”对应的第一条微指令“incPC”。输入单步时钟后，可看到存储器窗口中，PC=1，指向“mov a, 7”的第二个字节“07”，这是指令中的立即数操作数。
4. 执行完 incpc 后，微指令“path [pc], a”读出立即数并将立即数写入累加器 a 中。打开存储器模块 MEM，可看到，从存储器读出地址 1 指向的字节“07”；打开 ALU 模块，累加器 a 使能，输入单步时钟，立即数 7 写入累加器 a 中。

指令 mov a, 7 中包含了取指令和读立即数，需要两次对存储器 RAM 进行读访问。那么，存储器 RAM 是如何通过指令进行数据写入的呢？请读者继续单步运行，分析最后一条指令“mov [r2], a”，观察累加器 a 中的数据是怎么写入寄存器 r2 指向的存储单元中的。

# 实验 6 控制器实验

**实验性质：**验证+设计

**建议学时：**2 学时

## 一、实验目的

- 掌握控制器模块 CU 的工作原理及其在指令执行过程中所起的作用。
- 掌握 DM1000 指令集，学会编写特定功能的汇编源程序。

## 二、预备知识

控制单元是计算机内部的核心部件，可为完成不同指令发出各种操作命令——这些命令（控制信号）控制计算机的所有部件有次序地完成相应的操作，以达到执行程序的目的。控制单元的信号通常有以下几种，见表 6-1。

分类	名称	功能
输入信号	时钟	使控制单元按一定的先后顺序、一定的节奏发出控制信号
	指令寄存器	指令的操作码字段与时钟配合产生不同的控制信号
	标志	根据上条指令的结果产生不同的控制信号
	来自系统总线的控制信号	例如：中断请求、DMA 请求
输出信号	CPU 内的控制信号	CPU 内的寄存器之间的传送和控制 ALU 实现不同操作
	送至系统总线的信号	例如：命令主存或 I/O 读/写、中断响应等

**表 6-1：控制单元的信号**

DM1000 的控制单元是 CU 模块，CU 模块的主要功能是：

1. 取指令，将取出的指令写入指令寄存器 IR 中，同时根据指令获取指令对应微程序入口地址并实现 uPC 跳转执行。
2. 指令执行过程中，uPC 依次加 4，顺序执行微指令。
3. uPC 复位到 0，取下一条指令。

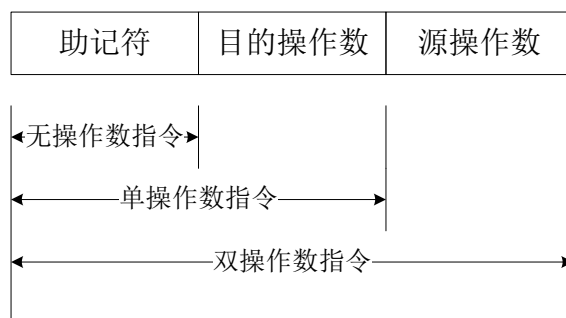
CU 模块中的微程序存储器 ROM 输出的每一条微指令都对应 32 位控制信号，这些控制信号控制 DM1000 完成指定的操作。控制信号的说明请参照附录 A 中的表 A-1。

DM1000 中的指令为单字节或双字节。若指令的操作数不涉及存储器操作，则指令的长度为一个字节，否则为两个字节。

### 2.1 指令结构

DM1000 指令集包括算术运算类指令、逻辑运算类指令、移位类指令、数据传输类指令、跳转类指令、中断返回类指令、输入/输出类指令等类别。DM1000 指令集提供的指令请参考附录 A 中的表 A-4。

根据操作数的个数不同，可以将模型机汇编指令分为 3 类，指令结构图如下：



**图 6-1：DM1000 汇编指令结构示意图**

- ① 无操作数指令。仅有 IN、OUT、RET、IRET 和 NOP 等功能简单的指令。
  - ② 单操作数指令。包含移位指令、取反指令和跳转指令等。
  - ③ 双操作数指令。机器中绝大部分指令均为这种类型，两个操作数中至少有一个是寄存器类型。
- 根据指令的长度可以将指令分为 2 类

- ① 单字节指令：只包含一个字节的机器码。当指令的寻址方式为累加器寻址、寄存器寻址、寄存器间接寻址时，一般为单字节指令
- ② 双字节指令：第一个字节为机器码，第二个字节通常用于保存数据或者数据所在存储单元的地址。当指令的寻址方式为立即数寻址或存储器直接寻址时，一般为双字节指令。

指令的机器码包含一个字节，其格式如图 6-2 所示。

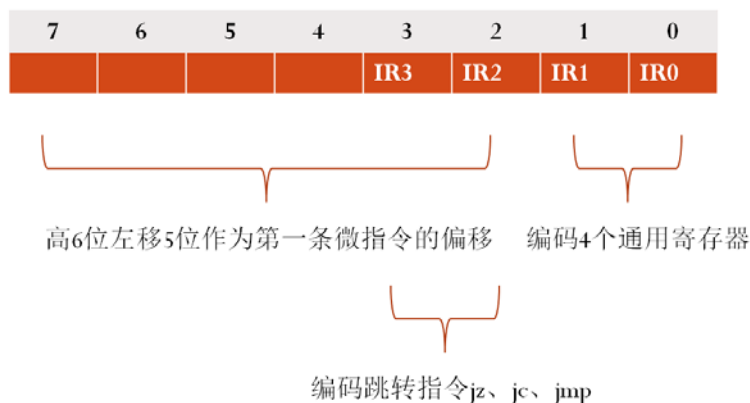


图 6-2：指令机器码格式

## 2.2 指令的寻址方式

DM1000 的寻址方式分为累加器寻址、寄存器寻址、寄存器间接寻址、存储器直接寻址 5 种方式，寻址方式及简要说明如下表所示：

寻址方式	说明
累加器寻址	操作数为累加器 A，例如 NOT A 是将累加器 A 的值取反；OUT 指令隐含寻址累加器 A，是将累加器 A 的值输出到 OUT 寄存器中。
寄存器寻址	参与运算的数据在寄存器 R0~R3 中，例如 ADD A, R0 指令是将 R0 寄存器的值加上累加器 A 的值，然后再写入累加器 A 中。
寄存器间接寻址	参与运算的数据在主存储器中，数据的地址在寄存器 R0~R3 中，例如 MOV A, [R1] 指令是将寄存器 R1 的值作为地址，把存储器中该地址单元的内容送入累加器 A 中。
存储器直接寻址	参与运算的数据在存储器中，数据的地址为指令的操作数。例如 MOV A, Symbol 是将符号 Symbol 指向的存储器中的数据放入累加器 A 中。
立即数寻址	参与运算的数据为指令的操作数。例如 SUB A, 0x10 是将累加器 A 的值减去立即数 0x10，结果存入累加器 A 中。

表 6-2：DM1000 寻址方式

## 三、实验内容

### 3.1 任务（一）

通过对 DM1000 中的控制单元 CU 进行单独仿真测试，观察 CU 模块取指令，执行指令微程序，然后取下一条指令的过程。

请读者按照下面方法之一在本地创建一个项目，用于完成本次任务：

#### 方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建个人项目，然后使用 Dream Logic 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

**方法二：不从 CodeCode.net 平台领取任务**

如果读者不使用CodeCode.net平台，就需要使用Dream Logic提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的URL

为<https://www.codecode.net/engintime/Dream-Logic/Project-Template/DM1000/DM1000.git>。

**建议教师按照下面的步骤在 CodeCode.net 平台上布置此次实验任务：**

- (1) 在计算机组成原理实验课程中，新建一个实验任务。
- (2) 在“新建任务”页面中，教师需要添写“任务名称”，推荐使用“实验6 控制器实验-任务（一）”；还需要填写“模板项目URL”，应该使用  
“<https://www.codecode.net/engintime/Dream-Logic/Project-Template/DM1000/DM1000.git>”。
- (3) 点击“新建任务”按钮，完成新建任务操作。

请读者按照下面的步骤完成本次实验：

1. 打开项目下的原理图文件 CU.dlsche，结合表 6-3，熟悉控制单元 CU 的内部电路原理。

器件或模块名称	说明	功能
IR	指令寄存器	存储指令机器码的第一个字节，为指令执行提供寄存器编码（IR0、IR1）或转移指令类码（IR2、IR3）
uPC	微程序计数器	为微程序存储器 ROM 提供输入地址，从而读出指定微指令，发出控制信号
ROM	微程序存储器	存储指令对应的微程序，执行指令时，只需顺序执行指令对应的微程序即可
uPC_NEXT	uPC 加计数模块	计算下一条微指令地址，由于一条微指令占用 4 个字节，因此，uPC 每次需要加 4 才能得到下一条微指令地址
IR_uAddress_gate	微程序入口地址输出门	将取出指令对应的微程序入口地址输出到 uPC 的输入端，下个时钟上升沿到来时，写入 uPC，从而使 uPC 转去执行指令对应的微程序。 只有取指令时，该三态门输出才打开
IR_Fetch_gate	取指微指令地址输出门	由于取指微指令的地址为 0，所以该输出门输出的是常数 0。 只有复位 uPC 时，该三态输出才打开。
uPC_gate	微指令地址输出门	当顺序执行指令对应的微程序时，打开该三态输出门，将下一条微指令的地址输出到 uPC 输入端。

**表 6-3：CU 模块内部电路**

2. 为了单独仿真 CU 模块，需要在原理图中添加控制信号源和数据输入源。请读者使用 8 位交互式数字信号源作为 DBUS0~DBUS7 的输入，使用一个一位交互式数字信号源控制复位信号 RESET，使用一个单周期时钟源提供时钟 CLK 信号。
3. 启动仿真。

**取指令**

观察取指微指令的执行对 CU 模块内部电路的影响。

1. 将复位信号 RESET 置为高电平，复位无效。左侧源代码窗口中蓝色箭头指向取指微指令。
2. 在电路中可看到 uPC=0，指令对应微程序入口地址输出门 IR\_uAddress\_gate 使能打开，从而将指令对应微程序入口地址输出到 uPC 输入端。
3. 将 8 位数字信号源输出修改为 10，观察此时指令寄存器 IR、微程序计数器 uPC 以及各个三态门

的输入和输出。

4. 输入一次单周期时钟,可看到 IR 写入指令 10,而通过指令译码后 uPC=0x80,指向指令“adda, rx”对应的第一条微指令。

### 执行指令

指令的执行过程实质上是对应微程序指令顺序执行的过程。

1. 左侧的源代码窗口中蓝色箭头指向起始微指令。
2. 在 CU 模块仿真电路中可看到,微指令地址输出 uPC\_gate 使能打开,下一条微指令地址输出到 uPC 输入端。
3. 输入单周期时钟 CLK,可注意到 uPC 依次加 4,左侧蓝色箭头指向表示微指令在顺序执行。
4. 当运行到微指令“resetupc”时,在电路中可看到,取指微指令地址输出 IR\_Fetch\_gate 使能打开,0 地址输出到 uPC 输入端。
5. 输入单周期时钟, uPC=0, 重新执行取指微指令,开始取下一条指令进行执行。

通过上述仿真可知,取指微指令将指令编码进行译码,从而得到指令对应的第一条微指令在 ROM 中的地址,将此地址赋值给 uPC 从而完成跳转,接下来顺序执行指令对应的每条微指令,从而完成指令的功能,最后将 uPC 复位对下一条指令进行取指。这样不停地重复“取指-顺序执行-复位再取指”的过程,从而完成所有指令的执行。

### 提交作业

在提交作业前,读者需要将 CU 模块 SVG 图形文件的链接添加到 README.md 文件中,这样,当使用浏览器查看提交后的线上项目时,就可以从 README.md 文件中看到修改后的 CU 模块了。方法如下:

1. 双击“项目管理器”中的 README.md 文件,使用一种合适的文本编辑器或者 Markdown 编辑器打开此文件。
2. 在 README.md 文件的末尾添加如下的文本:  
# 控制器模块 CU  
![raw svg](CU.dlsche.svg)
3. 保存 README.md 文件。
4. 提交作业。

### 3.2 任务(二)

修改汇编程序文件 ram.asm,编写一段简单的汇编程序,通过跳转指令实现循环累加运算  $1+2+3+4+5$ ,并使用 OUT 指令将计算的结果输出到外部设备的 ROUT 寄存器。

仿真运行编写的汇编程序,确保其可以正常运行。说明在 OUT 指令执行的过程中,控制器产生了哪些控制信号完成了将累加器 A 中的数据发送到外部设备的 ROUT 寄存器中。

请读者按照下面方法之一在本地创建一个项目,用于完成本次任务:

#### 方法一:从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务,从而在 CodeCode.net 平台上创建个人项目,然后使用 Dream Logic 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

#### 方法二:不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台,就需要使用 Dream Logic 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中,实验模板的 URL

为<https://www.codecode.net/engintime/Dream-Logic/Project-Template/DM1000/DM1000.git>。

**建议教师按照下面的步骤在 CodeCode.net 平台上布置此次实验任务:**



- (1) 在计算机组成原理实验课程中，新建一个实验任务。
- (2) 在“新建任务”页面中，教师需要添写“任务名称”，推荐使用“实验6 控制器实验-任务（二）”；还需要填写“模板项目URL”，应该使用  
“<https://www.codecode.net/engintime/Dream-Logic/Project-Template/DM1000/DM1000.git>”。
- (3) 点击“新建任务”按钮，完成新建任务操作。

请读者按照下面的提示完成本次实验：

1. 使用 ADD 指令完成加 1 操作，用于产生每个数据项。
2. 将循环次数存储在 R3 寄存器中，每完成一次累加操作，使用 SUB 指令对 R3 寄存器减 1。然后使用 JZ 指令判断循环计数变为 0 时，结束循环。

## 四、 思考与练习

1. 在设计一个处理器时，应该尽量简化设计，使用最少的逻辑门来实现功能，这样不但能够让处理器的架构简洁，还能减少芯片面积，降低功耗和散热，甚至还可以提高运行频率。CU 控制器模块中的 uPC\_NEXT 模块用于得到 uPC 加 4 后的值，其中包含了 4 个 74LS181 芯片，这会大大增加逻辑门的数量。请读者简化 CU 控制器模块的设计，删除 uPC\_NEXT 和 uPC\_gate 器件，将 uPC 替换为一个 16 位同步置数/异步清零二进制加法计数器（功能表参见实验四的预备知识），然后让 uPC 输出的 Q0 与 ROM 的 A2 连接，依此类推。这样当 uPC 加 1 时，ROM 的地址就会增加 4（数字系统设计中经常会用到此技巧）。需要注意的是，由于 uPC 的值左移 2 位后连接了 ROM 的地址，所以 PC 的值在译码时就需要少移动 2 位了。此外，读者还要设计如何控制 uPC 的 EN 和 LOAD 管脚，确保其在适当的时候进行加 1 操作。如果 CU 控制器修改完毕后，在仿真时发现源代码窗口中的 uPC 对应的蓝色箭头无法正确指向微指令也没有关系，毕竟此时 uPC 的值已经与 ROM 的地址不再对应了，此时读者只要判断汇编指令能够正确执行即可。
2. 如果读者顺利完成了思考与练习的第 1 题，请读者再思考一下，如果继续删除 CU 控制器中的 IR\_uAddress\_gate 和 IR\_Fetch\_gate 器件，然后让 UPC\_RESET\_EN 信号和 RESET 信号配合控制 uPC 的 CLR 信号，这样是否可行？在 uPC 复位时的电路是否仍然符合“同步时序电路”的规则呢？如果这样存在问题的话，请读者设计一种计数器，能够满足这里的要求。
3. 当前控制器中是直接使用 6 个控制信号（S0、S1、S2、S3、M、C）来控制运算器的运算方式的（共有 48 种）。但是指令集中并没有用到所有的运算方式，所以可以通过设计一个译码电路，使用更少的控制信号，来产生运算器需要的 6 个输入信号，从而可以节省控制信号，满足系统今后扩展的需要。请读者首先根据指令集中运算方式的数量，计算出最少需要使用几个控制信号，然后设计一个真值表，将 CU 模块的控制信号和译码后产生的 ALU 模块的 6 个输入信号填入真值表中，最后根据真值表设计一个译码电路，还可以将此译码电路绘制到 DM1000 的原理图中，完成仿真验证。



# 实验 7 数据通路实验

实验性质：验证+设计

建议学时：2 学时

## 一、实验目的

- 掌握数据通路的概念。
- 掌握微指令和数据通路之间的关系。
- 掌握数据通路的分析方法，并可以由数据通路得到微指令的机器码，为设计微指令打下基础。

## 二、预备知识

### 2.1 数据通路

计算机系统中，各个部件通过数据总线连接形成的数据传送路径称为数据通路。数据从什么地方开始，中间经过哪些部件，最后传送到什么位置，都是分析数据通路时需要关注的问题。

数据通路是保证计算机内数据正确流动的硬件基础。在计算机中，从抽象的程序执行到具体的微指令的执行，实际是从软件到硬件的转换过程。

每一条数据通路对应一条微指令，微指令实质上是一组高低电平编码的控制信号，当这些控制信号作用在模型机上时，就可以规定一条数据通路。

### 2.2 微指令集

DM1000 微指令集其实就是机器语言指令集。模型机执行指令时，将该指令取出后，送至指令寄存器 IR。同时将指令的高 6 位送至微程序计数器 uPC。指令寄存器的最低两位 IR<sub>1</sub>IR<sub>0</sub>作为通用寄存器的选择信号；高 6 位在程序执行时仅作为 uPC 的高 6 位使用。在跳转类指令 JMP、JC、JZ、CALL 中，指令的 3、4 位 IR<sub>2</sub>、IR<sub>3</sub>结合控制信号 ELP 起到编码跳转方式的作用。请查看“程序计数器实验”相关内容了解详情。

uPC 为微程序计数器，uPC 指向微程序存储器中的微指令。微程序存储器始终输出 uPC 指定地址单元的微指令。将 DM1000 中的所有指令分步，每一步对应一组 32 位编码的控制信号序列（微指令），那么一条指令对应数条连续的微指令（微程序），将每条指令对应的微程序存储到微程序存储器特定位置（指令的高 6 位与低三位 0 组合得到 9 位地址码，可寻址 0x10~0x1ff），由于不同的指令，其高 6 位编码不同，所以就将各个指令的微程序存储到不同的位置，在执行每一种指令时，就通过其高 6 位找到对应的微程序，通过 uPC+4 计数，依次取出并执行每一条微指令，微程序执行完，那么该指令也就执行完成了。注意：由于微程序存储器的最低两位地址始终是 0，所以，相连编码的两种指令，其对应的微程序地址至少相差 4 的倍数，这样，一条指令对应的微指令条数就是 8 的倍数，DM1000 中绝大多数指令对应 8 条微指令。结合项目下的微指令源程序文件 rom.masm 理解微指令，注意，不是每一个指令对应于一段微程序，而是每一种类型的指令对应于一段微程序。

微程序存储器中的微程序是如何得到的呢？由上可知，指令的编码是事先编好的，我们对每种指令执行过程进行分步，对每一步进行数据通路分析，从而得到其对应的 32 位控制信号编码，即微指令，这样就可以得到有序的微指令集。

### 2.3 同步时序电路

DM1000 模型机是按同步时序电路的原则设计的，模型机在运行程序的时候，在时钟的驱动下，总是由当前稳定状态迁入下一个稳定状态。

数字电路包含组合逻辑电路和时序逻辑电路。组合逻辑没有环路和竞争。如果将输入作用到组合逻辑电路中，输出总是在传播延迟内稳定为一个确定的值。但是，包含环路的时序电路存在不良的竞争或不稳定行为。为了避免时序电路中的竞争或不稳定，往往需要在环路中插入寄存器来断开电路，从而将电路转变成组合逻辑电路和寄存器的集合。寄存器包含系统的状态，这些状态仅仅在时钟上升沿到达时发生改变，

所以说状态同步于时钟信号。如果时钟足够慢，使得在下一个时钟上升沿到达之前输入到寄存器的信号都可以稳定下来，那么所有的竞争都将被消除。根据反馈环路上总是使用寄存器的原则，一个电路如果是同步时序电路，那么它由相互连接的电路元件构成，且满足以下条件：

- 每一个电路元件是寄存器或者组合电路。
- 至少有一个电路元件是寄存器。
- 所有寄存器都接收同一个时钟信号。
- 每个环路至少包含一个寄存器。

## 三、 实验内容

### 3.1 任务（一）

通过使用 DM1000 仿真执行若干条指令，详细分析指令包含的微指令在执行过程中产生的数据通路。

请读者按照下面方法之一在本地创建一个项目，用于完成本次任务：

#### 方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建个人项目，然后使用 Dream Logic 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

#### 方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，就需要使用 Dream Logic 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的 URL

为<https://www.codecode.net/engintime/Dream-Logic/Project-Template/DM1000/DM1000.git>。

**建议教师按照下面的步骤在 CodeCode.net 平台上布置此次实验任务：**

- (1) 在计算机组成原理实验课程中，新建一个实验任务。
- (2) 在“新建任务”页面中，教师需要添写“任务名称”，推荐使用“实验 7 数据通路实验-任务（一）”；还需要填写“模板项目 URL”，应该使用  
“<https://www.codecode.net/engintime/Dream-Logic/Project-Template/DM1000/DM1000.git>”。
- (3) 点击“新建任务”按钮，完成新建任务操作。

请读者按照下面的步骤完成本次实验：

1. 打开项目下的汇编源程序文件 ram.asm，输入下列程序：

```
.text
mov a, 3
mov r3, 4
add a, r3
```

2. 编译源程序。
3. 设置为单周期时钟仿真，然后启动仿真，并复位。

此时 DM1000 会在第一条微指令（取指微指令）产生的控制信号状态下，并等待时钟信号从而进入下一个状态。

#### 分析取指微指令的数据通路

取指令操作是所有指令必备的操作，且操作过程完全一致。在 DM1000 中，是如何完成取指令操作的呢？相关控制信号说明如下：

1. 打开程序计数器 PC 模块，可看到 PC\_A\_GATE\_EN 为低电平，程序计数器 PC 向地址总线输出门 PC\_gate 使能打开，PC 值作为存储器的地址输出到地址总线 ABUS 上。

2. 打开存储器 MEM 模块，可看到 MEM\_GATE\_EN 为低电平，存储器输出门 MEM\_gate 使能打开，从存储器读出一个字节（即指令的机器码 7C）输出到数据总线 DBUS 上。
3. 打开控制单元 CU，可看到 IREN 为低电平，指令寄存器 IR 写使能，时钟上升沿到来时即可将指令的机器码写入 IR 寄存器。由于 IREN 信号为低电平，总线输出门 IR\_uAddress\_gate 打开，同时微指令计数器 uPC 使能信号为低电平，时钟上升沿到来时，经过译码后的指令机器码就会写入 uPC，作为 ROM 的地址。

根据上述分析可以绘制出如图 7-1 所示的通路示意图。其中方框表示寄存器，圆形表示组合逻辑电路。由于此时是读存储器，不需要时钟信号，所以这里把存储器也作为组合逻辑电路来理解了。请读者根据预备知识中描述的同步时序电路的概念，判断图 7-1 所示的通路是否是一条同步时序电路。

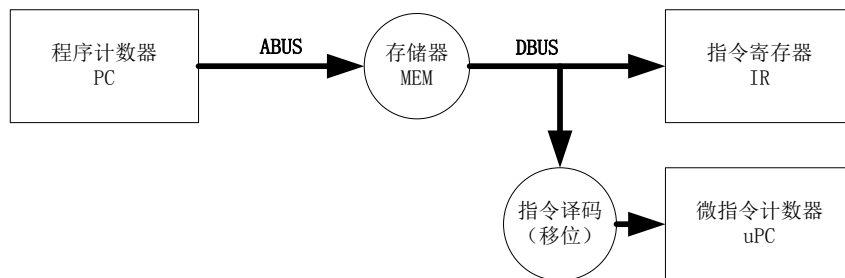


图 7-1: path [pc], ir 微指令的数据通路

为了完成取指操作，应使得 PC\_A\_GATE\_EN=0、MEM\_GATE\_EN=0、IREN=0、X1=0、X2=0。这就要求 ROM 输出的控制信号应该如下表所示：

位置	名称	值	位置	名称	值
31	保留	1	15	CSP_LOAD	1
30	保留	1	14	REG_WR	1
29	UPC_RESET_EN	1	13	CN	1
28	PC_ADD	0	12	FLAG_REG_EN	1
27	CSP_U\D	1	11	X3	1
26	IA_REG_EN	1	10	X2	0
25	MEM_WR	1	9	X1	0
24	ROUT_REG_EN	1	8	X0	1
23	PC_A_GATE_EN	0	7	W_REG_EN	1
22	IREN	0	6	A_REG_EN	1
21	保留	1	5	M	1
20	PC_LOAD_EN	1	4	C	1
19	MAR_REG_EN	1	3	S3	1
18	MAR_GATE_EN	1	2	S2	1
17	ASR_REG_EN	1	1	S1	1
16	SP_REG_EN	1	0	S0	1

表 7-1: 取指控制信号

上表中，“1”表示高电平，“0”表示低电平。根据表 7-1 可以得到取指微指令编码的二进制形式为：1110 1111 0011 1111 1111 1001 1111 1111。按照十六进制编码形式，可知其编码为 EF3FF9FF，这就是取指微指令在 ROM 中的机器码。

请读者按照下面的步骤验证微指令的机器码：

1. 打开控制单元 CU，将 ROM 输出信号与表 7-1 进行比较验证。
2. 将表 7-1 中的取指微指令的十六进制编码与源代码窗口 2 中的取指微指令“path [pc], ir”的

32 位机器码进行比较验证。

### 分析 ADD A, R3 指令的数据通路

加法指令 ADD A, R3 的功能可以描述为：取出 R3 寄存器的内容并送至工作寄存器 W 中，设定 ALU 做加法运算，ALU 将累加器 A 的值和工作寄存器 W 的值相加，结果回写到累加器 A 中。读者可以按照下面的步骤分析其数据通路。

1. 单周期运行程序直到 add a, r3 指令的第一条微指令 path rx, w 处。
2. 分析此时 DM1000 的状态：在寄存器模块 REG 中，通用寄存器的选择是由指令机器码的低两位决定的，要选择寄存器 R3，指令机器码的 IR1 和 IR0 的值都为 1；要读出通用寄存器的值，读信号 REG\_READ 需要为低电平，这样寄存器 R3 的值就可以放在数据总线上；ALU 单元的输入信号 W\_REG\_EN 为低电平，即工作寄存器 W 写使能，下个时钟上升沿到来时，数据总线上的值就会写入 W 寄存器。数据通路如图 7-2 所示。控制器模块 CU 中的 ROM 输出的控制信号如表 7-2 所示。

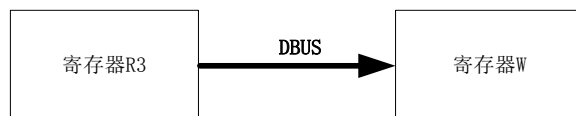


图 7-2: path rx, w 微指令的数据通路

位置	名称	值	位置	名称	值
31	保留	1	15	CSP_LOAD	1
30	保留	1	14	REG_WR	1
29	UPC_RESET_EN	1	13	CN	1
28	PC_ADD	0	12	FLAG_REG_EN	1
27	CSP_U\D	1	11	X3	1
26	IA_REG_EN	1	10	X2	0
25	MEM_WR	1	9	X1	1
24	ROUT_REG_EN	1	8	X0	0
23	PC_A_GATE_EN	1	7	W_REG_EN	0
22	IREN	1	6	A_REG_EN	1
21	保留	1	5	M	1
20	PC_LOAD_EN	1	4	C	1
19	MAR_REG_EN	1	3	S3	1
18	MAR_GATE_EN	1	2	S2	1
17	ASR_REG_EN	1	1	S1	1
16	SP_REG_EN	1	0	S0	1

表 7-2: 取数控制信号

根据表 7-2 可知，将 R3 的值送到 W 寄存器的微指令编码的二进制形式为：1110 1111 1111 1111 1111 1010 0111 1111。按照十六进制编码的形式，编码为 EFFFFA7F。

请读者按照下面的步骤验证微指令的机器码：

1. 打开控制单元 CU，将 ROM 输出信号与表 7-2 进行比较验证。
2. 将表 7-2 所得的微指令编码与源代码窗口 2 中的微指令“pathrx, w”的 32 位机器码进行比较验证。

接下来分析将求和的结果写到累加器 A 的数据通路：

1. 输入一次单周期时钟，完成将 r3 的数据写入 w 寄存器的微指令。

2. 此时准备执行微指令 path alu\_add, a。分析此时 DM1000 的状态：A 和 W 的值需要做无进位的加法运算，那么相关控制信号  $M=0$  (ALU 做算术运算)， $C=1$  (无进位)， $S_3 S_2 S_1 S_0 = 1001$  (加法)，FLAG\_REG\_EN 为低电平，标志寄存器使能，准备保存运算结果的各个标志； $X_3 X_2 X_1 X_0 = 0100$ ，就会打开 ALU 直通输出 D\_gate，将运算结果输出到数据总线上；A\_REG\_EN 为低电平，累加器 A 写使能，准备将数据总线上的计算结果写入累加器 A。数据通路如图 7-3 所示，注意，在环路上的累加器 A 确保这个通路符合同步时序电路的要求。控制器模块 CU 中的 ROM 输出的控制信号如表 7-3 所示。

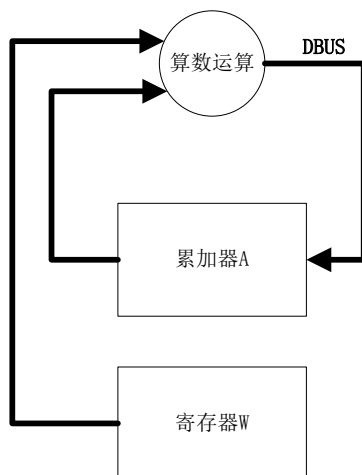


图 7-3: path alu\_add, a 微指令的数据通路

位置	名称	值	位置	名称	值
31	保留	1	15	CSP_LOAD	1
30	保留	1	14	REG_WR	1
29	UPC_RESET_EN	1	13	CN	1
28	PC_ADD	0	12	FLAG_REG_EN	0
27	CSP_U\D	1	11	X3	0
26	IA_REG_EN	1	10	X2	1
25	MEM_WR	1	9	X1	0
24	ROUT_REG_EN	1	8	X0	0
23	PC_A_GATE_EN	1	7	W_REG_EN	1
22	IREN	1	6	A_REG_EN	0
21	保留	1	5	M	0
20	PC_LOAD_EN	1	4	C	1
19	MAR_REG_EN	1	3	S3	1
18	MAR_GATE_EN	1	2	S2	0
17	ASR_REG_EN	1	1	S1	0
16	SP_REG_EN	1	0	S0	1

表 7-3: 加法运算控制信号

根据表 7-3 可得将 A 与 W 相加的结果写回累加器 A 的微指令编码的二进制形式为：1110 1111 1111 1111 1110 0100 1001 1001。按照十六进制编码的形式，编码为：EFFF E499。

请读者按照下面的步骤验证微指令的机器码：

1. 打开控制单元 CU，将 ROM 输出信号与表 7-3 进行比较验证。
2. 将表 7-3 所得微指令编码与源代码窗口 2 中的微指令“path alu\_add, a”进行比较验证。
3. 输入单步时钟，运算结果写入累加器 A。

#### 4. 结束仿真。

请读者参考图 7-3，将“path alu\_add, a”微指令的数据通路在 ALU 模块的原理图中使用红色标识出来，也就是将所有参与构造此数据通路的网络连线选中后，在右侧的“属性”窗口将绘制颜色修改为红色。

### 提交作业

在提交作业前，读者需要将 ALU 模块 SVG 图形文件的链接添加到 README.md 文件中，这样，当使用浏览器查看提交后的线上项目时，就可以从 README.md 文件中看到修改后的 ALU 模块了。方法如下：

1. 双击“项目管理器”中的 README.md 文件，使用一种合适的文本编辑器或者 Markdown 编辑器打开此文件。
2. 在 README.md 文件的末尾添加如下的文本：  
# 运算器模块 ALU  
![raw svg](ALU.dlsche.svg)
3. 保存 README.md 文件。
4. 提交作业。

### 3.2 任务（二）

通过使用 DM1000 仿真执行若干条指令，详细分析指令包含的微指令在执行过程中产生的数据通路。

请读者按照下面方法之一在本地创建一个项目，用于完成本次任务：

#### 方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建个人项目，然后使用 Dream Logic 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

#### 方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，就需要使用 Dream Logic 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的 URL

为 <https://www.codecode.net/engintime/Dream-Logic/Project-Template/DM1000/DM1000.git>。

**建议教师按照下面的步骤在 CodeCode.net 平台上布置此次实验任务：**

- (1) 在计算机组成原理实验课程中，新建一个实验任务。
- (2) 在“新建任务”页面中，教师需要添写“任务名称”，推荐使用“实验 7 数据通路实验-任务（二）”；还需要填写“模板项目 URL”，应该使用  
[“https://www.codecode.net/engintime/Dream-Logic/Project-Template/DM1000/DM1000.git”](https://www.codecode.net/engintime/Dream-Logic/Project-Template/DM1000/DM1000.git)。
- (3) 点击“新建任务”按钮，完成新建任务操作。

请读者打开项目下的汇编源程序文件 ram.asm，输入下列程序。这个汇编程序是将存储器中的数据放入累加器 A 中，num 标签用来在数据段中定义一个变量，也就是在内存中的指定位置放入了一个字节的数据，num 代表这个数据在内存中的地址。

```
.text
mov a, num
.data
num: 0x88
```

编译通过后启动仿真。在仿真过程中，请读者重点查看微指令 path [pc], mar 和 path [mar], a 的数据通路，并仿照图 7-1 和表 7-1 的形式绘制这两条微指令的数据通路图和控制信号表格，重点理解地址寄存器 mar 的作用。设想一下如果没有 mar 是否可以完成类似的工作呢？

请读者将“path [mar], a”微指令的数据通路在原理图中使用红色标识出来，也就是将所有参与构

造此数据通路的网络连线选中后，在右侧的“属性”窗口将绘制颜色修改为红色。

## 提交作业

在提交作业前，读者需要将 ALU 模块和 MEM 模块的 SVG 图形文件的链接添加到 README.md 文件中，这样，当使用浏览器查看提交后的线上项目时，就可以从 README.md 文件中看到修改后的 ALU 模块和 MEM 模块了。方法如下：

1. 双击“项目管理器”中的 README.md 文件，使用一种合适的文本编辑器或者 Markdown 编辑器打开此文件。
2. 在 README.md 文件的末尾添加如下的文本：
 

```
# 运算器模块 ALU
![raw svg](ALU.dlsche.svg)
# 存储器模块 MEM
![raw svg](MEM.dlsche.svg)
```
3. 保存 README.md 文件。
4. 提交作业。

## 3.3 任务（三）

通过使用 DM1000 仿真执行若干条指令，详细分析指令包含的微指令在执行过程中产生的数据通路。

请读者按照下面方法之一在本地创建一个项目，用于完成本次任务：

### 方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建个人项目，然后使用 Dream Logic 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

### 方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，就需要使用 Dream Logic 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的 URL

为 <https://www.codecode.net/engintime/Dream-Logic/Project-Template/DM1000/DM1000.git>。

**建议教师按照下面的步骤在 CodeCode.net 平台上布置此次实验任务：**

- (1) 在计算机组成原理实验课程中，新建一个实验任务。
- (2) 在“新建任务”页面中，教师需要添写“任务名称”，推荐使用“实验 7 数据通路实验-任务（三）”；还需要填写“模板项目 URL”，应该使用  
“<https://www.codecode.net/engintime/Dream-Logic/Project-Template/DM1000/DM1000.git>”。
- (3) 点击“新建任务”按钮，完成新建任务操作。

请读者打开项目下的汇编源程序文件 ram.asm，输入下列程序。这个汇编程序完成了函数调用功能，即调用一个函数 fun 完成加法运算。参与加法运算的两个加数是通过寄存器 R0 和 R1 传入函数 fun 的，在函数中完成了加法运算，然后将计算结果通过寄存器 R0 返回给调用者，最后调用者使用 out 指令将计算结果输出到外部寄存器。fun 标签用来定义函数的入口点，call 指令完成函数调用，在函数中使用 ret 指令返回。

```
.text
```

```
mov sp, 0xf8
```

```
mov r0, 1
```

```
mov r1, 2
```

```
call fun
```

```

mov a, r0
out

fun:
mov a, r0
add a, r1
mov r0, a
ret

```

在读者开始仿真这段汇编程序之前，需要对 call 指令、ret 指令和栈的使用有一个初步的了解。

## 栈

栈在处理器运行过程中有十分重要的作用。DM1000 使用 sp 寄存器指向栈顶，所以在汇编程序的开始就将 sp 的值进行了初始化，使之指向存储器中的一个可用的位置。当有数据入栈时，先将 sp 的值减 1，使 sp 寄存器指向存储器中的新栈顶，然后将从某个寄存器得来的一个字节的数据写入 sp 指向的栈顶位置；当出栈时，先将 sp 指向的栈顶位置的数据放入一个寄存器中，然后将 sp 的值加 1。在函数调用、中断调用等场合都会用到栈。以函数调用为例，call 指令执行时会将 PC 寄存器中的函数返回地址入栈，ret 指令会将函数返回地址出栈到 PC 寄存器中，从而让程序从 call 指令的下一条指令继续运行。

## Call 指令

Call 指令主要完成两项工作：一项工作是将 PC 寄存器中的函数返回地址入栈，另外一项工作是将函数入口点的地址传递给 PC 寄存器，从而跳转到函数入口点继续运行。

## Ret 指令

Ret 指令主要是将函数返回地址出栈，并放入 PC 寄存器，从而让程序从 call 指令的下一条指令继续运行。

编译通过后启动仿真。在仿真过程中，请读者重点查看 call 指令的微指令和 ret 指令的微指令的执行过程，掌握这两条指令是如何配合起来完成函数调用功能的。另外，还要重点掌握 DM1000 是如何使用 sp 寄存器实现栈功能的，以及 sp 寄存器是如何通过 csp 计数器实现加 1 和减 1 操作的。

完成仿真后，尝试回答下面的问题：

1. 函数 fun 的入口地址是多少？Call 指令是如何知道 fun 函数的入口地址的？
2. Call 指令执行时是如何得到函数的返回地址的？函数的返回地址是多少？函数的返回地址入栈操作本质上就是将函数的返回地址放在了存取器中，其在存储器中的地址是多少？

为了加深理解，请读者仿照图 7-1 和表 7-1 的形式绘制微指令 path sp\_dec, csp 的数据通路图和控制信号表格。然后在原理图中将 path sp\_dec, csp 微指令的数据通路使用红色标识出来，也就是将所有参与构造此数据通路的网络连线选中后，在右侧的“属性”窗口将绘制颜色修改为红色。

## 提交作业

在提交作业前，读者需要将 REG 模块的 SVG 图形文件的链接添加到 README.md 文件中，这样，当使用浏览器查看提交后的线上项目时，就可以从 README.md 文件中看到修改后的 REG 模块了。方法如下：

1. 双击“项目管理器”中的 README.md 文件，使用一种合适的文本编辑器或者 Markdown 编辑器打开此文件。
2. 在 README.md 文件的末尾添加如下的文本：
 

```
# 寄存器模块 REG
!raw svg (REG.dlsche.svg)
```
3. 保存 README.md 文件。
4. 提交作业。



# 实验 8 中断实验

实验性质：验证+设计

建议学时：4 学时

## 一、实验目的

- 熟悉中断请求、中断响应、中断处理及中断返回的过程。
- 掌握软中断和硬中断实现原理。

## 二、预备知识

### 2.1 中断

在计算机执行程序的过程中，当出现异常情况或特殊请求时，计算机停止现行程序的运行，转向对这些异常情况或特殊请求的处理，处理结束后再返回到现行程序的中断处，继续执行原程序，这就是“中断”。中断是现代计算机能有效合理地发挥效能和提高效率的一个十分重要的功能。通常把实现这种功能所需的软硬件技术统称为中断技术。

中断的类型可以参见表 8-1。

分类	名称	定义
硬中断	可屏蔽中断	可通过在中断屏蔽寄存器中设定位掩码来关闭
	非可屏蔽中断	无法通过在中断屏蔽寄存器中设定位掩码来关闭
	处理器间中断	一种特殊的硬件中断，由处理器发出，被其他处理器接收
	伪中断	一类不希望被产生的中断
软中断	软中断	由一条中断指令触发，用以自陷进入一个中断服务程序

表 8-1：中断类型

### 2.2 软中断

在 DM1000 中，软中断是指一个程序通过执行一条 INT 指令去主动调用中断服务程序，中断服务程序执行完毕后，中断返回，CPU 从中断位置继续执行原来的程序。

### 2.3 硬中断

在 DM1000 中，硬中断是指当 CPU 检测到外部设备发出的中断请求时，CPU 打断正在执行的程序，转去执行对应的中断服务程序，中断服务程序执行完毕后，CPU 返回到中断位置继续执行原来的程序。

硬中断的执行过程可以分为四个步骤：

#### (1) 保护现场

为了从中断服务程序返回后，被中断的程序可以继续运行，有必要将中断时的现场保存起来，这通常是将处理器内部的一些寄存器入栈来实现的，例如标志寄存器等。此外还需要将程序中断处的下一条指令的地址入栈。DM1000 的初始版本还没有实现保护现场功能。

#### (2) 执行对应的中断服务程序

不同的中断请求源对应的中断服务程序是不同的，这样才能为不同的外部设备提供不同的中断服务。将所有中断服务程序的入口地址（起始地址）保存在一个连续的主存储区域中，就得到了一个可供查询的中断向量表。当发生外部中断时，CPU 根据中断号在中断向量表中查询到对应的中断向量，然后在从中断向量中得到中断服务程序的入口地址，就可以调用中断服务程序了。

在 DM1000 中，由于硬件电路设计的原因，中断向量表位于存储器中的 0xf8~0xff 区域，一个中断向量占用一个字节，其中存放了中断服务程序的入口地址，所以 DM1000 的中断向量表中只能存放八个中断向量。

中断向量表中的内容必须由汇编程序的开发者填入，所以读者在编写中断服务程序后，必须通过北京英真时代科技有限公司 <http://www.engintime.com>

编程的方式将中断向量写入中断向量表中对应的位置。下图展示的是含有两个中断服务程序时主存储器的布局：

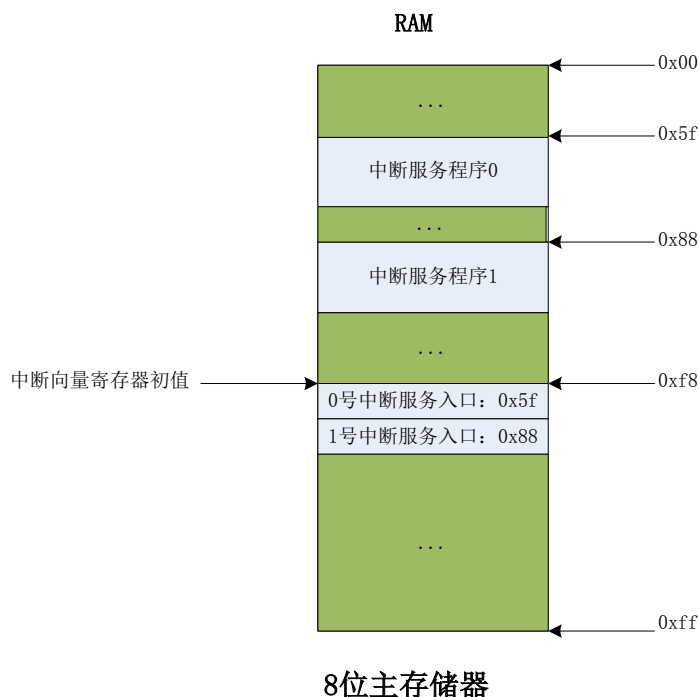


图 8-2：DM1000 中断服务程序及中断向量存储位置示意图

### (3) 恢复现场

在中断服务程序结束时，需要将之前被打断程序的“现场”恢复到原来的寄存器中，例如将标志寄存器的值出栈。由于在 DM1000 的初始版本中还没有实现中断调用保护现场功能，所以在中断服务程序结束时，也就无需恢复现场了。

### (4) 中断返回

中断服务程序的最后一条指令通常是一条中断返回指令 `iret`，该指令会将之前保存在栈中的中断返回地址出栈到 PC 寄存器中，从而回到被打断的程序处继续运行。

## 三、实验内容

### 3.1 任务（一）软中断

仿真运行含有软中断指令 `INT` 的汇编程序，观察软中断的实现过程。

请读者按照下面方法之一在本地创建一个项目，用于完成本次任务：

#### 方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建个人项目，然后使用 Dream Logic 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

#### 方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，就需要使用 Dream Logic 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的 URL

为 <https://www.codecode.net/engine/Dream-Logic/Project-Template/DM1000/DM1000.git>。

**建议教师按照下面的步骤在 CodeCode.net 平台上布置此次实验任务：**

- (1) 在计算机组成原理实验课程中，新建一个实验任务。
- (2) 在“新建任务”页面中，教师需要添写“任务名称”，推荐使用“实验 8 中断实验-任务（一）”；还需要填写“模板项目 URL”，应该使用

“<https://www.codecode.net/engintime/Dream-Logic/Project-Template/DM1000/DM1000.git>”。

(3) 点击“新建任务”按钮，完成新建任务操作。

请读者按照下面的步骤完成本次实验：

1. 打开项目下的源程序文件 ram.asm，将下面的汇编程序写入 ram.asm 文件中。此汇编程序安装了 0 号软中断的中断服务程序，并通过 INT 指令完成了对 0 号软中断的调用。

```
.text
```

```
mov sp, 0xf8 ; 初始化栈。栈底紧邻中断向量表
```

```
lea a, INT0 ; 将 0 号中断的入口地址放入寄存器 A
```

```
mov r0, 0xf8 ; 将中断向量表中 0 号中断的位置放入 R0 寄存器
```

```
mov [r0], a ; 在中断向量表中安装 0 号中断的中断向量
```

```
int 0 ; 调用 0 号中断
```

```
ENDLOOP:
```

```
jmp ENDLOOP
```

```
INT0:
```

```
mov a, 0xff ; 0 号中断服务程序，只是简单修改寄存器 A 的值
```

```
iret
```

2. 保存并编译 ram.asm 源程序。
3. 启动仿真，使用单周期时钟运行程序。观察每一条指令的执行过程，重点理解中断调用和中断返回的实现过程。

完成仿真后，尝试回答下面的问题：

1. 0 号中断服务程序的入口地址是多少？中断向量表中存放的 0 号中断向量的值是多少？
2. INT 指令是如何得到中断号 0 的？又是如何根据中断号 0 得到其中断向量在中断向量表中的位置的？
3. INT 指令执行时是如何得到中断返回地址的？此次中断的返回地址是多少？中断返回地址入栈操作本质上就是将中断返回地址放在了存取器中，其在存储器中的地址是多少？

请读者在此汇编程序的基础上进行修改，再安装一个 1 号软中断的中断服务程序。在 1 号软中断的中断服务程序中，将寄存器 R0 和 R1 的值相加，结果通过 R0 寄存器返回。然后，通过调用 1 号软中断计算 8+16 的结果。

### 3.2 任务（二）硬中断

请读者按照下面方法之一在本地创建一个项目，用于完成本次任务：

#### 方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建个人项目，然后使用 Dream Logic 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

#### 方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，就需要使用 Dream Logic 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的 URL

为<https://www.codecode.net/engintime/Dream-Logic/Project-Template/DM1000/Hardware-IRQ.git>。

**建议教师按照下面的步骤在 CodeCode.net 平台上布置此次实验任务：**

- (1) 在计算机组成原理实验课程中，新建一个实验任务。
- (2) 在“新建任务”页面中，教师需要添写“任务名称”，推荐使用“实验8中断实验-任务（二）”；还需要填写“模板项目URL”，应该使用  
[“https://www.codecode.net/engintime/Dream-Logic/Project-Template/DM1000/Hardware-IRQ.g.it”](https://www.codecode.net/engintime/Dream-Logic/Project-Template/DM1000/Hardware-IRQ.g.it)。
- (3) 点击“新建任务”按钮，完成新建任务操作。

请读者按照下面的步骤完成本次实验：

Hardware-IRQ 是一个基于 DM1000，能响应并处理硬中断的 8 位模型机。与 DM1000 相比，主要变化体现在电路以及微程序上。请读者打开项目下的原理图文件 Hardware-IRQ.dlsche，结合电路，理解下列所述变化。

#### 电路的变化：

- 1) 增加了一个硬中断控制器 IRQ，用来判别不同优先级的中断请求，并向 CPU 发出中断请求信号，申请中断服务。该中断控制器支持 8 级中断请求，支持中断嵌套。IRQ 由 3 个功能模块构成：
  - 中断请求模块 IRR。记录 8 个中断源发出的中断请求信号，上升沿触发中断请求。当某个中断请求被响应后，清除对应的位，避免重复响应。
  - 中断优先级判别器 PR。将新的中断请求与当前正在服务的中断请求进行优先级比较，若新的中断请求优先级低于当前正在服务的中断请求，则需要等待当前正在执行的中断服务程序返回后才能获取中断服务。若新的中断请求优先级高于当前正在服务的中断请求，则 CPU 转去执行高优先级的中断服务程序，执行完后，返回继续执行还未完成的低优先级中断服务程序。
  - 中断服务寄存器 ISR。记录当前正在服务的所有中断请求。若有多个标志处于有效状态，则表示当前存在中断嵌套。
- 2) 增加了一个硬中断使能控制信号 IF，用来控制 CPU 是否屏蔽硬中断请求。当 IF=1 时，CPU 允许响应硬中断请求；当 IF=0 时，CPU 屏蔽硬中断请求。

#### 微程序 rom.asm 的变化：

- 1) 在每条指令的微指令最后位置添加了一条微指令 ask\_for\_int，用于查询是否有硬中断请求。
- 2) 硬中断处理微程序中增加微指令 inta1 用来记录中断号，inta2 用来读取中断号并将其写入中断向量寄存器 IA 中，IA 指向的存储单元中存放了中断服务程序的入口地址。
- 3) 中断返回指令 iret 中增加了微指令 eoi，该微指令用来清除中断服务寄存器 ISR 中对应的标志位，表示对应中断服务程序已经执行完毕，将其移除正在服务的中断请求队列。

下面，我们通过实验，完成硬中断请求、判优、响应、服务和返回的整个过程。实验步骤如下：

1. 熟悉中断控制器 IRQ 的内部电路。
2. 在 Hardware-IRQ.dlsche 中找到用于控制中断是否允许的信号源 IF。
3. 打开汇编源程序文件 ram.asm，熟悉代码，理解中断的安装。
4. 打开项目下的微程序源文件 rom.asm，在每种指令后有一条微指令 ask\_for\_int，该微指令用来查询中断请求。

在对电路有了基本的了解后，接下来通过仿真，模拟单个中断响应过程。模型机启动仿真后，首先需要安装所有的中断服务程序。

1. 打开原理图 Hardware-IRQ.dlsche，启动仿真。
2. 通过按键 F，使 IF=0，禁止中断。因为在执行中断服务程序安装的过程中，还无法响应中断，所以必须先禁止中断。

3. 通过按键 R, 进行复位。
4. 通过按键 S, 切换为自动时钟, 程序自动执行。当代码运行到 3 号中断安装时, 切换为单步时钟。

我们通过 3 号中断的安装过程, 熟悉安装中断的原理。单步运行, 完成下列步骤:

1. 逐步运行安装 3 号中断的第一条指令 `lea a, int_3`, 该条指令将 3 号中断服务程序的首地址 `int_3` 写入累加器 `a` 中。
2. 运行指令 `mov r0, 0xfb`, 将 `0xfb` 写入 `r0` 中, `0xfb` 指向的存储单元存放的是 3 号中断服务程序的入口地址 `int_3`。
3. 运行指令 `mov [r0], a`, 将 3 号中断服务程序的入口地址 (`int_3`) 写入 `r0` (`0xfb`) 指向的存储单元中, 完成 3 号中断安装。
4. 安装完成后, 打开 `ram` 存储器窗口, 查看 `0xfb` 所指单元的内容, 将其与源代码窗口 1 中标号 `int_3` 下的第一条指令地址相比, 相同则安装正确。
5. 切换为自动时钟, 安装后续的中断。
6. 所有中断安装完成后, 查看存储器窗口 `ram` 中地址为 `0xf8~0xff` 的内容, 分别与中断 `0~7` 的入口地址相比, 相同则所有安装正确。

通过上述的实验步骤, 我们得出这样的结论, 中断的安装过程就是将各个中断服务程序的第一条指令地址 (入口地址/首地址) 分别写入对应中断向量指向的存储单元中。具体到 `Hardware-IRQ`, 中断向量表从地址 `0xf8` 开始 (包含 `0xf8`), 到 `0xff`, 依次存放 `0~7` 号中断的服务程序入口地址。

所有中断安装完成后, 模型机开始运行主程序, 此时就可以允许中断了。我们通过仿真, 跟踪 3 号中断的响应过程。步骤如下:

1. 切换为单步时钟。
2. 通过按键 F, 设置 `IF=1`, 打开中断允许开关。
3. 通过按键 3, 按下然后抬起, 发出一个上升沿, 模拟 3 号中断源发出中断请求。INTR 输出高电平。
4. 单周期运行程序, 直到出现第一条微指令 `ask_for_int`, 打开控制单元 CU, 当前微程序计数器 `uPC` 的输入端的值是 780 (但是还没有写入 `uPC`), 它指向硬中断处理微程序的首地址。结合控制信号 `IF`、中断请求信号 `INTR` 以及 `ASK_FOR_INT`, 分析为什么是这个值?
5. 输入单步时钟, `uPC` 变为 780, 源代码窗口 2 中蓝色箭头指向 780 处, 这是硬中断处理程序的第一条微指令。

通过上述步骤可知, 在允许中断的情况下, 当 CPU 通过微指令 `ask_for_int` 查询到中断请求时, 将 ROM 中固定位置处的硬中断处理微程序首地址传送到 `uPC` 的输入端, 下个时钟上升沿到来时, 写入 `uPC`, 从而跳转到硬中断处理微程序处执行。

1. 输入单步时钟, 运行硬中断处理微程序。结合注释以及单步仿真电路数据通路, 分析每一条微指令的功能是如何实现的? 返回地址入栈后, 从存储器窗口观察入栈后的内容。
2. 当硬中断微程序执行完后, `PC` 转去执行 3 号中断服务程序。
3. 中断服务程序执行完后, 结合指令 `iret` 的执行过程, 理解中断返回地址出栈的过程。
4. 继续单步运行程序, 直到中断返回。
5. 结束仿真。

上述的实验完成了一个硬中断响应的全过程。下面请读者自行模拟三种中断情形 (允许中断的条件下), 并详细记录这三种情形下的工作过程。

- 打开中断允许标志后, 微指令 `ask_for_int` 查询到有多个中断请求的情形。
- 当 CPU 正在运行一个中断服务程序时, 发出一个或多个低优先级中断请求的情形。
- 当 CPU 正在运行一个中断服务程序时, 发出一个或多个高优先级中断请求的情形。

## 四、思考与练习

1. 软件中断和硬件中断有什么不同？从触发机制以及中断处理过程两方面进行分析。
2. 在带有外部中断请求的模型机上运行硬中断服务程序的过程中，若触发一个优先级更高的硬件中断，那么机器将响应新的硬件中断服务程序，即中断嵌套，请通过仿真进行验证？
3. 计算机在运行中断服务程序之前需要保护现场。在 DM1000 中，只需要保存中断返回地址即可，结合仿真过程，分析中断是如何保存返回地址的？又是如何恢复返回地址的？
4. 当计算机在工作时出现异常情况，需要产生一个中断对这些异常情况或特殊请求进行处理，如访问存储器时发生的缺页中断。翻阅资料，了解系统工作时对异常情况的处理过程以及处理结果，分析其对计算机运行效率的影响。
5. 在计算机工作过程中，往往会由于执行某些指令导致异常，比如除法指令的除数为 0，从而导致异常，这时，会触发一个异常中断，调用异常中断服务程序，处理完异常后，返回重新执行发生异常的那条指令。请读者尝试在 DM1000 上做适当的改进，实现异常的处理。
6. 在任务二中，当 DM1000 被中断请求打断时，直到所有中断请求都得到响应和服务后才会返回。若修改中断返回指令 `iret` 的微程序，将 `iret` 中的微指令 `ask_int` 删除，并在最后补充一条占位微指令 `dup 1, null`。DM1000 响应单中断请求的过程有变化吗？多中断呢？为什么。
7. 手动控制硬中断的允许与禁止标志 `IF` 是一件麻烦的事，若能通过指令控制标志 `IF` 就方便多了。请读者在标志寄存器中添加一个 `IF` 标志位，并实现指令 `CLI` 和 `STI`，分别用来将标志寄存器中的中断标志 `IF` 清零或置 1，当 CPU 需要禁止硬中断时，将 `IF` 清零，当 CPU 允许中断时，将 `IF` 置 1。注意，CPU 复位时会将标志寄存器中所有的标志位清零，`IF` 标志也会清零，所以一开始 CPU 是无法响应硬中断的，必须执行一条 `STI` 指令后才能响应硬中断。
8. 尝试为 DM1000 实现中断现场保护功能，也就是在发生中断时，自动将标志寄存器入栈，并将标志寄存器清零，在中断返回时，再自动将标志寄存器出栈。这样，中断服务程序在执行过程中，就可以使用一个新的标志寄存器，而不是被之前的程序修改后的标志寄存器，而且中断服务程序对标志寄存器的修改，也不会影响到被打断程序的运行，因为中断返回后，旧的标志寄存器的值会出栈，被打断的程序仍然可以正常运行。注意，由于中断服务程序开始运行时，标志寄存器被清零了，`IF` 标志也会变为 0（如果已经在标志寄存器中实现了 `IF` 标志位的话），所以如果需要允许中断嵌套的话，必须在中断服务程序开始的位置使用 `STI` 指令允许中断嵌套。

# 实验 9 指令和微指令设计实验

实验性质：设计

建议学时：4 学时

## 一、实验目的

- 掌握指令和微指令的概念。
- 能够修改指令汇编器和微指令汇编器的 C 源代码文件，从而设计新的指令和微指令。

## 二、预备知识

### 指令汇编器源代码文件 dmasm.c

指令汇编器源代码文件 dmasm.c 存放在 DM1000 的项目文件夹中，使用 C 语言编写，可以使用 C 编译器将其编译为 dmasm.exe 可执行文件，作为 DM1000 的指令汇编器，用于将汇编程序编译为可以在 DM1000 中运行的二进制文件。此汇编器使用经典的两遍扫描方法处理汇编代码，其 main 函数的流程图如图 9-1 所示。

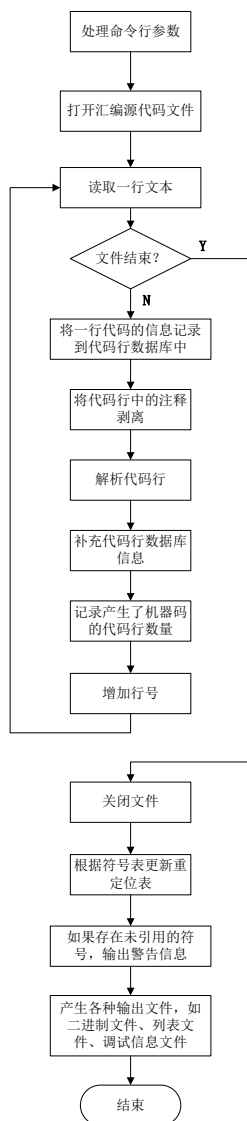


图 9-1: dmasm.c 文件中 main 函数的流程图。

需要特别说明的是，在解析代码行时使用了“表驱动”的编程模式，就是将指令操作码作为关键字，使用该关键字在表 `keyword_function_table` 中查找到对应的表项，然后调用表项中记录的处理函数。所以在表 `keyword_function_table` 的每个表项中都存储了两个元素，第一个元素是关键字字符串指针的地址（指针的指针），第二个元素是处理函数的函数指针。使用表驱动编程方法最大的一个好处就是，当需要增加新的指令时，只需要使用新指令的操作码定义新的关键字，并编写对应的处理函数，然后在表 `keyword_function_table` 中增加一个新的表项就可以了，其他部分的代码基本不用修改。

### 微指令汇编器源代码文件 `microasm.c`

微指令汇编器源代码文件 `microasm.c` 存放在 DM1000 的项目文件夹中，使用 C 语言编写，可以使用 C 编译器将其编译为 `microasm.exe` 可执行文件，作为 DM1000 的微指令汇编器，用于将微指令程序编译为可以在 DM1000 中运行的二进制文件。此汇编器使用一遍扫描方法处理微指令代码，其 `main` 函数的流程图如图 9-2 所示。需要特别说明的是，在解析代码行时也使用了“表驱动”的编程模式，使用的表名称也是 `keyword_function_table`。

此外，在解析 `path` 类型的微指令时，需要根据其使用的两个操作数的不同组合，得到不同的 32 位微指令编码，这个 32 位微指令编码就是存放在 ROM 中的微指令。所以，这里也使用了“表驱动”的编程模式，使用的表名称是 `path_operand_table`。

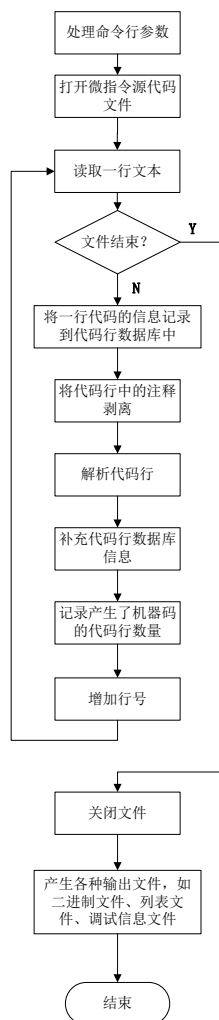


图 9-2: `microasm.c` 文件中 `main` 函数的流程图。

## 三、实验内容



指令系统是计算机硬件的语言系统，也叫机器语言，是机器所具有的全部指令的集合，反映了计算机所拥有的基本功能。机器的指令系统决定了一台计算机的功能，而一旦计算机的指令系统被确定以后，必须有相应的硬件支持。指令系统主要体现在它的操作类型、数据类型、地址格式和寻址方式等方面。

本次实验的重点是让读者通过定义新的指令从而扩展 DM1000 的指令集，并使用新指令编写程序，进行功能验证。

## 2.1 任务（一）

定义新指令 POP A，该指令将栈寄存器 SP 指向存储单元的内容出栈，出栈结果保存到累加器 A 中。

请读者按照下面方法之一在本地创建一个项目，用于完成本次任务：

### 方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建个人项目，然后使用 Dream Logic 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

### 方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，就需要使用 Dream Logic 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的 URL

为 <https://www.codecode.net/engintime/Dream-Logic/Project-Template/DM1000/DM1000.git>。

**建议教师按照下面的步骤在 CodeCode.net 平台上布置此次实验任务：**

- (1) 在计算机组成原理实验课程中，新建一个实验任务。
- (2) 在“新建任务”页面中，教师需要添写“任务名称”，推荐使用“实验 9 设计指令/微指令系统实验-任务（一）”；还需要填写“模板项目 URL”，应该使用  
“<https://www.codecode.net/engintime/Dream-Logic/Project-Template/DM1000/DM1000.git>”。
- (3) 点击“新建任务”按钮，完成新建任务操作。

以下步骤是设计一条 POP A 指令的完整过程，请读者按下列步骤完成 POP A 指令的设计，从而掌握为 DM1000 设计新指令的基本方法。

由于 POP A 指令需要用到的所有微指令都已经存在了，所以并不需要设计新的微指令，不需要修改 microasm.c 文件，也不需要修改任何原理图文件。只需要修改 rom.masm 文件，在其中添加 POP A 指令需要用到的微指令，然后修改 dmasm.c 文件，使指令汇编器能够正确处理新的 POP A 指令即可。

首先修改微指令程序 rom.masm 文件：

1. 通过分析 POPA 指令的数据通路，得到其对应的微指令为：

```

; pop a
path sp, mar      ; 将栈指针寄存器 SP 传送到地址寄存器 mar
path [mar], a     ; 将 SP 指向的存储单元的数据传送到累加器 A，即出栈
path sp, csp      ; sp + 1
path sp_inc, csp
path csp, sp
inc pc            ; pc 加 1，指向下一条指令
reset upc        ; uPC 复位，指向取指微指令

dup 1, null      ; 1 条占位微指令

```

2. 在 DreamLogic 中打开微指令源程序文件 rom.masm，在该文件中找到一块连续的不小于 8 条微指令的空白区域，例如在第 293 行有一块连续区域，可以将其替换为 POP A 对应的微指令。该步操作实际上是将 POPA 对应的微程序写入微程序存储器 rom 中尚未使用的存储空间。

3. 在“项目管理器”窗口中右键单击批处理文件 rom.bat，选择“运行批处理文件”，对微指令源程序进行编译，确保可以正常生成可执行的二进制文件 rom.rxm。
4. 由于 POP A 指令是单操作数指令，所以它的机器码只需要一个字节就可以了，机器码主要用来确定第一条微指令在 ROM 中的偏移位置。所以要想得出 POP A 指令的机器码，就需要知道刚刚添加的微指令在 ROM 中的偏移。读者此时可以启动仿真，在“源代码 2”窗口中会显示出刚刚添加的第一条微指令在 ROM 中的偏移为 0x480，根据在实验 6 中学习的指令译码（偏移）的方法，可以从 0x480 得到其对应的机器码为 0x90。

在添加了 POP A 指令的微指令，并得到其机器码为 0x90 之后，就可以修改 dmasm.c 文件了，使指令汇编器能够正确处理新的 POP A 指令即可：

1. 在 Dream Logic 左侧的“项目管理器”窗口中，右键点击项目节点，选择“打开所在文件夹”，在打开的磁盘目录下找到指令汇编器源文件 dmasm.c 并打开，然后按照下列步骤添加代码：

- 1) 在第 115 行后添加 POP 指令关键字：

```
const char* pop_instruction_keyword = "pop";
```

- 2) 在表 keyword\_function\_table 中添加一个新的表项（第 1574 行后面）

```
, { &pop_instruction_keyword, parse_pop }
```

- 3) 在第 487 行后添加 POP 指令的处理函数。注意，将 POP A 指令的机器码设置为 0x90：

```
void parse_pop(int line_num)
```

```
{
```

```
    char *op;
```

```
    unsigned long op_type;
```

```
    if(assembly_state != AS_TEXT)
```

```
    {
```

```
        warning_msg_invalid_line(line_num);
```

```
        return;
```

```
    }
```

```
    op = strtok(NULL, delimit_char);
```

```
    if(NULL == op)
```

```
    {
```

```
        error_msg_miss_op(pop_instruction_keyword, line_num);
```

```
    }
```

```
    op_type = get_operand_type(op);
```

```
    if(OT_REGISTER_A == op_type)
```

```
    {
```

```
        // pop a
```

```
        machine_code[machine_code_address] = 0x90;
```

```
        machine_code_address++;
```

```
    }
```

```
    else
```

```
    {
```

```
        error_msg_wrong_op(pop_instruction_keyword, line_num);
```

```

    }

    //
    // 在代码行数据库中，标记此行是一个指令行
    //
    line_database[line_count].flag |= LF_INSTRUCTION;
}

```

2. 保存文件 dmasm.c。
3. 任选一种 C 语言编译器将 dmasm.c 文件编译为 dmasm.exe 文件。可选择 Microsoft Visual Studio 或者 GNU GCC 编译器。
4. 用步骤 3 中生成的 dmasm.exe 文件替换项目所在磁盘目录下的指令汇编器文件 dmasm.exe。

至此，DM1000 已经可以运行 POP A 指令了，为了进行测试，请读者按照下面的步骤进行实验：

1. 在“项目管理器”窗口中打开项目下的汇编源程序文件 ram.asm，输入以下代码并保存。此汇编代码将 0xf7 赋值给 sp 寄存器，从而在存储器的 0xf7 位置构造了一个栈顶，然后通过让 r0 寄存器指向栈顶位置，从而在栈顶位置存储了一个字节的值 0xff，最后使用 POP A 指令出栈，将 0xff 放入 a 寄存器中。

```

.text
mov sp, 0xf7
mov r0, 0xf7
mov a, 0xff
mov [r0], a
mov a, 0
pop a

ENDLOOP:
jmp ENDLOOP

```

2. 在“项目管理器”窗口中右键单击批处理文件 ram.bat，选择“运行批处理文件”，对汇编文件进行编译。
3. 在“项目管理器”窗口中打开项目下的原理图文件 DM1000.dlsche 后，启动仿真。
4. 按 C 键单步时钟执行指令，重点查看 POP A 指令的执行情况，确保最终累加器 a 的值为 0xff，完成出栈指令后 sp 寄存器的值加 1 变为 0xf8，并且完成出栈指令后可以继续运行后面的指令。

## 2.2 任务（二）

仿照任务（一）中的步骤，设计 PUSH 和 POP 指令，并使用一段汇编程序对新设计的指令进行验证：

1. 实现 PUSH RX 指令：将一个通用寄存器（R0~R3）中的数据入栈。
2. 实现 POP RX 指令：出栈一个字节，并将这个字节的数据放入通用寄存器（R0~R3）。
3. 实现 PUSH [RX] 指令：将通用寄存器（R0~R3）指向的存储单元中的数据入栈。
4. 实现 POP [RX] 指令：出栈一个字节，并将这个字节的数据放入通用寄存器（R0~R3）指向的存储单元。

## 四、思考与练习

1. 目前 DM1000 实现的指令“IN”是将 外部设备的输入寄存器 RIN 的值传送到累加器 A，而指令“OUT”是将累加器 A 的值传送到外部设备的输出寄存器 ROUT 中。请在 DM1000 模型机的基础上添加输入控制模块以及输出控制模块（主要是添加译码功能），然后改进指令“IN”和“OUT”的微指令，完成多个输入设备向累加器 A 输入数据以及累加器 A 向多个输出设备输出数据。例如，当 2 号输

入设备向累加器 A 输入数据时，指令形式为“IN 2”，同理，当累加器 A 向 0 号设备输出数据时，指令形式为“OUT 0”。

2. 总线是指计算机中多个部件之间公用的一组连线，是若干信号线的集合，由它构成系统模块间的信息通路。在 DM1000 中，包含了数据总线 DBUS 和地址总线 ABUS。若将控制单元 CU 发出的所有控制信号集成总线 CBUS，即可称之为控制总线。请修改 DM1000 电路，实现控制单元 CU 通过控制总线 CBUS 对其他模块的控制。
3. 参考 DM1000 模型机中提供的编译器源代码文件 dmasm.c，用 c 语言编写一个反汇编器，其功能是将 ram.rxm 中的机器码反汇编成 DM1000 中的汇编指令。
4. 在实验 8 的基础上，修改 DM1000 模型机电路，添加 PUSHF/POPF 指令实现标志寄存器 FLAG 的入栈和出栈功能。
5. 读者是否发现在微指令源代码文件中写好指令对应的微指令后，需要知道第一条微指令在 ROM 中的偏移地址，这样才能产生指令的机器码。但是第一条微指令的偏移地址不是很容易获得，一种方法是启动仿真后从“源代码 2”窗口中获得其偏移，但是这种方法不是很方便。请读者修改微指令汇编器的 C 源代码，让微指令汇编器在生成机器码的同时，再生成一个文本格式的映射文件（后缀名为 map），在此映射文件中记录指令的名称及其第一条微指令的偏移。
6. 软件会有缺陷（Bug），处理器也不例外。当 DM1000 的指令存在缺陷时，可以重新编写指令的微指令，然后更新 DM1000 的 ROM 即可解决。但是，请读者设想一种情况，如果一条指令的微指令原来有 8 条，但是修复后变成了 10 条，由于原来位置前后都有其它指令的微指令，所以就必须在 ROM 中另外找一个位置存放修复后的微指令，这就会导致指令的机器码也要发生变化，而之前用到该指令的二进制可执行文件就无法正常运行了，必须重新编译。这种问题对于一款已经推向市场的处理器来说是毁灭性的。请读者设计一种方案，使 DM1000 能够解决这个问题，也就是当指令对应的微指令在 ROM 中的位置发生变化时，之前编译好的二进制可执行文件仍然可以正常运行。

# 实验 10 阵列乘法器实验

实验性质：设计

建议学时：4 学时

## 一、实验目的

- 掌握四位阵列乘法器的设计方法。
- 设计八位阵列乘法器。
- 为 DM1000 的运算器模块 ALU 添加八位阵列乘法器，并实现 MUL 指令。

## 二、预备知识

### 2.1 四位阵列乘法器

无符号二进制乘法和十进制乘法很相似，部分积同样是乘数的一位乘以被乘数的所有位，然后移位这些部分积，并将它们相加就可以得到最后的结果。图 10-1 展示了两个四位二进制数  $X$  和  $Y$  进行无符号二进制乘法得到一个八位二进制数  $P$  的过程，其中  $X_0Y_0$  表示将两个二进制位进行逻辑与运算。总之， $N \times N$  乘法器对两个  $N$  位二进制数相乘，产生一个  $2N$  位的二进制数作为结果。由于 1 位二进制乘法相当于逻辑与运算，所以与门电路用于产生部分积，一位全加器用于部分积相加运算。

$$\begin{array}{r}
 \begin{array}{cccc}
 & X_3 & X_2 & X_1 & X_0 \\
 \times & Y_3 & Y_2 & Y_1 & Y_0 \\
 \hline
 & & X_3Y_0 & X_2Y_0 & X_1Y_0 & X_0Y_0 \\
 & X_3Y_1 & X_2Y_1 & X_1Y_1 & X_0Y_1 & \\
 & X_3Y_2 & X_2Y_2 & X_1Y_2 & X_0Y_2 & \\
 + & X_3Y_3 & X_2Y_3 & X_1Y_3 & X_0Y_3 & \\
 \hline
 P_7 & P_6 & P_5 & P_4 & P_3 & P_2 & P_1 & P_0
 \end{array}
 \end{array}$$

图 10-1：两个四位二进制数  $X$  和  $Y$  的无符号二进制乘法

图 10-2 显示了一个四位阵列乘法器的电路符号和实现。阵列乘法器接收被乘数  $X$  和乘数  $Y$ ，然后产生积  $P$ 。每一个部分积是一个单独的乘数位与被乘数的所有位进行逻辑与运算得出的，同时使用一位加法器将部分积相加，当然还要考虑进位。需要说明的是，将  $Y_0$  与被乘数  $X$  的四个位分别做与运算时，输入的部分积是 0，可以认为只是做了与运算，而没有做加法运算，同时，在最右侧的  $X_0$  与乘数  $Y$  的四个位分别做运算时，进位输入都是 0。

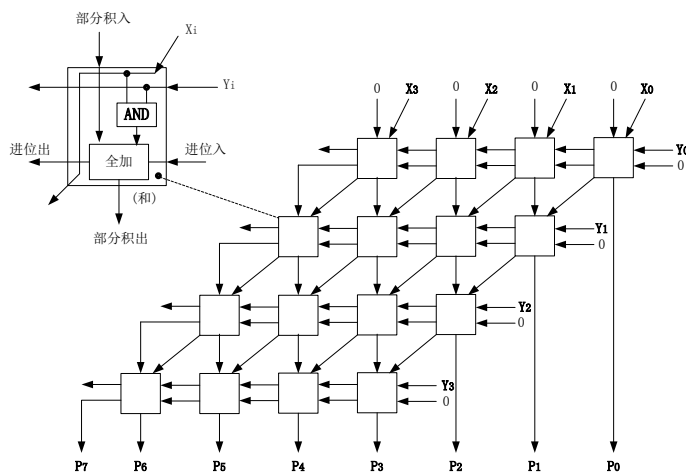


图 10-2：四位阵列乘法器的实现

## 三、 实验内容

### 3.1 任务（一）设计八位阵列乘法器

在本任务中，读者需要学习 DM1000 提供的四位阵列乘法器的实现方法，然后在此基础上将其修改为八位阵列乘法器，并对其进行单独仿真，确保其可以正常工作。

请读者按照下面方法之一在本地创建一个项目，用于完成本次任务：

#### 方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建个人项目，然后使用 Dream Logic 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

#### 方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，就需要使用 Dream Logic 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的 URL

为<https://www.codecode.net/engintime/Dream-Logic/Project-Template/DM1000/DM1000.git>

**建议教师按照下面的步骤在 CodeCode.net 平台上布置此次实验任务：**

- (1) 在计算机组成原理实验课程中，新建一个实验任务。
- (2) 在“新建任务”页面中，教师需要添写“任务名称”，推荐使用“实验 10 阵列乘法器实验-任务（一）”；还需要填写“模板项目 URL”，应该使用<https://www.codecode.net/engintime/Dream-Logic/Project-Template/DM1000/DM1000.git>
- (3) 点击“新建任务”按钮，完成新建任务操作。

#### 四位阵列乘法器

请读者首先从“项目管理器”中打开已经设计完毕的四位阵列乘法器原理图文件 MUL.dlsche 和乘法器运算单元原理图文件 MUL\_unit.dlsche。其中，乘法器运算单元的电路与图 10-2 中描述的是完全一致的，其主要是将乘数的一位 Y 和被乘数的一位 X 做与运算后作为全加器的一个输入，将上一次的部分积 Pin 做为全加器的另外一个输入，同时还有一个进位输入 Cin，然后由一位全加器得到部分积输出 Pout 和进位输出 Cout。

四位阵列乘法器中一共使用了 16 个乘法器运算单元，其电路与图 10-2 中描述的也是完全一致的。需要特别说明的是，所有网络都使用网络标签进行了命名，只有这样才能将各个模块中模块接口连接的网络进行隔离，否则会由于这些网络的名称一致，而产生连接关系，导致它们都连接在一起。例如，所有子模块的 X 接口就算使用网络进行了分别连接（那四个斜线），但是由于这四个网络的名称默认都是 X，也就意味着它们是连接在一起的，只有使用网络标签 X0~X3 将这四个网络分别命名后，它们才是分离的。

#### 设计八位阵列乘法器

读者掌握了四位阵列乘法器电路的设计方法，就可以开始设计八位阵列乘法器了。将四位阵列乘法器的 16 个乘法器运算单元扩展到 64 个就可以实现八位阵列乘法器，但是显然这种方法绘制原理图的工作量比较大。一种更加合理的方法是将四位阵列乘法器修改为可以继续利用的子模块，然后在一个新的原理图中使用 4 个四位阵列乘法器模块就可以设计出八位阵列乘法器了。

读者可以参考下面的步骤完成八位阵列乘法器的设计：

1. 首先需要对现有的四位阵列乘法器进行改造，将其输出和输入信号全部改造为端口（Port），这样才能从模块外部使用这些信号。在已有的八个输入和八个输出信号的基础上，读者还需要：
  - 将 MUL\_unit\_15、MUL\_unit\_14、MUL\_unit\_13、MUL\_unit\_12、MUL\_unit\_8、MUL\_unit\_4、MUL\_unit\_0 模块的每个 Pin 信号分别接一个端口，可以使用名称 Pin0~Pin6，作为输入信号。这些端口可以像 X0~X3 一样放在原理图的顶部。注意每个 Pin 信号与端口连接的网络要使用网络标签进行命名，网络标签的名字可以和端口名称一致。
  - 将 MUL\_unit\_15、MUL\_unit\_11、MUL\_unit\_7、MUL\_unit\_3 模块的 Cin 信号分别接一个端口，可以使用名称 Cin0~Cin3，作为输入信号。这些端口可以像 Y0~Y3 一样放在原理图的右侧。注意每个 Cin 信号与端口连接的网络要使用网络标签进行命名，网络标签的名字可以和端口

- 名称一致。
- 将 MUL\_unit\_12、MUL\_unit\_8、MUL\_unit\_4、MUL\_unit\_12 模块的 Cout 信号分别接一个端口，可以使用名称 Cout0~Cout3，作为输出信号。注意，原来的 P7 端口将会变为 Cout3 端口。这些端口可以放在原理图的左侧。注意每个 Cout 信号与端口连接的网络要使用网络标签进行命名，网络标签的名字可以和端口名称一致。
  - 将原来的 P0~P6 端口的名称修改为 Pout0~Pout6。
2. 在“项目管理器”的“子模块”文件夹上点击鼠标右键，在弹出的菜单中选择“添加”中的“新建文件”，在“新建文件”对话框中选择原理图文件模块，然后在文件名称中填写“8MUL”后点击“确定”按钮，就会新建一个名称为 8MUL 的原理图，用于设计八位阵列乘法器。
  3. 在 8MUL 原理图中点击鼠标右键，在弹出的菜单中选择“从原理图文件新建模块”，在弹出的对话框中选择 MUL.dlsche 文件后，点击“确定”按钮，就可以在原理图中放置一个四位阵列乘法器模块。按“Tab”键可以继续放置同样的模块，直到四个模块都放置完毕。
  4. 移动四个模块中的接口到合适的位置，主要目的是为了更方便连接网络。然后为本模块添加输入端口 X0~X7、Y0~Y7，添加输出端口 P0~P15，并完成所有网络的连接，在连接时要注意部分积的移位。最后，一定要使用网络标签为所有网络命名。

**提示：**

在调整模块中的接口位置时可能会比较繁琐，建议读者可以使用下面的绘图技巧提高效率：

1. 放置对象时最好是参考网格线，便于对齐。
2. 选中对象后，可以使用空格键让对象旋转。
3. 在按下 Ctrl 键的同时用鼠标选中对象，可以选中多个对象。选中多个对象后再进行平移等操作可以大大提高绘图效率。
4. 在按下 Shift 键的同时用鼠标选中对象，可以取消选中对象。
5. 在“属性”窗口中可以编辑所有选中对象的属性。也可以使用“属性”窗口中的筛选下拉列表从选中的对象中继续筛选要编辑的对象。
6. 可以使用“编辑”菜单中的“矩形选择”功能选中多个对象。

在八位阵列乘法器设计完毕后，读者需要对其进行单独仿真，从而确保其可以正常工作。读者可以添加两个八位交互式数字信号，分别与端口 X0~X7、Y0~Y7 连接，用来输入被乘数和乘数，然后再添加四个 16 进制 7 段数码管，与端口 P0~P15 连接，用来显示最后的结果。

**提交作业**

在提交作业前，读者需要将 8MUL 模块 SVG 图形文件的链接添加到 README.md 文件中，这样，当使用浏览器查看提交后的线上项目时，就可以在 README.md 文件中看到 8MUL 模块了。方法如下：

1. 双击“项目管理器”中的 README.md 文件，使用一种合适的文本编辑器或者 Markdown 编辑器打开此文件。
2. 在 README.md 文件的末尾添加如下的文本：  
# 八位阵列乘法器模块 8MUL  
![raw svg](8MUL.dlsche.svg)
3. 保存 README.md 文件。
4. 提交作业。

**3.2 任务（二）为 DM1000 添加 MUL 指令**

在本任务中，读者需要将上一个任务中设计完成的八位阵列乘法器添加到 DM1000 的 ALU 模块中，使之可以将寄存器 A 和 W 分别作为被乘数和乘数，完成无符号数的乘法运算。

请读者按照下面方法之一在本地创建一个项目，用于完成本次任务：

**方法一：从 CodeCode.net 平台领取任务**

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建个



人项目，然后使用 Dream Logic 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

### 方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，就需要使用 Dream Logic 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的 URL

为<https://www.codecode.net/engintime/Dream-Logic/Project-Template/DM1000/DM1000.git>

**建议教师按照下面的步骤在 CodeCode.net 平台上布置此次实验任务：**

- (1) 在计算机组成原理实验课程中，新建一个实验任务。
- (2) 在“新建任务”页面中，教师需要添写“任务名称”，推荐使用“实验 10 阵列乘法器实验-任务（二）”；还需要填写“模板项目 URL”，应该使用<https://www.codecode.net/engintime/Dream-Logic/Project-Template/DM1000/DM1000.git>
- (3) 点击“新建任务”按钮，完成新建任务操作。

请读者按照下面的步骤完成本次任务的准备工作：

1. 使用任务（一）中改造过的四位阵列乘法器模块文件 MUL.dlsche 覆盖本次任务项目文件夹中的 MUL.dlsche 文件。
2. 将任务（一）中设计的八位阵列乘法器模块文件 8MUL.dlsche 复制到本次任务项目文件夹中，然后在 Dream Logic 左侧的“项目管理器”的“子模块”文件夹上点击鼠标右键，在弹出的菜单中选择“添加”中的“现有文件”，弹出添加现有文件对话框，在其中选择刚刚复制到项目文件夹中的 8MUL.dlsche 文件后，点击“打开”按钮，就可以将八位阵列乘法器模块文件添加到本次任务的项目中了。
3. 读者可以尝试仿真一下八位阵列乘法器模块，确保其可以正常工作。

如果八位阵列乘法器可以正常工作，读者就可以参考下面的提示信息为 DM1000 添加 MUL 指令了：

1. 读者首先需要定义 MUL 指令的格式，为了便于实现，可以将其定义为 MUL RX，该指令将寄存器 A 中的数作为被乘数，将寄存器 RX (R0~R3) 中的数作为乘数，将结果的低 8 位保存到寄存器 RX 中，高 8 位保存到寄存器 A 中。
2. 还需要准备一个汇编程序用于验证 MUL 指令是否可以正常执行，例如下面的一段程序用于计算 0x16 乘以 0x24 的积，并将结果的低 8 位保存到寄存器 r0 中，高 8 位保存到寄存器 r1 中：

```
mov a, 0x16
mov r0, 0x24
mul r0
mov r1, a
```

3. 读者需要将八位阵列乘法器放置在 ALU 中，方法是首先打开 ALU 的原理图，然后在原理图中点击鼠标右键，选择菜单中的“从原理图文件新建模块”，在弹出的对话框中选择 8MUL.dlsche 文件后，点击确定按钮，就可以将八位阵列乘法器模块放置在 ALU 中了。注意，建议读者先在 8MUL.dlsche 文件中为所有端口设置好输入输出类型，例如将端口 X0~X7 和 Y0~Y7 设置为输入类型，将端口 P0~P15 设置为输出类型，这样在放置模块的时候，模块接口 X0~X7 和 Y0~Y7 就会出现在模块的左边，用于输入数据，模块接口 P0~P15 就会出现在模块的右边，用于输出数据。
4. 在 ALU 中放置好八位阵列乘法器后，就可以进行网络连接了。网络连接很简单，只要将寄存器 A 的输出接乘法器的 X0~X7，将寄存器 W 的输出接乘法器的 Y0~Y7，然后将乘法器输出的低 8 位通过一个 8 位单向总线收发器（在型号库“桶形移位器和总线收发器”中）输出到数据总线，将乘法器输出的高 8 位也通过一个 8 位单向总线收发器输出到数据总线。最后，还需要为这两个新添加的 8 位总线收发器的使能端提供控制信号，使用 CU 模块中 U2 器件上的两个还未被使用的控制信号即可。这样，虽然乘法器总是会计算寄存器 A 和 W 的乘积，但是可以通过 8 位总线收发器决定是否将结果放在数据总线上。



- 读者可以利用实验 9 中学到的知识，设计 MUL 指令和它的微指令，当然还需要修改指令汇编器和微指令汇编器的 C 源代码，使它们支持新的 MUL 指令和微指令。在设计微指令时，读者也需要使用一定的技巧，下面的代码给出了一种可以工作的微指令。需要读者注意的是，一定要先将结果的低 8 位通过数据总线传送给 RX 寄存器，然后才能将结果的高 8 位通过数据总线传送给寄存器 A，请读者思考一下其中的原因。

```
path rx, w
path alu_mul_low, rx
path alu_mul_high, a
inc pc
reset upc
```

### 提交作业

在提交作业前，读者需要将 ALU 模块 SVG 图形文件的链接添加到 README.md 文件中，这样，当使用浏览器查看提交后的线上项目时，就可以在 README.md 文件中看到 ALU 模块了。方法如下：

- 双击“项目管理器”中的 README.md 文件，使用一种合适的文本编辑器或者 Markdown 编辑器打开此文件。
- 在 README.md 文件的末尾添加如下的文本：  
# 添加乘法器后的运算器模块 ALU  
![raw svg](ALU.dlsche.svg)
- 保存 README.md 文件。
- 提交作业。

## 四、思考与练习

- 改进任务（二）中的 ALU 模块，在进行乘法运算时，如果乘法器的运算结果为 0，需要将标志寄存器 FLAG 中的 ZF 位设置为高电平，否则将其置为低电平。注意，需要与原有的 74LS181 计算的结果设置 ZF 位的电路协同工作，不能产生冲突。
- 对于无符号乘法，是不存在进位和溢出问题的，请读者说明原因。因此，原有的 CF 标志位在乘法运算时就没有意义了，但是为了方便检查字节相乘的结果是一个字节还是一个字（两个字节），可以当乘积的高 8 位为 0 时，将 CF 标志位置为低电平，否则置为高电平。请读者在任务（二）的 ALU 模块中实现此功能。注意，需要与原有的 74LS181 计算的结果设置 CF 位的电路协同工作，不能产生冲突。
- 请读者自行查阅资料，实现阵列除法器。

## 第三部分 MIPS 微体系结构处理器实验

MIPS 微体系结构处理器实验包含四章内容。

**第 1 章**是 MIPS 体系结构的概述，主要介绍 MIPS 指令格式，MIPS 指令的特点：规整，简单，执行速度快。

**第 2 章**是单周期 MIPS 微处理，该部分在单周期 MIPS 微处理器上进行实验，引导读者掌握 MIPS 指令编程，并通过单周期 MIPS 微处理运行 MIPS 指令程序，使读者理解单周期 MIPS 微处理器指令的数据通路以及电路实现原理。

**第 3 章**是多周期 MIPS 微处理，该部分在多周期 MIPS 微处理器上进行实验，通过多周期 MIPS 微处理运行 MIPS 指令程序，使读者理解多周期 MIPS 微处理器指令的数据通路以及电路实现原理。

**第 4 章**是五级流水线 MIPS 微处理，该部分以理想五级流水线 MIPS 微处理器为例，通过理想五级流水线 MIPS 微处理运行 MIPS 指令程序，使读者了解流水线微处理器执行指令的机制和原理。

理解理想五级流水线 MIPS 微处理器的原理后，读者需要在理想五级流水线的基础上，逐步改进电路，依次解决写后读数据冲突、LW 指令数据冲突、控制冲突，最终设计一个解决所有冲突的五级流水线 MIPS 微处理器。

该部分实验按照由基础到复杂，由易到难的顺序进行，逐步引导读者深入理解 MIPS 体系结构微处理器，培养读者的计算机系统设计能力。

# 第 1 章 体系结构概述

## 一、 体系结构

在本章，我们将介绍体系结构的相关概念，并介绍 MIPS 微体系结构、16 位 MIPS 指令及其格式。

体系结构是程序员所见到的计算机的属性，即计算机的逻辑结构和功能特征，它由指令集（汇编语言）和操作数地址（寄存器和存储器）来定义，计算机体系结构不定义底层的硬件实现。现在有不同类型的体系结构，例如 x86、MIPS、SPARC 和 PowerPC 等。对计算机系统设计者，计算机体系结构是指研究计算机的基本设计思想和由此产生的逻辑结构；对程序设计者是指对系统的功能描述（如指令集、编制方式等）。

理解任何计算机体系结构的第一步是学习它的语言。计算机语言中的单词叫作指令。计算机的词汇表叫作指令集。在同一个计算机上运行的所有程序使用相同的指令集。即使非常复杂的软件应用程序也最终编译为一序列诸如加法、减法或跳转的简单指令。计算机指令包含需要完成的操作和需要使用的操作数两部分，其中操作数来自存储器、寄存器或者指令本身。

### 1.1 机器语言

计算机硬件只能处理二进制信息，所以指令也编码为二进制数，其格式称为机器语言。

机器语言是用二进制代码表示的计算机能直接识别和执行的一种机器指令的集合。它是计算机的设计者通过计算机的硬件结构赋予计算机的操作功能。机器语言具有灵活、直接执行和速度快等特点。

正如使用字母来编码人类的语言一样，计算机使用二进制数编码机器语言。

### 1.2 微处理器

微处理器由一片或少数几片大规模集成电路组成的一个可以读并执行机器语言指令的数字系统。这些电路执行控制部件和算术逻辑部件的功能。

微处理器能完成取指令、执行指令，以及与外界存储器和逻辑部件交换信息等操作，是微型计算机的运算控制部分。它可与存储器和外围电路芯片组成微型计算机。

微处理器的基本组成部分有：寄存器堆、运算器、时序控制电路，以及数据和地址总线。

### 1.3 汇编语言

因为人类直接阅读二进制格式的机器语言会非常困难，所以使用符号格式来表示机器指令，称为汇编语言。

汇编语言是汇编指令集、伪指令集和使用它们规则的统称，使用具有一定含义的符号为助忆符，用指令助忆符、符号地址等组成的符号指令称为汇编格式指令。

### 1.4 体系结构的指令集

不同体系结构的指令集更像不同的方言，而不是不同的语言。几乎所有的体系结构都定义了基本指令，例如加法、减法和跳转，对存储器和寄存器进行操作。一旦学习了一个指令集，理解其它指令集就相当简单了。

### 1.5 微体系结构

寄存器、存储器、ALU 和其他模块形成微处理器的特定方式称为微体系结构。

## 二、MIPS 体系结构

MIPS 体系结构是由美国计算机科学家 John Hennessy 和他的同事于 20 世纪 80 年代在斯坦福大学首先提出的。MIPS 处理器得到了广泛的应用，例如 Silicon Graphics、Nintendo 和 Cisco 等公司都采用了这种处理器。

Patterson 和 Hennessy 提出了 MIPS 体系结构设计的 4 个准则。

### 2.1 简单设计

汇编语言是计算机机器语言的人类可阅读表示。每条汇编语言指令都指明了需要完成的操作和操作所处理的操作数。最常见的计算操作是加法。MIPS 汇编中的加法和减法代码示例如下。

```
add a, b, c
```

```
sub a, b, c
```

汇编指令的第一部分 add (sub) 是助记符，它指明需要执行的操作。该操作基于源操作数 b 和 c，将结果写入目的操作数 a。示例说明减法类似于加法，除了操作码不同以外，减法指令格式完全与加法指令相同。

这种一致的指令格式很好地证明了第一个设计原则：简单设计有助于规整化。指令格式中包含了固定数目的操作数（在本例中，有 2 个源操作数和一个目的操作数）将易于编码和硬件处理。更复杂的高级语言代码可以转化为多条 MIPS 指令，比如高级语言代码：

```
a=b+c-d
```

就可以通过两条 MIPS 指令：

```
sub t, c, d // t = c-d
```

```
add a, b, t // a = b+t
```

使用多条汇编指令执行复杂的操作体现了 MIPS 计算机体系结构的第二个设计准则：加快常见功能。

### 2.2 加快常见操作

MIPS 指令集只包含简单的常见指令，这些指令完成简单的操作，能较快执行。也就是使常见的基本操作能快速执行，复杂但不常见的操作由多条简单指令序列执行。因此，MIPS 属于精简指令集计算机（简称 RISC）体系结构。具有复杂指令的体系结构，如、Intel x86，称为复杂指令集计算机（简称 CISC）。例如，x86 中定义的“字符串移动”指令将字符串从内存的一个位置复制到另一个位置。这样的操作需要很多条（很可能上百条）RISC 机器的简单指令。CISC 体系结构实现复杂指令的是要付出代价的，增加硬件而且降低了简单指令执行的速度。

RISC 体系结构在降低硬件复杂性的同时，还使得在指令编码中区分不同操作的位数比较少。例如，有 64 条简单的指令集需要  $\log_2 64 = 6$  位来编码每个操作。有 256 条复杂指令的指令集需要  $\log_2 256 = 8$  位来编码每条指令。在 CISC 处理器中，即使复杂指令使用的频率非常小，它们也增加了复杂指令连带的简单指令的开销。

通过比较发现：MIPS 指令集虽然简单，只包含常见的基本操作，但是它通过多步简单操作同样可以实现复杂操作，相较于 CISC 代码，这会导致执行指令数目的增加，但是 MIPS 指令执行较快。

### 2.3 越小的设计越快

一条指令的操作要基于操作数。操作数可以存放在寄存器或存储器中，也可以作为常数直接编码到指令中。访问指令中的常数或寄存器中的操作数非常快，但是它们只能包含少量数据。更多的数据需要访问存储器得到。存储器虽然容量大，但是访问速度比较慢。

只有快速访问操作数，指令才能快速执行。但是存放在存储器中的操作数需要较长时间才能访问到。因此，大多数体系结构定义了几个寄存器用于存放常用操作数。寄存器越少，访问速度越快，这验证了北京英真时代科技有限公司 <http://www.engintime.com>

第三个设计准则：越小的设计越快。

## 2.4 折中设计

MIPS 使用装入字指令 `lw` 将存储器中读出的数据装入寄存器中。`lw` 指令指定内存有效地址为基地址与偏移量的和。基地址存放在寄存器中，偏移量是一个立即数。

MIPS 中加立即数 (`addi`) 指令是一个以立即数为操作数的常见指令。`addi` 将指令指定的立即数与某一个寄存器中的值相加，结果写回指令寄存器中。

前面提到的 `add` 和 `sub` 指令使用三个寄存器操作数，而 `lw` 和 `addi` 使用两个寄存器操作数和一个常数。因此，指令格式不同，所以 `lw` 和 `addi` 指令违反了设计原则 1：简单设计有助于规整化。因此引入了折中设计原则。

MIPS 指令集支持三种指令格式。

- 第一种是 R（寄存器）类型指令，包含三个寄存器操作数。
- 第二种是 I（立即数）型指令，包含两个寄存器和一个立即数。
- 第三种是 J（跳转）类型指令，只有一个立即数作为操作数。

## 三、 MIPS 指令

在 32 位 MIPS 体系结构处理器中，指令是 32 位编码的，包含 32 个各类寄存器。详情请参考附录 B，该附录包含 32 位 MIPS 指令编码以及常见的 32 位 MIPS 指令。

我们在 Dreamlogic 平台上设计的是 16 位 MIPS 微处理器，它们只有 4 个寄存器（可作为指令操作数的寄存器，不包含其他的非操作数寄存器），每个寄存器都是 16 位的。下面介绍 Dream Logic 开发的 MIPS 系列微处理器使用的 16 位 MIPS 指令编码格式。

注意：16 位 MIPS 指令与 32 位 MIPS 指令编码格式相同，只是每个字段的位数不同。字段的位数差别，主要体现在可编码的 MIPS 指令数目的多少，可用作操作数的寄存器数目的多少，ALU 支持的运算方式的多少，可移位数的多少，指令立即数的大小等。

### 3.1 R 类型指令

R 类型指令有 3 个寄存器操作数：2 个为源操作数，1 个为目的操作数。图 1-1 给出了 R 类型指令格式。

op	rs	rt	rd	shamt	funct
15~12	11~10	9~8	7~6	5~4	3~0

图 1-1：R 类型机器指令格式

- `op` 字段：操作码，编码指令类型，所有 R 类型指令的操作码都是 0。
- `funct` 字段：函数，特定的 R 类型操作由函数字段决定。类如，`add` 指令的 `funct` 字段为 2 (0010)，表示 ALU 做加法运算。而 `sub` 指令的 `funct` 字段为 3 (0011)，表示 ALU 做减法运算。
- `rs`, `rt`, `rd` 字段：编码寄存器，`rs` 和 `rt` 是源操作数寄存器，`rd` 是目的操作数寄存器。2 位编码可以寻址 4 寄存器。
- `shamt` 字段：仅仅用于移位操作，该字段只在移位指令 `sll`（逻辑左移）、`srl`（逻辑右移）、`sra`（算术右移）中有作用，表示移位数。

指令	15~12	11~10	9~8	7~6	5~4	3~0	指令功能
----	-------	-------	-----	-----	-----	-----	------

or	0000	rs	Rt	rd	00	000	$\$rd = \$rs \mid \$rt$
and	0000	rs	Rt	rd	00	001	$\$rd = \$rs \& \$rt$
add	0000	rs	Rt	rd	00	010	$\$rd = \$rs + \$rt$
sub	0000	rs	Rt	rd	00	011	$\$rd = \$rs - \$rt$
sllv	0000	rs	Rt	rd	00	100	$\$rd = \$rs \ll \$rt$
srlv	0000	rs	Rt	rd	00	101	$\$rd = \$rs \gg \$rt$
srav	0000	rs	Rt	rd	00	110	$\$rd = \$rs \gg \$rt$
slt	0000	rs	Rt	rd	00	111	$\$rd = (\$rs < \$rt) ? 1 : 0$

表 1-2: 16 位 MIPS 处理器中 R 类型指令格式表

### 3.2 I 类型指令

I 类型指令有两个寄存器操作数和一个立即数操作数。图 1-3 给出了 I 型机器指令格式。

op	rs	rt	imm
15~12	11~10	9~8	7~0

图 1-3: I 类型指令格式

- op 字段: 操作码, 编码指令类型, 不同 I 类型指令的操作码不同。例如: lw 指令的操作码为 6 (110), beq 指令的操作码为 8 (1000)。
- rs 字段: rs 源操作数寄存器, 2 位编码可以寻址 4 寄存器。
- rt 字段: 在有些指令中 (如 addi 和 lw), rt 用作目的操作数; 而在其他指令中 (如 sw), rt 用作源操作数。
- imm 字段: 立即数字段, 表示 8 位立即数。负立即数表示为 8 位有符号二进制补码数。

I 类型指令有一个 8 位立即数字段, 但这个立即数用于 16 位操作中。例如, lw 将 8 位偏移量与 16 位源寄存器相加。所以, 需要将 8 位立即数进行位扩展。对于负立即数, 高 8 位应该全是 1, 称为符号扩展。逻辑操作 (andi、ori、xori) 将 0 放在高 8 位, 这称为 0 扩展。

指令	15~12	11~10	9~8	7~6	5~4	3~0	指令功能
ori	0011	rs	rt	immediate-u			$\$rt = \$rs \mid \text{imm}$
andi	0100	rs	rt	immediate-u			$\$rt = \$rs \& \text{imm}$
addi	0101	rs	rt	immediate-s			$\$rt = \$rs + \text{imm}$
lw	0110	rs	rt	immediate-s			$\$rt = \text{MEM}[\$rs + \text{imm}]$
sw	0111	rs	rt	immediate-s			$\text{MEM}[\$rs + \text{imm}] = \$rt$
beq	1000	rs	rt	offset-s			beq=?
bne	1001	rs	rt	offset-s			bne!=?
bgt	1010	rs	rt	offset-s			bgt>?

表 1-4: 16 位 MIPS 处理器中 I 型指令格式表

### 3.3 J 类型指令

J 类型指令格式仅仅用于跳转指令。其指令格式如图 1-5 所示。

op	imm
15~12	7~0

图 1-5: J 类型指令格式

- op 字段: 操作码, 编码指令类型。
- imm 字段: 立即数操作数, 用于指定地址。

指令	15~12	11~10	9~8	7~6	5~3	2~0	指令功能
jump	1011	jump address					jump

表 1-6: 16 位 MIPS 处理器中 J 型指令格式表

### 3.4 存储程序

用机器语言编写的程序是一个表示指令的一序列二进制数。与其他二进制数一样, 这些指令存储在存储器中。这就是存储程序的概念, 也是计算机如此强大的一个关键原因。运行一个新的程序不需要花费大量的时间和精力对硬件进行重新装配或重新布线, 只需要将新的程序写入存储器中。存储程序提供了通用计算能力, 而不是特定的硬件。在这种方式下, 计算机只改变存储程序就可以运行计算器、文字处理程序、影音播放器等多种应用程序。

存储程序中的指令从存储器中取出, 由处理器执行。即使大量复杂程序也可以简化为一序列存储器读和指令执行。

在 MIPS 处理器中, 存储地址是字节寻址, 顺序执行程序时, 16 位指令地址 PC 每次增加 2 (字节) 才能获取下一条指令, 而 32 位指令则需要 PC 每次增加 4 (字节) 才能指向下一条指令。

运行存储程序时, 处理器从存储器中顺序取出指令, 然后, 数字电路硬件解码和执行这些取出的指令。当前指令的地址存放在一个程序计数器 (PC) 的寄存器中。

## 四、 MIPS 微体系结构处理器

### 4.1 微体系结构

微体系结构是将寄存器、ALU、有限状态机、存储器和其他逻辑模块组合在一起, 实现一种体系结构。一个特定的体系结构可以有不同的微体系结构。

### 4.2 MIPS 微体系结构

- **单周期微体系结构**在一个时钟周期中执行一条完整的指令。该结构易于解释且控制单元简单。由于它在一个周期内完成操作, 时钟周期由最慢的指令决定。
- **多周期微体系结构**利用多个较短的时钟周期执行一条指令。简单指令的执行周期数较少。而且, 多周期微体系结构可以通过对加法器和存储器等昂贵硬件部件的复用减少硬件成本。例如, 同一个加法器可以在一条指令的不同时钟周期中用于不同的目的。多周期处理器在任意时刻只执行一条指令, 但是

每条指令需要多个时钟周期。

- **流水线微体系结构**将单周期微体系结构流水线化，使得同时可以执行多条指令，显著提高了吞吐量。流水线结构必须增加一些逻辑来处理多条正在执行指令之间的相关性。同时，还需要非体系结构流水线处理器（非指令操作数寄存器）。增加这些逻辑和寄存器是值得的。当前，所有商业高性能处理器都使用流水线结构。

## 五、 利用 Dream Logic 开发 MIPS 微处理器

Dream Logic 是一款使用简单，功能强大的数字电路仿真软件，集电路原理图绘制、原理图编译、可视化交互式仿真于一体。

无需掌握 Verilog、VHDL 等硬件描述语言，可直接利用其丰富的元器件库绘制图形化的电路原理图，功能多样的数字芯片、丰富的基本逻辑门器件、方便快捷的电路连接方式、高效的电路功能模块化设计，充分满足了从基本门级电路到复杂系统级电路功能多样性设计，大大提高了电路设计效率。Dream Logic 全面支持系统级模块化自顶向下和自底向上的设计方式，大大方便了复杂系统的设计。

支持从门级到系统级的可视化交互式仿真。基于 Dream Logic 的功能支撑，我们在 Dream Logic 上开发了 3 种 MIPS 微体系结构处理器：MIPS 单周期微处理器、MIPS 多周期和 MIPS 五级流水线微处理器，这些 MIPS 微体系结构处理器实现了 MIPS 指令系统的一个子集，具体包括：

- R 型算术/逻辑指令：add、sub、and、or、slt、sllv、srlv 和 srav。
- 扩展指令：addi、ori、andi。
- 存储器指令：lw、sw。
- 条件分支指令：beq、bne、bgt。
- 无条件分支指令（跳转指令）：jump。

在后续的章节中，我们将一一介绍利用 Dream Logic 开发的 MIPS 微体系结构处理器，并引导读者在现有基础上改进微处理器，完成诸如指令扩展，解决流水线数据冲突，解决流水线控制冲突等实验任务。最终我们希望读者通过改进现有处理器（改进处理器电路）实现以下指令扩展后的 MIPS 微处理器：

- 丰富指令后的 MIPS 单周期微处理器。
- 丰富指令后的 MIPS 多周期微处理器。
- 丰富指令后（不包含 lw 指令和分支指令），通过数据重定向解决写后读（RAW）冲突的 MIPS 五级流水线微处理器。
- 丰富指令后（不包含分支指令），利用阻塞解决 lw 指令数据冲突，同时包含数据重定向解决数据冲突功能的 MIPS 五级流水线微处理器。
- 丰富指令后（包含所有指令），解决所有冲突（控制冲突和数据冲突）的 MIPS 五级流水线微处理器。



## 第 2 章 单周期 MIPS 微处理器

### 一、单周期 MIPS 微处理器设计过程

通过分析指令的数据路径，从而得到构成单周期处理器的各个功能模块。

#### 1.1 程序计数寄存器

程序计数寄存器 PC 保存访问指令地址。

#### 1.2 指令存储器

利用 MIPS 指令编写汇编程序，然后使用编译器将汇编程序编译为机器码指令程序。机器码指令程序保存在指令存储器中。

#### 1.3 指令译码控制模块

指令取出后，经过译码控制模块译码，输出特定的控制信号，这些控制信号作为相关组件的输入，从而决定它们执行何种操作。

比如，lw 指令功能是将数据存储器中指定地址单元的内容加载到指定寄存器中，为了实现这一功能，译码单元对 lw 指令译码后输出两个关键信号，一个是 MemtoReg 信号，在该信号作用下，选择从数据存储器中指定地址单元读出的内容写回目的寄存器，另一个是寄存器写信号，该信号使能，使得读出的结果能写入目的寄存器中。

#### 1.4 寄存器文件

寄存器文件包含指令中可以使用的所有寄存器，微处理器可以从寄存器文件中读出源寄存器操作数进行运算，然后将运算结果写回寄存器文件中的目的寄存器。

#### 1.5 ALU 模块

ALU 模块接收两个操作数，并在指令译码信号控制下进行算术逻辑运算、移位等操作，是处理器的核心功能模块。

#### 1.6 数据存储器

数据存储器在单周期处理器中作为一个独立部件，只接受 lw 和 sw 指令的访问。可通过 lw 指令将数据存储器中指定地址单元的数据读出并加载到目的寄存器中，也可通过 sw 指令将源寄存器的内容保存到数据存储器中指定位置。

#### 1.7 其他部件

除了以上这些基本的组件外，还需要一些满足特定需要的功能部件作为补充。比如，符号扩展部件，该部件主要是将指令中的 8 位立即数根据需要扩展为 16 位。此外，还需要复选器，用于选择不同的数据来源，使其流向下一个目的地。比如 PC 复选器，用于从下一条指令（顺序执行）地址、条件分支程序地址（条件成立，执行分支程序）、跳转指令地址（无条件跳转）中选择一个地址，作为新的 PC 值，以便下个时钟周期取指执行。

划分好基本模块后，我们就可以根据各个模块所要实现的功能进行单独设计。电路模块的功能定义决定了模块的输入输出信号，设计模块的首要步骤是根据电路模块功能确定所有的输入/输出信号以及各信号的作用。

将经过功能验证的各个模块组装成完整的单周期 MIPS 微处理器，如图 2-1 所示。

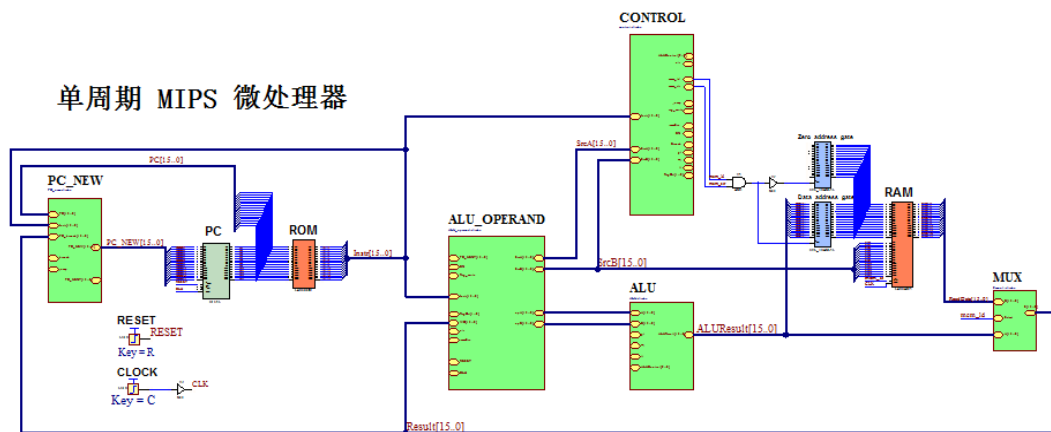


图 2-1: 单周期 MIPS 微处理器

## 二、实验任务

请读者按照下面的内容完成实验，扩展 MIPS 单周期微处理器的指令集。扩展后所有 MIPS 指令如以下 3 个表格所示，其中红色显示的是扩展指令，需要读者改进 MIPS 单周期微处理器才能实现。

如果读者在学习本次实验之前已经系统学习过本书第二部分的实验内容，在学习本实验的内容时就会相对容易。否则，强烈建议读者首先按照本书第二部分中的实验一，详细了解 Dream Logic 是如何设计和仿真一个处理器的。

指令	15~12	11~10	9~8	7~6	5~4	3~0	指令功能
or	0	rs	rt	rd	0	0	$\$rd = \$rs   \$rt$
and	0	rs	rt	rd	0	1	$\$rd = \$rs \& \$rt$
add	0	rs	rt	rd	0	2	$\$rd = \$rs + \$rt$
sub	0	rs	rt	rd	0	3	$\$rd = \$rs - \$rt$
sllv	0	rs	rt	rd	0	4	$\$rd = \$rs \ll \$rt$
srlv	0	rs	rt	rd	0	5	$\$rd = \$rs \gg \$rt$
srav	0	rs	rt	rd	0	6	$\$rd = \$rs \gg \$rt$
slt	0	rs	rt	rd	0	7	$\$rd = (\$rs < \$rt) ? 1 : 0$
xor	0	rs	rt	rd	0	8	$\$rd = \$rs \oplus \$rt$
nor	0	rs	rt	rd	0	9	$\$rd = \sim (\$rs   \$rt)$
jr	1	rs	rt	rd	0	0	$pc = \$rs$
jalr	2	rs	rt	rd	0	0	$\$ra = pc + 2, pc = \$rs$
sll	12	rs	rt	rd	0	4	$\$rd = \$rs \ll shamt$
srl	13	rs	rt	rd	0	5	$\$rd = \$rs \gg shamt$

提示：符号“ $\oplus$ ”表示异或运算，符号“ $\sim$ ”表示取反。

表 2-2: 16 位 MIPS 处理器中 R 类型指令格式表

指令	15~12	11~10	9~8	7~6	5~4	3~0	指令功能
ori	3	rs	rt	immediate-u			$\$rt = \$rs   imm$
andi	4	rs	rt	immediate-u			$\$rt = \$rs \& imm$
addi	5	rs	rt	immediate-s			$\$rt = \$rs + imm$
lw	6	rs	rt	immediate-s			$\$rt = MEM[\$rs + imm]$
sw	7	rs	rt	immediate-s			$MEM[\$rs + imm] = \$rt$
beq	8	rs	rt	offset-s			beq=?
bne	9	rs	rt	offset-s			bne!=?
bgt	10	rs	rt	offset-s			bgt>?
bltz	14	rs	rt	offset-s			$\$rs < 0 ?$

表 2-3: 16 位 MIPS 处理器中 I 型指令格式表

指令	15~12	11~10	9~8	7~6	5~4	3~0	指令功能
jump	11	jump address					jump
jal	15	jump address					$\$ra = pc+2, \text{ jump}$

表 2-4: 16 位 MIPS 处理器中 J 型指令格式表

## 2.1 从 CodeCode.net 平台领取任务

CodeCode.net 平台是用于教师在线布置实验任务，并统一管理学生提交的作业的平台。如果没有使用到 CodeCode 平台，可以跳过此部分，直接从 2.2 节继续进行实验。

1. 读者通过浏览器访问<https://www.codecode.net>，可以打开 CodeCode.net 平台的登录页面。
2. 在登录页面中，读者输入帐号和密码，点击“登录”按钮，可以登录 CodeCode.net 平台。
3. 登录成功后，在“课程”列表页面中，可以找到 MIPS 微体系结构处理器相关的实验课程。点击此课程的链接，可以进入该课程的详细信息页面。
4. 在课程的详细信息页面中，可以查看课程描述信息，该信息对于完成实验十分重要，建议读者认真阅读。点击左侧导航中的“任务”链接，可以打开任务列表页面。
5. 在任务列表中找到本次实验对应的任务，点击右侧的“领取任务”按钮，可以进入“领取任务”页面。
6. 在“领取任务”页面中，查看任务信息后，填写“新建项目名称”和“新建项目路径”，然后选择“项目所在的群组”，点击“领取任务”按钮后，可以创建个人项目用于完成本次实验，并自动跳转到该项目所在的页面。

**提示：**在“领取任务”页面中，“新建项目名称”和“新建项目路径”这两项的内容可以使用默认值，如果选择的群组中已经包含了名称或者路径相同的项目，就会导致领取任务失败，此时需要修改新建项目名称和新建项目路径的内容。

7. 在个人项目页面中，包括左侧的导航栏、项目信息、文件列表、readme.md 文件内容等，如图 2-5 所示。
8. 在个人项目页面的图 2-5 中，点击红色方框中的按钮，可以复制当前项目的 URL，在本实验后面的练习中会使用这个 URL 将此项目克隆到读者计算机的本地磁盘中，从而供读者进行修改。

**提示：**领取任务的本质是：教师在新建任务时填写了实验模板的 URL，然后读者在领取任务时，根据任务中记录的 URL 来派生（Fork）出一个新的项目，供读者在其中进行修改。

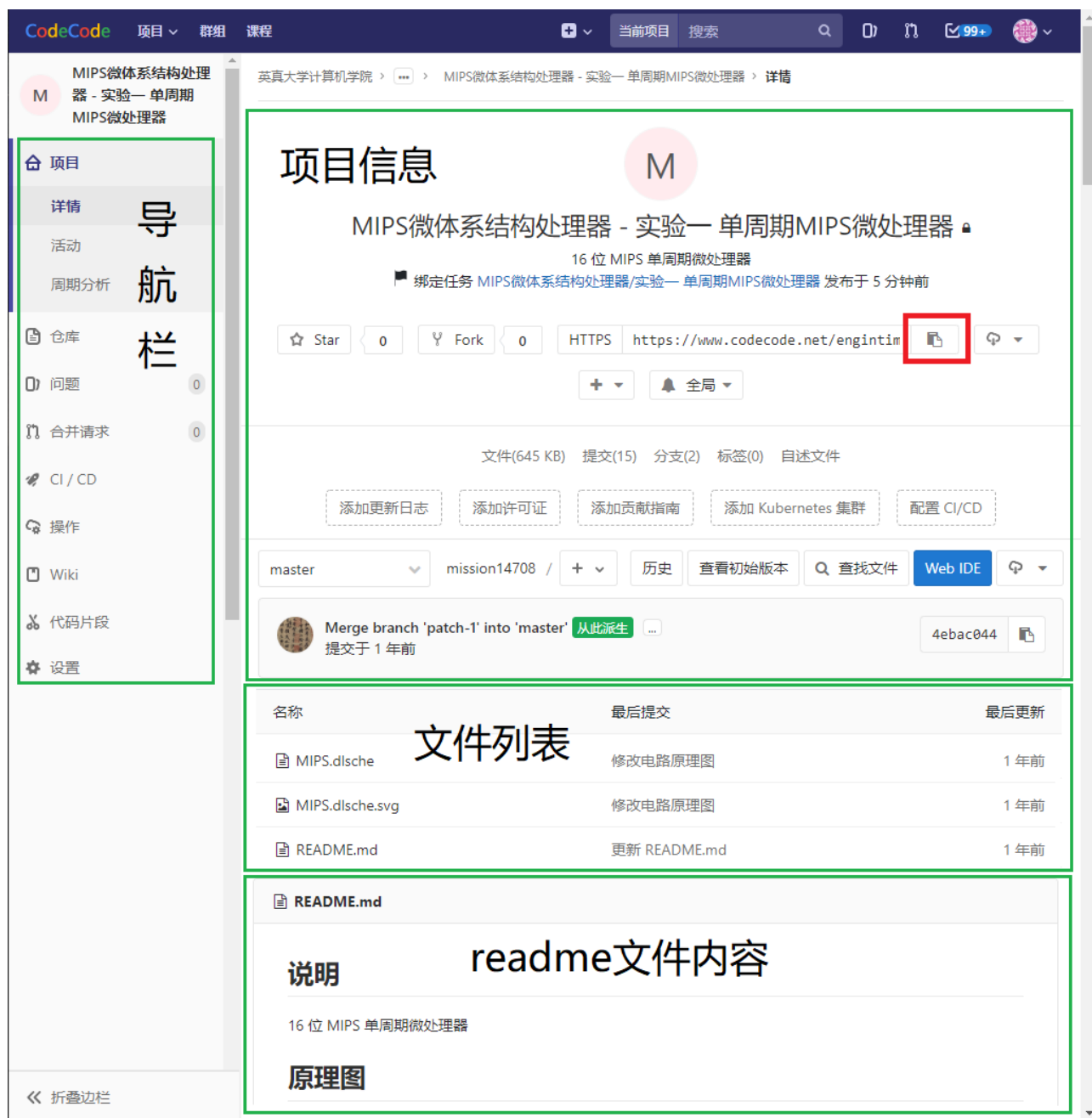


图 2-5：个人项目页面

## 2.2 启动 Dream Logic

在安装有 Dream Logic 的计算机上，可以使用两种方法来启动 Dream Logic：

- 在桌面上双击“Engintime Dream Logic”图标。  
或者
- 点击“开始”菜单，在“所有程序”中的“Engintime Dream Logic”中选择“Engintime Dream Logic”即可启动程序。

启动 Dream Logic 后会首先打开“登录”窗口，可以使用两种方法完成登录。

- 如果读者在 CodeCode.net 平台上注册了帐号，在连接互联网的情况下，可以使用 CodeCode.net 平台中已注册的用户名和密码进行登录。

- 如果读者还没有 CodeCode.net 平台上的帐号，可以点击左下角“从加密锁获取授权”按钮，获取授权之后，完成登录。

## 2.3 将 CodeCode.net 平台上的项目克隆到本地

如果读者从 CodeCode.net 平台领取了任务，可以在领取任务后将读者的个人项目克隆到本地磁盘中；如果读者没有 CodeCode.net 平台账号无法领取任务，可以直接将实验模板项目克隆到本地磁盘中。

### 将领取任务新建的个人项目克隆到本地

1. 选择 Dream Logic 菜单“文件->新建->从 Git 远程库新建项目”，打开“从 Git 远程库新建项目”对话框。
2. 在“Git 远程库 URL (G)”中填入读者个人项目的 Git 远程库的 URL 地址（在图 2-5 中，点击红色方框中的按钮，可以复制当前项目的 URL）。
3. “项目文件夹名称”填入“mips-single-cycle”。
4. “项目位置”选择新建项目需要保存到的磁盘目录。
5. 点击“确定”按钮后会弹出一个 Windows 控制台窗口，并开始运行 Git 克隆命令将 Git 远程库克隆到本地，一定要等克隆成功后再关闭该窗口。
6. 克隆项目成功后，在对话框中选择“克隆成功，打开新建项目”就可以打开新建的项目。

### 直接将实验模板项目克隆到本地

由于 CodeCode.net 平台上提供的实验模板是开放的，所有的人都可以访问。所以，Dream Logic 可以直接将 DM1000 八位模型机实验模板克隆到本地磁盘。步骤如下：


1. 选择 Dream Logic 菜单“文件->新建->从 Git 远程库新建项目”，打开“从 Git 远程库新建项目”对话框。
2. 在“Git 远程库 URL (G)”中项目模板 Git 远程库的 URL 地址：  
<https://www.codecode.net/engintime/Dream-Logic/Project-Template/MIPS/mips-single-cycle.git>
3. “项目文件夹名称”填入“mips-single-cycle”。
4. “项目位置”选择新建项目需要保存到的磁盘目录。
5. 点击“确定”按钮后会弹出一个 Windows 控制台窗口，并开始运行 Git 克隆命令将 Git 远程库克隆到本地，一定要等克隆成功后再关闭该窗口。
6. 克隆项目成功后，在对话框中选择“克隆成功，打开新建项目”就可以打开新建的项目。

#### 注意：

1. 为了能正常使用从 Git 远程库新建项目功能，用户需要在本地安装 Git 客户端程序，并且在安装 Git 的过程中会有一个步骤询问是否设置 Windows 环境变量，此时一定要设置上 Windows 环境变量。
2. 文件路径中不允许包含中文，请读者在使用之初就养成使用英文命名项目或文件的习惯。

## 2.4 任务（一）

新建单周期 MIPS 微处理器项目后，按下列实验步骤进行实验。

1. 打开项目子文件夹“源程序文件”下的“rom.asm”MIPS 源程序文件，结合注释了解 16 位 MIPS 指令和程序功能。
2. 选择项目子文件夹“源程序文件”下的批处理文件“rom.bat”，单击右键，选择弹出菜单中的“运行批处理文件”即可将“rom.asm”中的 MIPS 源程序编译为二进制机器码指令文件。生成的机器码文件已经预先加载到了处理器的程序存储器 ROM 中了，可以在原理图中找到 ROM 并查看其属性中的相关信息。
3. 打开原理图文件“MIPS.dlsche”。
4. 选择菜单“仿真->启动仿真”（快捷键为 F5），启动仿真时要确保焦点在原理图“MIPS.dlsche”中。
5. 按下键盘按键“R”初始化微处理器内部的相关寄存器，为执行程序做准备。
6. 按下键盘按键“C”，然后抬起，这样就给微处理器输入了一个时钟脉冲“”。每输入一个时钟脉

冲，处理器执行一条指令。

**提示：**如果读者是第一次使用 Dream Logic，请务必在本书第一部分的实验一中找到原理图常用操作的介绍内容，并自行练习，为后面的实验内容做好准备。

在指令的执行过程中注意观察左侧源代码窗口中的蓝色箭头指向。该蓝色箭头指向当前正在执行的指令。指令执行可以分为取指、译码、执行（运算）、存储、写回寄存器等五个步骤，分析每个步骤是如何实现的？每个步骤主要由哪部分电路完成？哪些步骤是由指令译码信号控制的，这些部件接收什么样的控制信号，实现什么样的功能。

**注意：**寄存器只能在时钟上升沿到来时才能写入，所以，指令执行完成后并没有将结果立刻写入寄存器中，而是当下个时钟上升沿到来时才写入结果，同时也取出了下一条指令并执行。

## 2.5 任务（二）实现 xor、nor 指令

### 实验要求

在单周期 MIPS 微处理器的基础上改进电路，实现 xor（异或）指令，nor（或非）指令，并进行仿真验证。

扩展指令及其功能：

xor 指令格式：xor \$rd, \$rs, \$rt。将源操作寄存器 rs 和 rt 中的值进行异或运算，结果写回目的寄存器 rd 中。xor 指令属于寄存器类型指令。

nor 指令格式：nor \$rd, \$rs, \$rt。将源操作寄存器 rs 和 rt 中的值进行或非运算，结果写回目的寄存器 rd 中。nor 指令也属于寄存器类型指令。

### 测试方法

电路改造完成后，运行测试程序，检验新增指令的功能是否正确实现了。测试程序如下：

```
.text
addi $r0, $r0, 0xff
addi $r1, $r1, 0xf0
xor $r2, $r0, $r1
nor $r3, $r1, $r1
```

测试程序运行后的结果为：r0 = 0xff, r1 = 0xf0, r2 = 0x0f, r3 = 0x0f。

### 提示

在算术逻辑运算器（项目子模块下的原理图“al\_operation.dlsche”）中只包含了加、减、与、或运算，需要增加“异或”和“或非”两种运算来满足 xor、nor 指令的运算需要。可以此作为突破口，进行电路改进。需要注意的是，译码模块需要对 4 位 funct 字段进行译码，得到 ALU 的运算方式，这样，4 位编码总共可以译出 16 种运算方式。

## 2.6 任务（三）实现 sll、srl 指令

### 实验要求

在任务二的基础上进一步改进微处理器电路，实现逻辑左移指令（sll）和逻辑右移指令（srl），并进行仿真验证。

扩展指令：

逻辑左移指令 sll 格式：sll \$rd, \$rt, imm。指令功能是将 rt 寄存器中的值逻辑左移 shamt 表示的

位数，结果写回目的寄存器 rd 中。

逻辑右移指令 srl 格式：sll \$rd, \$rt, imm。指令功能是将 rt 寄存器中的值逻辑右移 shamt 表示的位数，结果写回目的寄存器 rd 中。

指令中的 imm 表示 2 位无符号立即数 ( $0 \sim 4$ )，因为移位数是指令编码中的 shamt 字段，而 shamt 字段只有 2 位编码（指令的 5~4 位），所以，最多可以左/右移 4 位。虽然这两条指令包含了两个寄存器操作数和一个立即数操作数，但是它们的指令编码格式是 R 类型，所以这两条指令都属于 R 类型指令。

## 测试方法

电路改造完成后，运行测试程序，检验新增指令的功能是否正确。测试程序如下：

```
.text
addi $r0, $r0, 1
addi $r1, $r1, 8
sll $r0, $r0, 2
srl $r3, $r1, 1
```

程序运行结果为：r0 = 4, r1 = 8, r3 = 4。

## 提示

首先需要改造控制单元，使其能对 sll、srl 指令译码产生必要的控制信号；其次，需要增加组件，将 16 位机器码指令中的 shamt 字段取出并进行无符号扩展成 16 位，与 rt 寄存器分别作为 ALU 的两个操作数进行移位运算。

## 2.7 任务（四）实现 bltz 指令

### 实验要求

在任务（三）的基础上进一步改进单周期 MIPS 微处理器，实现小于 0 则转移指令（bltz），并验证。

扩展指令：

bltz 指令格式：bltz rs, label。指令功能是，如果寄存器 rs 的值小于 0，则转移到标号 label 处运行程序。

## 测试方法

电路改造完成后，运行测试程序，检验 bltz 指令的功能是否正确。测试程序如下：

```
.text
addi $r0, $r0, -1      ; r0 = -1
addi $r1, $r1, 8       ; r1 = 8
bltz $r0, F2           ; r0 < 0, 程序跳转到 F2 处
```

```
F1:
sll $r0, $r0, 2
srl $r3, $r1, 1
```

```
F2:
sub $r1, $r0, $r0      ; r1 = 0
bltz $r1, F1           ; r1 不小于 0, 程序不跳转
add $r1, $r1, $r0      ; r1 = r1 + r0 = -1
```

完成 bltz 指令功能后，假设 bltz 指令实现的是大于或等于 0 则转移功能，电路该作何修改。

### 提示

可以参考条件分支指令 beq rs, rt, label。bltz 指令中需要将 rs 寄存器与常数 0 比较，可以考虑在寄存器文件中添加一个常数寄存器，该寄存器始终保存常数 0。将 rs 寄存器与该常数寄存器中的值 0 取出进行比较，确定程序是否分支。

## 2.8 任务（五）实现 jal、jr、jalr 指令

### 实验要求

在任务（四）的基础上继续改进单周期 MIPS 微处理器，实现跳转并链接指令（jal），跳转寄存器指令（jr），跳转和链接寄存器（jalr）。它们的组合使用可以实现程序调用和返回。

#### 1. 扩展指令 jal:

jal 格式: jal label。

jal 指令的功能: 将下一条指令地址 PC+2（因为一条 16 位 MIPS 指令占用两个字节，而 PC 表示的是字节地址，所以 PC 加 2 指向下一条指令）保存到返回地址寄存器 ra 中，然后跳转到标号 label 处执行程序。

#### 2. 扩展指令 jr:

jr 指令格式: jr rs。

jr 指令功能: 将寄存器 rs 的值作为地址加载到 PC 中，从而实现程序跳转。

#### 3. 扩展指令 jalr:

jalr 指令格式: jalr rs。

jalr 指令功能: 将返回地址 PC+2 保存到地址寄存器 ra 中，然后将 rs 寄存器的值加载到 PC 中，实现跳转。

### 测试方法

电路改造完成后，运行测试程序，检验新增指令的功能是否正确。测试程序如下（假设返回地址寄存器为 ra）:

```
.text
addi $r0, $r0, -1      ; r0 = -1
addi $r1, $r1, 8       ; r1 = 8
jal F2                  ; ra = 6, 跳转到 F 处
sll $r0, $r0, 2         ; 程序调用返回地址指令
srl $r3, $r1, 1         ;
jump end

F:
sub $r1, $r0, $r0       ; r1 = 0
add $r1, $r1, $r0       ; r1 = r1 + r0 = -1
jr $ra                  ; PC=6, 程序调用返回

end:
jump end                ; 死循环
```

### 提示



1. 扩展指令 jal 可以参考跳转指令 jump 进行电路改造, 需要指出的是, 现有单周期 MIPS 微处理器的寄存器文件中, 没有返回地址寄存器 ra, 所以需要读者自行添加并设计相关的数据路径。
2. jr 指令实现相对简单, 可以从寄存器文件中引出一条数据路径到 PC, 然后使用复选器选择寄存器的值加载到 PC 中。jal 指令与 jr 指令搭配使用可以实现程序调用和返回。

### 思考与练习

以上程序中没有包含嵌套调用和递归的情况, 若想实现嵌套调用和递归, 该如何改造电路? 提示: 嵌套调用和递归与简单调用的区别是, 嵌套和递归有多个返回地址, 所以不能使用地址返回寄存器存放这些返回地址, 而应将这些返回地址按序存放在存储器中, 我们可以称之为入栈, 调用返回时依次取出返回地址, 称之为栈展开。因此, 需要一个栈寄存器保存栈顶地址, 同时需要改造电路, 实现入栈和出栈的相关操作。每次程序调用都需要将返回地址入栈, 而每次调用返回, 出栈的同时, 还需要更新栈顶指针寄存器。更新栈顶寄存器只需将栈寄存器的值做加减法即可。由于每个返回地址占用两个字节, 所以出/入栈只需将栈寄存器的值加/减 2 即可。

嵌套和递归作为扩展练习, 感兴趣的读者可以在完成前面任务的基础上对电路进一步改造, 同时还需要重新设计调用和返回指令 (除了更新 PC 以外, 还需更新栈寄存器)。

### 2.9 提交作业

如果读者是通过从 CodeCode.net 平台领取任务创建的个人项目, 并将个人项目克隆到本地进行实验, 实验结束后可以将本地已更改的项目再推送到 CodeCode.net 平台的个人项目中, 方便教师通过 CodeCode.net 平台查看读者提交的作业。步骤如下:

1. 在项目管理器窗口中, 右键点击项目节点, 在弹出的菜单中, 选择“Git”中的“推送当前分支到 Git 远程库”菜单项, 弹出“推送当前分支到 Git 远程库”的对话框。
2. 在“推送当前分支到 Git 远程库”对话框中, 输入本次项目更改的提示信息。
3. 点击“推送”按钮, 可以将当前项目推送到 CodeCode.net 平台的个人项目中。

可以按照下面的步骤查看推送后的项目:

1. 使用浏览器访问<https://www.codecode.net>, 使用已注册的帐号登录 CodeCode.net 平台。
2. 在“课程”列表中选择对应的实验课程, 打开实验课程的详细信息页面, 点击左侧导航栏的“任务”链接, 打开任务列表页面。
3. 在任务列表页面打开本次实验对应的任务, 在任务详情中点击“查看项目详情”按钮, 可以跳转到读者的个人项目页面。
4. 在个人项目页面中, 可以查看当前项目的全部内容。
5. 在项目左侧的导航栏中点击“仓库”中的“提交”链接, 可以打开提交列表页面。
6. 在提交列表页面中, 点击最后一次提交, 可以查看此次提交发生变更的文件。

**技巧:** 在 Dream Logic 的项目管理器窗口中, 选中项目节点, 点击鼠标右键, 在弹出的右键菜单中, 选择“Git”中的“使用浏览器访问 Git Origin”菜单项, 可以自动打开本地项目在 CodeCode.net 平台上对应的个人项目。

## 第 3 章 多周期 MIPS 微处理器

### 一、多周期 MIPS 微处理器的开发背景

单周期 MIPS 微处理器存在几个方面的缺点。第一，它需要足够长的周期来完成最慢的指令（1w），即使大部分指令的速度都非常快。第二，它需要 3 个加法器（一个用于 ALU，两个用于 PC 的逻辑），而加法器是相对占用芯片面积的电，尤其是如果它们的速度比较快。第三，它采用独立的指令存储器和数据存储器，而这在实际系统中是不现实的。大多数计算机有一个单独的大容量存储器来存储指令和数据，并且支持读写操作。

多周期 MIPS 微处理器通过将指令执行过程分解为多个较短的步骤来解决这些问题。在每个短步骤中，处理器可以读或写存储器或寄存器文件，或者使用 ALU。不同的指令使用不同的步骤，因此简单指令可以比复杂指令更快地执行。处理器只需要一个加法器，这个加法器在不同的步骤中可以根据不同的目的重复使用。而且处理器使用一个可以存储指令和数据的组合存储器。第一步从存储器中取出指令，而在后续的步骤中可以从存储器读出数据或将数据写入存储器，也就是在不同的步骤中，同一个存储器可以作为指令存储器或者数据存储器重复使用。

我们使用与单周期处理器相同的方法来设计多周期处理器。首先，我们通过将寄存器和存储器与组合逻辑连接来创建一条数据路径。在设计中，我们增加了非体系结构状态元件（寄存器）来保存每个步骤的中间结果。接着，我们设计控制器。在每条指令执行期间控制器针对不同的步骤产生不同的控制信号，控制器是一个有限状态机而不是组合逻辑。

通过逐条分析指令的数据路径，得到寄存器、存储器、组合逻辑模块之间的连接关系。最终得到多周期 MIPS 微处理器的电路原理图，如图 3-1 所示。

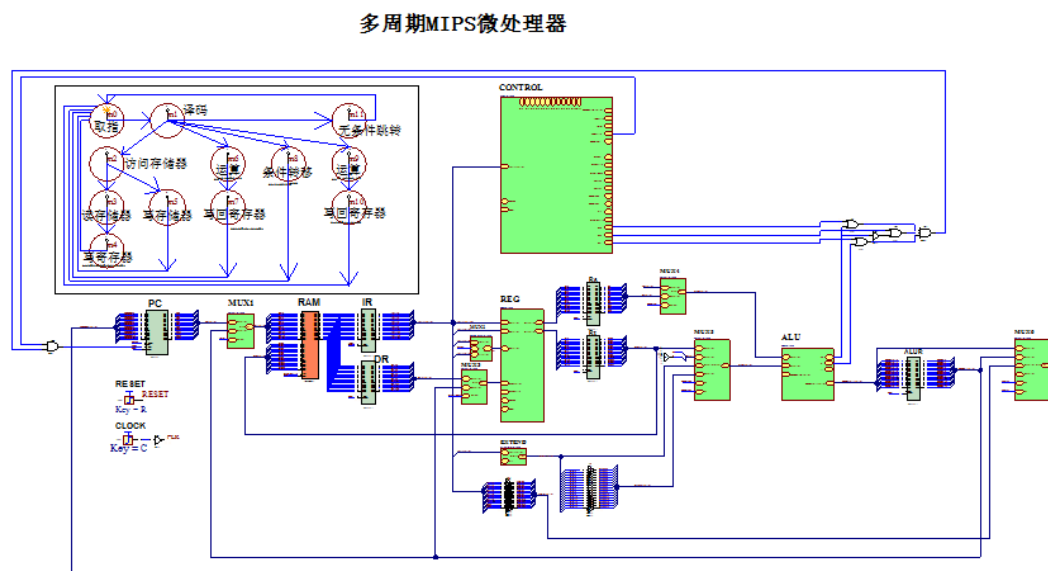


图 3-1：多周期 MIPS 微处理器

### 二、实验任务

请读者按照下面方法之一在本地创建一个多周期 MIPS 微处理器项目，用于完成本次实验：

**方法一：从 CodeCode.net 平台领取任务**

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建个人项目，然后使用 Dream Logic 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。


**方法二：不从 CodeCode.net 平台领取任务**

如果读者不使用 CodeCode.net 平台，就需要使用 Dream Logic 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的 URL 为

<https://www.codecode.net/engintime/Dream-Logic/Project-Template/MIPS/mips-multi-cycle.git>

**2.1 任务（一）**

通过多周期 MIPS 微处理器运行 MIPS 指令程序，熟悉多周期 MIPS 微处理器的电路结构以及执行指令的原理。

1. 打开项目子文件夹“源程序文件”下的“ram.asm”程序文件，进一步熟悉 MIPS 指令功能，可以利用 MIPS 指令编写程序。
2. 选择项目子文件夹“源程序文件”下的批处理文件“ram.bat”，单击右键，选择弹出菜单中的“运行批处理文件”即可将“ram.asm”中的程序编译为机器码文件。生成的机器码文件已经预先加载到了处理器的程序存储器 RAM 中了，可以在原理图中找到 RAM 并查看其属性中的相关信息。
3. 打开原理图文件“MIPS.dlsche”，启动仿真。
4. 按下键盘按键“R”复位微处理器，此时可以看到处理器处于取指状态，准备取出存储器中 0 地址处的指令。
5. 按下键盘按键“C”，然后抬起，这样就给微处理器输入了一个时钟脉冲“”。每输入一个时钟脉冲，微处理器完成指令执行过程中的一个步骤。

在多周期 MIPS 微处理器执行指令时，注意观察微处理器的状态图，图中包含了当前处理器所处的状态，每条路径上的状态数目表示该指令执行需要的时钟周期数目。左侧源代码窗口中的蓝色箭头可能指向当前正在执行的指令，也可能指向当前执行指令的下一条指令。

观察每条指令执行过程中经过了哪些状态，每个状态下，哪些部件在工作，完成了什么功能。通过运行程序，着重理解有限状态机是如何控制微处理器逐步实现指令功能的。了解有限状态机的相关理论和知识，为后续指令扩展，电路修改做准备。

**注意：**寄存器只能在时钟上升沿到来时才能写入，所以，在写回寄存器的状态下并没有将结果立刻写入寄存器中，而是当下个时钟上升沿到来时才写入结果，同时微处理器也进入了取指状态，等待取出了下一条指令并执行。

**2.2 任务（二）多周期 MIPS 微处理器指令扩展****● 实验要求**

参照单周期 MIPS 微处理器指令扩展，在多周期 MIPS 微处理器上实现对应指令。

- (1) 实现 R 类型指令 xor、nor、sll、srl。
- (2) 实现 bltz 指令。
- (3) 实现 jal、jr、jalr 指令。

**● 测试方法**

电路改造完成后，请自行设计测试程序，通过仿真，检验处理器是否正确执行了新增指令。如有错误，反复检查和修改电路，直到多周期 MIPS 微处理器能全部正确执行所有指令。

**● 提示**

现有的 MIPS 多周期微处理器的状态图如图 3-2 所示。

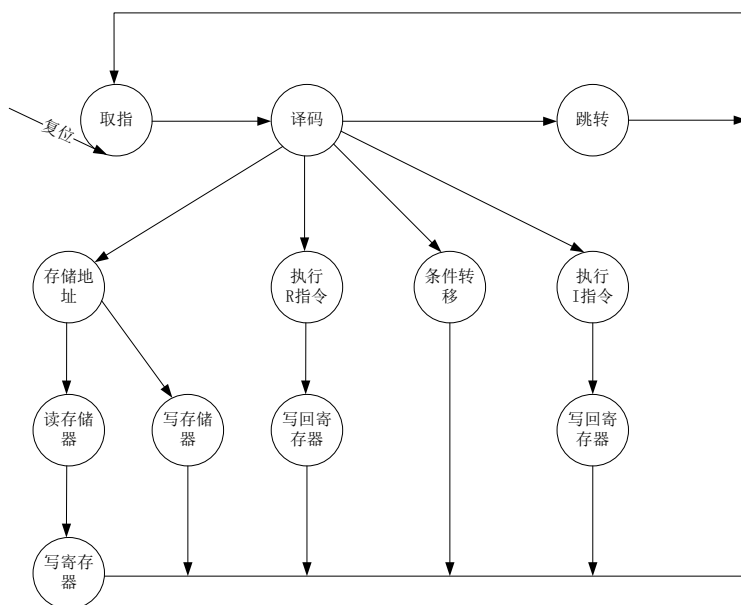


图 3-2: 多周期 MIPS 微处理器状态图

1. 实现 R 类型指令 xor、nor、sll、srl。从图 2 中可以看出，所有的 R 类型指令依次经过取指、译码、运算、写回寄存器共 4 个状态（时钟周期或步骤）。所以新增指令 xor、nor、sll、srl 不需要增加新的状态，它们的主要不同体现在运算的操作数以及运算方式方面。因此，可以在运算状态下设置不同的控制信号，以便控制 ALU 使用指令规定的操作数进行指定的运算，得到正确的结果。
2. 实现 bltz 指令。bltz 指令属于条件分支指令，从图 2 中可以看出，所有条件分支指令依次经过取指、译码、条件转移共 3 个状态。所以，bltz 指令也不会导致状态机的状态数增加，条件分支指令的不同之处在于判断转移条件是否成立。可以在译码阶段利用 ALU 计算 bltz 转移条件是否成立，可以参考 bne 等已经实现的分支指令。
3. 实现 jal、jr、jalr 指令。jal、jr 属于无条件跳转指令，从图 2 中可以看出，无条件跳转指令依次经过取指、译码、无条件跳转 3 个状态。与 jump 指令相比，jal 指令可在译码状态进入无条件跳转状态时将 PC+2 写入地址返回寄存器 ra（该寄存器需要读者自行添加到寄存器文件中），而 jr 指令可在无条件跳转状态产生一个有效复选信号，该信号负责从跳转寄存器读出的地址和无条件跳转地址中选择一个值作为新的 PC，当其有效时（jr 指令）选择跳转寄存器读出的地址作为 PC 值，当其无效时（jump 指令），选择无条件跳转地址作为新的 PC 值。
4. 主要改进的电路模块是译码控制单元，结合状态与指令，根据需要增加新的控制信号，控制相关部件完成特定的步骤。

## 第 4 章 五级流水线 MIPS 微处理器

### 一、流水线技术

流水线技术是提高数字系统吞吐量的有效手段。通过将单周期处理器分解成 5 个流水线阶段来构成流水线处理器。因此，可以在每阶段流水线中同时执行 5 条指令。由于每阶段仅有整个逻辑的 1/5，所以时钟频率几乎可以提高 5 倍。因此，虽然每条指令的延迟并未改变，但理想情况下吞吐量可以提高 5 倍。微处理器每秒执行上百万条甚至数十亿条指令，所以吞吐量比延迟更重要。流水线引入了一些开销，因此吞吐量不能到达理想要求那么高，但是流水线依然有小成本的强大优势，所有现代高性能微处理器都使用流水线技术。

读/写存储器和寄存器文件、使用 ALU 通常构成处理器中的最大延迟。我们选择 5 个流水线阶段，每一个阶段只完成一个慢操作。具体地，我们称这 5 个阶段为：取指令(Fetch)、译码 (Decode)、执行 (Execute)、存储 (Memory) 和写回 (WriteBack)。它们类似于多周期处理器中执行 1w 指令的 5 个步骤。流水线示意图如图 4-1 所示。

- 在取指阶段，处理器从指令存储器中读取指令。
- 在译码阶段，处理器从寄存器文件中读取源操作数并对指令译码以便产生控制信号。
- 在执行阶段，处理器使用 ALU 执行计算。
- 在存储器阶段，处理器读或写数据存储器。
- 在写回阶段，如果需要，处理器将结果写回到寄存器文件。

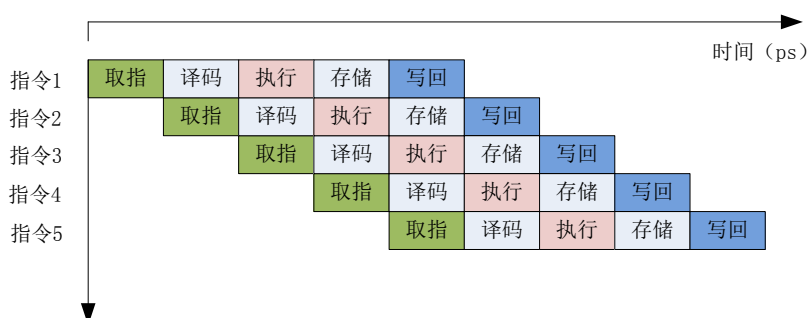


图 4-1：理想流水线处理器程序运行示意图

#### 1.1 流水线中的冲突

流水线系统中的核心问题是化解冲突。理想五级流水线处理器是不处理任何冲突的流水线系统，它是建立在所运行的程序不会导致任何冲突的基础之上的，而现实中是不可能的。那么，流水线系统中主要存在哪些冲突呢？

1. 写后读 (RAW) 冲突：当一条指令写寄存器，而后一条指令读这个寄存器时，将产生冲突。通俗地说，后一条指令依赖于前面指令的执行结果，而前面指令的执行结果还没来得及写入寄存器，后一条指令就读取该寄存器的值，进行运算，这就导致后一条指令计算错误的结果。
2. 1w 指令数据冲突：1w 指令是将数据存储器中的数据读出，然后写入指定寄存器中。若 1w 指令执行结果还没写入寄存器，后续指令就读取该寄存器的值进行运算，同样会导致错误的计算结果。
3. 控制冲突：在取指令时还未确定下一条指令应取的地址时将发生控制冲突。在执行 beq 条件转移指令

时将产生控制冲突，因为在取下一条指令时分支是否发生还尚未确定，所以流水线处理器不知道该取那条指令。

接下来，我们将在 MIPS 五级理想流水线处理器的基础上开展任务，逐步改进处理器，直至其能处理所有的冲突。

## 二、实验任务

请读者按照下面方法之一在本地创建一个五级理想流水线 MIPS 微处理器项目，用于完成本次实验：

### 方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建个人项目，然后使用 Dream Logic 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。


### 方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，就需要使用 Dream Logic 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的 URL 为

<https://www.codecode.net/engintime/Dream-Logic/Project-Template/MIPS/mips-pipeline.git>

### 2.1 任务（一）

运行 MIPS 程序，理解指令执行过程以及微处理器的内部电路原理。

1. 打开项目子文件夹“源程序文件”下的“rom.asm”程序文件，熟练掌握 mips 指令编程。
2. 选择项目子文件夹“源程序文件”下的批处理文件“rom.bat”，单击右键，选择弹出菜单中的“运行批处理文件”即可将“rom.asm”中的程序编译为机器码文件。生成的机器码文件已经预先加载到了处理器的程序存储器 ROM 中了，可以在原理图中找到 ROM 并查看其属性中的相关信息。
3. 打开原理图文件“MIPS.dlsche”，启动仿真。
4. 按下键盘按键“R”复位微处理器，准备取出存储器中 0 地址处的指令。
5. 按下键盘按键“C”，然后抬起，这样就给微处理器输入了一个时钟脉冲“”。每输入一个时钟脉冲，取出一条新的指令，已经取出的指令进入译码阶段，已经译码的指令进入执行阶段，已经执行的指令进入存储阶段，处于存储阶段的指令进入写回阶段。

为了便于观察流水线各个阶段中处理的指令，我们设计了一个阶梯型显示器，如图 4-2 所示。从图中可以看出，此时流水线中填充了五条指令，地址为 4 的指令已经执行到了写回阶段，地址为 6 的指令处于存储阶段，地址为 8 的指令正在执行，而地址为 A（10）的指令在译码，地址为 C（12）的指令即将被取出。注意图中“红色”和“灰色”文本表示的不同含义。“灰色”表示指令经过的流水线阶段，而红色表示指令当前时刻所处流水线阶段。



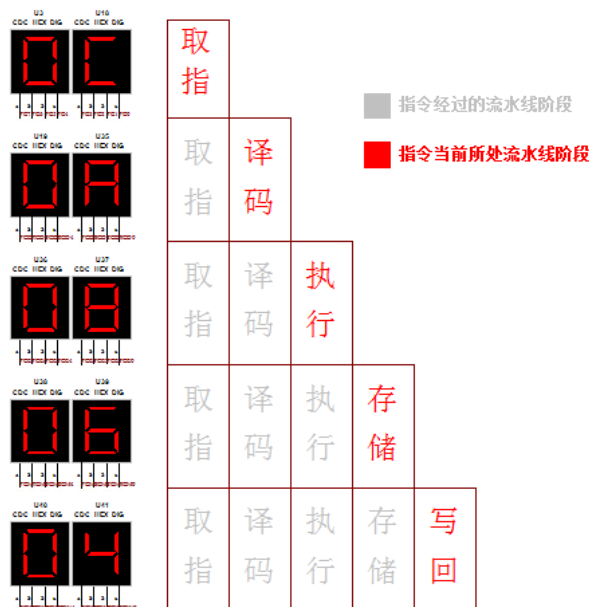


图 4-2：流水线处理器显示器

6. 结合图中的显示器，逐步执行程序，了解流水线处理器的工作原理。

**总结：**流水线处理器中各个阶段之所以可以同时分工，独立做不同的工作而不相互影响，是因为各个阶段是通过中间寄存器隔离开的，就好比不同的车间是通过墙壁隔离开一样，每个阶段产生的结果只有在时钟的驱动下才能传递到下一个阶段，下一个阶段接上个阶段完成的“半成品”继续加工，依次类推，最后一个阶段完成一个“成品”，也就是指令执行完了。

从取指到最终写回寄存器之间的各个阶段是相互独立的，但是它们之间又是协同工作的。可以类比为生产流水线，上一个车间（流水线阶段）完成的半成品（输出）以及尚未加工的原材料（没有使用的控制信号）必须一起传送到下一个车间（流水线阶段），该车间利用半成品和部分尚未使用的原材料加工得到一个新的半成品或成品（成品不需要传到下个车间再行加工），依次类推，直到所有材料（指令包含的所有信息）都加工成成品（完成指令功能）才退出生产线（流水线）。

**注意：**寄存器只能在时钟上升沿到来时才能写入，但是流水线中目的寄存器的写与单周期和多周期处理器不同，流水线中为了能在一个周期内先读寄存器旧值，然后将新值写入寄存器，所以，流水线利用时钟的后半个周期的下降沿将值写入目的寄存器中。这样，我们可以看到，在指令的写回阶段，目的寄存器的值就写入了，而无需等待下个时钟上升沿。

以上内容介绍了如何使用 MIPS 理想五级流水线处理器运行程序。按下列实验步骤进行实验，了解流水线中的写后读（RAW）数据冲突。

1. 打开项目下的程序源文件“rom.asm”，输入下列程序：

```
.text
addi $r0, $r0, 1    ; 第 1 条指令
addi $r1, $r1, 2    ; 第 2 条指令
add $r2, $r0, $r0   ; 第 3 条指令
sub $r3, $r1, $r2   ; 第 4 条指令
sllv $r0, $r0, $r3  ; 第 5 条指令
srlv $r3, $r1, $r0  ; 第 6 条指令
```

```

or $r1, $r0, $r0 ; 第 7 条指令
add $r1, $r1, $r0 ; 第 8 条指令
sllv $r3, $r2, $r1 ; 第 9 条指令

```

2. 分析程序，将指令执行结果采用注释的方式记录在每条指令行末，以便与仿真结果对比。
3. 运行批处理文件，生成指令机器码文件。
4. 按下快捷键“F5”启动仿真。
5. 按下键盘按键“R”进行复位。
6. 按下键盘按键“C”，单步运行程序，观察流水线处理器显示器，分别记录当第 3、4 条指令处于执行阶段时，ALU 的输出结果。当它们分别处于写回阶段时，记录目的寄存器 r2、r3 的值。
7. 将 r2、r3 的值与正确的结果相比较，可以看到 r2、r3 写入了错误的结果，也就是说第 3、4 条指令计算出了错误的结果（ALU 输出），并写入了各自的目的寄存器中。

分析：前 5 条指令执行过程如图 4-3 所示，

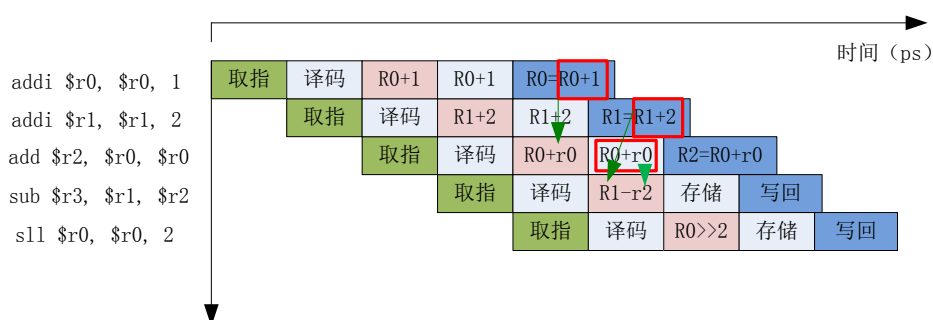


图 4-3：前 5 条指令执行过程示意图

第 3 条指令功能是  $r2 = r0 + r0$ ，它依赖于第 1 条指令  $r0 = r0 + 1$  执行后的结果。当第 3 条指令利用  $r0$  的值进行运算时，第 1 条指令才将结果写入  $r0$  寄存器，从而导致第 3 条指令计算错误的结果。同理，可以分析第 4 条指令在计算  $r1 - r2$  时， $r1$  才写回，而  $r2$  还没写回，也导致指令计算错误的结果。这就是写后读（RAW）数据冲突。

通过数据重定向的方式可以解决此类冲突。从图 3 中可以看到，第 3 条指令的执行依赖于第 1 条指令的执行结果。第 3 条指令执行时，第 1 条指令处于写回阶段，那么就可以将写回阶段中的结果  $r0+1$  重定向到正在进行计算的第 3 条指令中替换  $r0$  进行计算。同理，我们可以将第 2、3 条指令写回和存储阶段的结果重定向到第 4 条指令，分别替换  $r1$  和  $r2$  进行运算。如图 3 中的绿色箭头所示。

在此，我们可以做个形象的比喻。假设流水线的五个阶段分别叫做 1、2、3、4、5 车间，那么执行阶段就是 3 号车间，而存储阶段是 4 号车间，写回阶段是 5 号车间。当 3 号车间需要使用 4 号或者 5 号车间的产品时，直接从 4、5 号车间“拷贝”一份来提前使用，而不是等到 4、5 号车间将产品送入仓库后再到仓库去取。也可以说 4、5 号车间的产品重定向到了 3 号车间。

以上内容介绍了流水线中的写后读（RAW）数据冲突。按下列实验步骤进行实验，了解流水线中的 1w 指令数据冲突。

1. 打开项目下的程序源文件“rom.asm”，输入下列程序：

```

.text
addi $r0, $r0, 1 ; 第 1 条指令
sw $r0, 0($r1) ; 第 2 条指令
lw $r2, 0($r1) ; 第 3 条指令
add $r3, $r0, $r2 ; 第 4 条指令

```



```
sllv $r0, $r0, $r3 ; 第 5 条指令
srlv $r3, $r1, $r0 ; 第 6 条指令
or $r1, $r0, $r0 ; 第 7 条指令
add $r1, $r1, $r0 ; 第 8 条指令
sllv $r3, $r2, $r1 ; 第 9 条指令
```

2. 分析程序，将指令执行结果采用注释的方式记录在每条指令行末，以便与仿真结果对比。
3. 运行批处理文件，生成指令机器码文件。
4. 按下快捷键“F5”启动仿真。
5. 按下键盘按键“R”进行复位。
6. 按下键盘按键“C”，单步运行程序，观察流水线处理器显示器，记录第 4 条指令处于执行阶段时，ALU 的输出结果，处于写回阶段时目的寄存器 r3 的值。
7. 将 r3 的值与正确的结果相比较，可以看到第 4 条指令计算出了错误的结果（ALU 输出），并写入了目的寄存器 r3 中。

分析：前 5 条指令执行过程如图 4-4 所示，

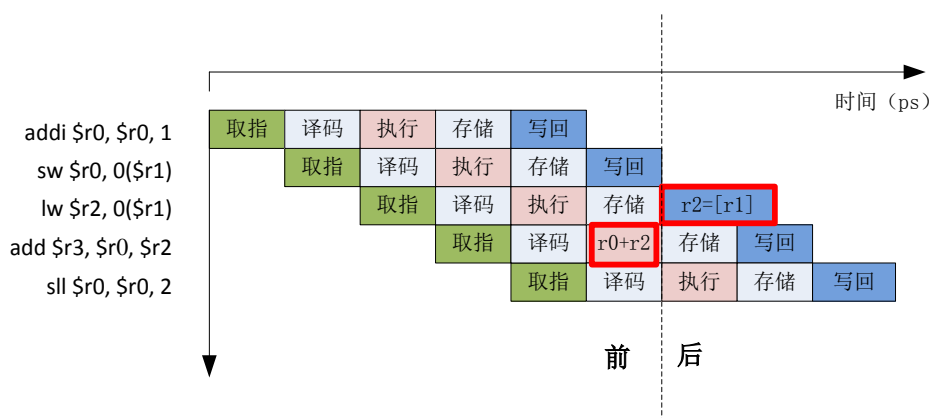


图 4-4：前 5 条指令执行过程示意图

第 4 条指令功能是  $r3 = r0 + r2$ ，它依赖于第 3 条指令  $r2 = [r1+0]$  执行后的结果。由于  $r2 = [r1+0]$  滞后于  $r3 = r0 + r2$  一个周期，所以不能使用数据重定向的方法，这就是 `lw` 指令导致的数据冲突，使得依赖于 `lw` 指令的下一条指令计算了错误的结果。

该如何解决 `lw` 指令导致的数据冲突呢？如果将第 4 条指令的执行阶段延长一个周期或者将第 4 条指令阻塞在译码阶段，当 `lw` 指令进入存储阶段时，解除阻塞，当 `lw` 指令进入写回阶段时，第 4 条指令进入执行阶段。此时就可以将 `lw` 指令从数据存储器读出的结果从写回阶段重定向到第 4 条指令的执行阶段，从而使得第 4 条指令使用正确的操作数计算，得出正确的结果。注意，第 4 条指令阻塞在译码阶段时，后续取指操作也将被阻塞。

以上内容介绍了 `lw` 指令数据冲突。按下列实验步骤进行实验，了解流水线中的控制冲突。

1. 新建一个 MIPS 五级理想流水线微处理器项目。
2. 打开项目下的程序源文件“rom.asm”，输入下列程序：

```
.text
addi $r0, $r0, 1 ;第 1 条指令
beq $r0, $r0, BX ; 第 2 条指令, r0 == r0, 程序跳转到 BX 处执行
add $r1, $r1, $r0 ; 第 3 条指令
add $r3, $r0, $r2 ; 第 4 条指令
```

BX:

```
sll $r0, $r0, 2    ; 第 5 条指令
srl $r3, $r1, 1    ; 第 6 条指令
or $r1, $r0, $r0   ; 第 7 条指令
add $r1, $r1, $r0   ; 第 8 条指令
sllv $r3, $r2, $r1 ; 第 9 条指令
```

3. 运行批处理文件，生成指令机器码文件。
4. 按下快捷键“F5”启动仿真。
5. 按下键盘按键“R”进行复位。
6. 按下键盘按键“C”，单步运行程序，观察流水线处理器显示器，可以看到当第 2 条指令处于最后一个流水线阶段时，程序才实现了跳转。此时第 2 条指令之后的指令分别处于不同的流水线阶段，这些本不应该执行的指令也进入了流水线并最终执行了。这就是分支指令导致的控制冲突。

理想流水线 MIPS 处理器中，分支指令导致的控制冲突使得微处理器执行了本该跳过的指令。

## 2.2 任务（二）：MIPS 理想五级流水线处理器指令扩展

### 实验要求

在流水线微处理器上实现 R 类型指令：xor、nor、sll、srl。

### 测试方法

电路修改完成后，请自行设计测试程序，通过仿真，检验扩展指令执行是否正确。

### 提示

在实现新增指令功能的时候，可将指令实现过程分步，并划分到各个流水线阶段，明确每个流水线阶段完成指令的哪一个步骤。这样就可以确定流水线处理器电路的每个流水线阶段应该做什么样的扩展或改进。每个阶段分工做好了，那么指令也就可以正确执行了。

**注意**，由于条件分支指令 bltz、跳转指令 jal、jr 会导致控制冲突，无需在理想流水线处理器上实现它们，留待在解决控制冲突后的流水线处理器上扩展实现。

## 2.3 任务（三）利用数据重定向改进流水线处理器

### 实验要求

在任务（二）的基础上改进流水线处理器，利用数据重定向解决写后读（RAW）数据冲突。

### 测试方法

电路改造完成后，运行测试程序，检验是否解决了流水线中的写后读数据冲突。测试程序如下：

```
.text
addi $r0, $r0, 1    ; 第 1 条指令
addi $r1, $r1, 2    ; 第 2 条指令
add $r2, $r0, $r0   ; 第 3 条指令
sub $r3, $r1, $r2   ; 第 4 条指令
sll $r0, $r0, 2     ; 第 5 条指令
srl $r3, $r1, 1     ; 第 6 条指令
or $r1, $r0, $r0    ; 第 7 条指令
```

```
add $r1, $r1, $r0 ; 第 8 条指令
```

```
sllv $r3, $r2, $r1 ; 第 9 条指令
```

若成功解决了流水线中存在的写后读数据冲突，则以上每一条指令的执行结果都是正确的。

#### 提示

1. 当执行阶段中的指令有一个与存储阶段或写回阶段的目的寄存器相同的源寄存器时，需要重定向。
2. 增加冲突检测单元和两个重定向复用器。冲突检测单元接收在执行阶段中指令的两个源寄存器，以及在存储阶段和写回阶段中指令的目的寄存器，它还从存储阶段和写回阶段接收 RegWrite 信号以便明确是否要写目的寄存器。冲突检测单元为重定向复用器计算控制信号以便确定选择来自寄存器文件的操作数，还是来自存储阶段或写回阶段的结果。

## 2.4 任务（四）利用阻塞解决 lw 指令数据冲突

### 实验要求

在任务三的基础上，进一步改进处理器，利用阻塞方式解决 lw 指令数据冲突。

### 测试方法

电路改造完成后，运行测试程序，检验是否解决了流水线中的写后读数据冲突。测试程序如下：

```
.text
```

```
addi $r0, $r0, 1 ; 第 1 条指令
```

```
sw $r0, 0($r1) ; 第 2 条指令
```

```
lw $r2, 0($r1) ; 第 3 条指令
```

```
add $r3, $r0, $r2 ; 第 4 条指令
```

```
sll $r0, $r0, 2 ; 第 5 条指令
```

```
srl $r3, $r1, 1 ; 第 6 条指令
```

```
or $r1, $r0, $r0 ; 第 7 条指令
```

```
add $r1, $r1, $r0 ; 第 8 条指令
```

```
sllv $r3, $r2, $r1 ; 第 9 条指令
```

若成功解决了流水线中存在的写后读数据冲突，则以上每一条指令的执行结果都是正确的。

#### 提示

1. 当译码阶段中的指令有一个与执行阶段 lw 指令的目的寄存器相同的源寄存器时，需要将译码阶段的指令阻塞，同时取指阶段也被阻塞。直到 lw 指令进入存储阶段以后，才取消阻塞。
2. 增加冲突检测单元和重定向复用器。冲突检测单元接收译码和执行阶段中指令的两个源寄存器，以及执行阶段的 MemtoRegE 信号，该信号明确执行阶段的指令是否是 lw 指令，同时接收存储阶段和写回阶段中指令的目的寄存器，还从存储阶段和写回阶段接收 RegWrite 信号以便明确是否要写目的寄存器。冲突检测单元为重定向复用器计算控制信号以便确定选择来自寄存器文件的操作数，还是来自存储阶段或写回阶段的结果，并提供阻塞信号用来控制译码和取指阶段是否阻塞。

## 2.5 任务（五）解决所有冲突的流水线处理器

### 实验要求

在任务四的基础上，进一步改造处理器，解决控制冲突，实现分支指令（包含条件转移指令和无条件跳转指令）功能。解决控制冲突以后，就可以得到解决所有冲突的五级流水线 mips 微处理器。

### 测试方法

电路改造完成后，运行测试程序，检验是否解决了流水线中的写后读数据冲突。测试程序如下：

```
.text
addi $r0, $r0, 1      ; 第 1 条指令
beq $r0, $r0, BX      ; 第 2 条指令, r0 == r0, 程序跳转到 BX 处执行
add $r1, $r1, $r0      ; 第 3 条指令
add $r3, $r0, $r2      ; 第 4 条指令
```

BX:

```
sll $r0, $r0, 2        ; 第 5 条指令
srl $r3, $r1, 1        ; 第 6 条指令
or $r1, $r0, $r0        ; 第 7 条指令
add $r1, $r1, $r0        ; 第 8 条指令
sllv $r3, $r2, $r1      ; 第 9 条指令
```

若成功解决了流水线中存在的控制冲突，则以上每一条指令的执行结果都是正确的。

### 提示

1. 处理控制冲突的一种机制是阻塞流水线直到确定分支是否发生为止。因为确定分支是在存储阶段完成的（执行阶段需要计算分支条件），所以流水线将在每个分支阻塞 3 个周期。这将严重降低系统的性能。
2. 如果能尽早的确定是否发生分支，则可以减少流水线阻塞周期数。确定分支只需要比较两个寄存器是否相等。使用一个专门的比较器比执行减法和 0 检测快得多。如果比较器足够快，可以将其放到译码阶段，这样从寄存器文件中读出操作数并比较，就可以在译码阶段结束时确定下一个 PC。
3. 在译码阶段增加一个比较器，提前判断条件分支指令转移条件是否成立。同时，需要在译码阶段计算出转移地址，以便确认转移后将转移地址设为 PC。此外，条件分支指令处于译码阶段时，若分支条件成立，那么需要跳过进入取指阶段的指令，可以通过取指寄存器使能控制，阻止其进入译码阶段。在下个时钟上升沿到来时，将转移指令地址置入 PC，而分支指令后的那条指令，由于取指寄存器被禁止，所以它并没有被取出并进入译码阶段，相当于被流水线剔除。
4. 需要注意的是，提前确定分支会产生新的写后读（RAW）数据冲突。特别是，如果分支指令的一个源操作寄存器由前一条指令计算得到且还没有写入寄存器，分支指令将从寄存器中读取错误的操作数值。我们可以采用前面使用的方法来解决数据冲突（如果数据有效则进行重定向，如果数据还没有准备好，那么阻塞流水线直至数据准备好，可以重定向为止）。

## 第四部分 微机原理与接口技术实验

微机原理与接口技术是一门工科大学生必修的专业基础课，该课程需要学生掌握汇编语言编程、微机原理以及接口技术三大部分内容。本部分实验将汇编语言、微机原理以及接口芯片融为一体，通过实验，使读者全方位地掌握相关的知识。

### ◆ 汇编语言编程

实验中使用的是 8086 汇编指令，通过实验，可提升读者接口芯片编程以及微机系统编程能力。此外，通过将编译后的机器码文件加载到存储器中，读者对指令的编译过程以及机器语言等概念将会有更加深刻的认识。

### ◆ 微机原理

实验中使用的 CPU 是 8086 微处理器，通过实验，读者不仅能从指令和微指令的角度理解 8086 微处理的工作原理，还能从电路的角度了解 8086 微处理器执行指令的具体过程。

实验中，读者能进入 8086 的内部电路，能看到每一条指令的机器码，能看到每一条指令对应的微程序，能看到每一条微指令机器码。整个 8086 微处理器就是一个透明的微型机，读者可以从三个方面跟踪指令的执行过程：从指令层面，即程序计数器 PC 是如何顺序取指和实现跳转的；从微指令层面，即每一条指令对应一段微程序，每一段微程序包含若干条微指令，这些微指令是如何一步步协同配合，完成指令功能的；从微命令层面，每一条微指令包含几十个控制信号，这些控制信号密切配合，打通数据与地址信息在各个模块与寄存器之间的数据通路，从而完成指令规定的一个任务，简称微操作。当一条指令对应的所有微指令顺序执行结束，那么指令的使命也就达成了。

理解微机原理的关键是理解 8086 汇编指令的数据通路，而该部分实验将所有的数据通路展现在读者面前，读者必能深刻理解微机原理。

### ◆ 接口技术

本部分实验提供了各种各样的接口芯片，既有并行接口芯片 8255，又有串行通信接口芯片 8250；有用于定时计数的芯片 8253，也有用于中断控制的芯片 8259A；还有 DMA 控制器芯片 8237A 以及用于模数转换的 A/D 和 D/A。

所有芯片都有完整的内部电路原理图，每一种接口芯片对应一个实验，读者不仅可以通过芯片管脚了解芯片的功能，还能通过芯片内部电路分析芯片管脚为什么有这样的功能。

除了从各个芯片的内部电路来了解芯片功能外，读者还需要学会使用芯片，将芯片与 CPU 建立电路连接，构成一个微机系统，然后通过编程来控制和使用芯片功能。

具体到每个实验的安排上，我们遵循渐进式的学习原则，先引导读者对单个芯片电路进行功能测试，以熟悉芯片的管脚功能以及内部的主要功能模块。然后，引导读者完成芯片与 CPU 的电路连接以及测试程序的编写，通过仿真，向读者直观的呈现芯片与 CPU 之间的关系以及工作过程。在实验的最后，一般会要求读者使用芯片以及 CPU 搭建电路，通过编程，完成指定的功能或任务，以此来提升读者的综合能力。

如果将 CPU 比作人的大脑，那么外围的接口芯片就是 CPU 的眼、耳、鼻、手等器官，CPU 通过接口芯片获取外部信息，然后进行处理。CPU 根据处理结果给特定的接口部件发送命令，控制其完成指定的操作，或者将处理的结果通过接口芯片传送到外部。

# 实验 1 实验环境的使用

实验性质：验证

建议学时：2 学时

## 一、实验目的

- 掌握 Dream Logic 软件的基本使用方法。
- 通过编写、编译、加载以及仿真运行一段程序，熟悉 8086 微机系统的工作过程。并在程序运行的过程中，结合相关电路的状态，深入理解指令取指、译码、执行的原理。

## 二、预备知识

### 2.1 8086 微机系统的组成

8086 微机系统的组成如下图 1-1 所示：

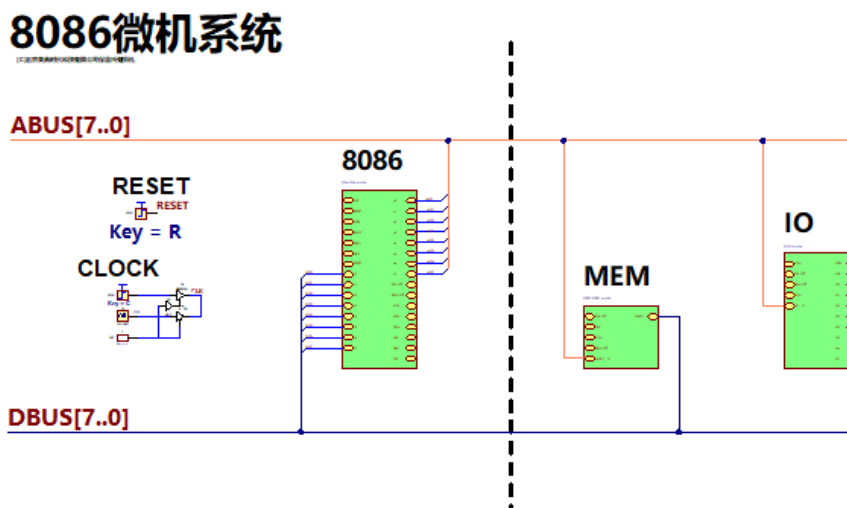


图 1-1：8086 微机系统

### 2.2 使用模块化的方法设计数字系统

在设计复杂电路系统的时候，往往需要分析系统的功能，然后将整个系统按功能划分成不同的子模块，然后分别设计各个子模块，子模块还可以按同样的方法进一步划分为更小的功能模块，这样子模块功能单一，便于设计和验证，然后将设计好的各个子模块按从小到大的顺序组装成一个完整的系统。这就是自顶向下的功能划分以及自底向上的功能实现相结合的设计方法。

8086 模型机以及后续的各种接口芯片都是使用这种方法设计的。该方法类似于面向对象的程序设计中的重要原则：封装和松耦合。

8086 微机系统的主要模块为：

1. 8086 微处理器（CPU）：8 位处理器，支持 8086 汇编指令。
2. MEM 主存储器：存储指令和数据，存储空间为 256 字节，最大访问地址为 0xff。
3. I/O 接口地址译码器：对 8 位接口地址译码，输出接口芯片片选信号，CPU 对选中的芯片进行读/写访问。

## 三、实验内容

请读者按照下面的步骤完成实验内容，着重理解 8086 微机系统运行汇编指令的基本过程。并掌握 Dream

Logic 软件和 CodeCode.net 平台的基本使用方法, 为完成后续实验做好准备。如果读者在学习本次实验之前已经系统学习过本书第二部分的实验内容, 在学习本实验的内容时就会相对容易。否则, 强烈建议读者首先按照本书第二部分中的实验一, 详细了解 Dream Logic 是如何设计和仿真一个处理器的。

### 3.1 从 CodeCode.net 平台领取任务

CodeCode.net 平台是用于教师在线布置实验任务, 并统一管理学生提交的作业的平台。如果没有使用到 CodeCode 平台, 可以跳过此部分, 直接从 3.2 节继续进行实验。

1. 读者通过浏览器访问<https://www.codecode.net>, 可以打开 CodeCode.net 平台的登录页面。
2. 在登录页面中, 读者输入帐号和密码, 点击“登录”按钮, 可以登录 CodeCode.net 平台。
3. 登录成功后, 在“课程”列表页面中, 可以找到微机原理与接口技术相关的实验课程。点击此课程的链接, 可以进入该课程的详细信息页面。
4. 在课程的详细信息页面中, 可以查看课程描述信息, 该信息对于完成实验十分重要, 建议读者认真阅读。点击左侧导航中的“任务”链接, 可以打开任务列表页面。
5. 在任务列表中找到本次实验对应的任务, 点击右侧的“领取任务”按钮, 可以进入“领取任务”页面。
6. 在“领取任务”页面中, 查看任务信息后, 填写“新建项目名称”和“新建项目路径”, 然后选择“项目所在的群组”, 点击“领取任务”按钮后, 可以创建个人项目用于完成本次实验, 并自动跳转到该项目所在的页面。

**提示:** 在“领取任务”页面中, “新建项目名称”和“新建项目路径”这两项的内容可以使用默认值, 如果选择的群组中已经包含了名称或者路径相同的项目, 就会导致领取任务失败, 此时需要修改新建项目名称和新建项目路径的内容。

7. 在个人项目页面中, 包括左侧的导航栏、项目信息、文件列表、readme.md 文件内容等, 如图 1-2 所示。
8. 在个人项目页面的图 1-2 中, 点击红色方框中的按钮, 可以复制当前项目的 URL, 在本实验后面的练习中会使用这个 URL 将此项目克隆到读者计算机的本地磁盘中, 从而供读者进行修改。

**提示:** 领取任务的本质是: 教师在新建任务时填写了实验模板的 URL, 然后读者在领取任务时, 根据任务中记录的 URL 来派生 (Fork) 出一个新的项目, 供读者在其中进行修改。

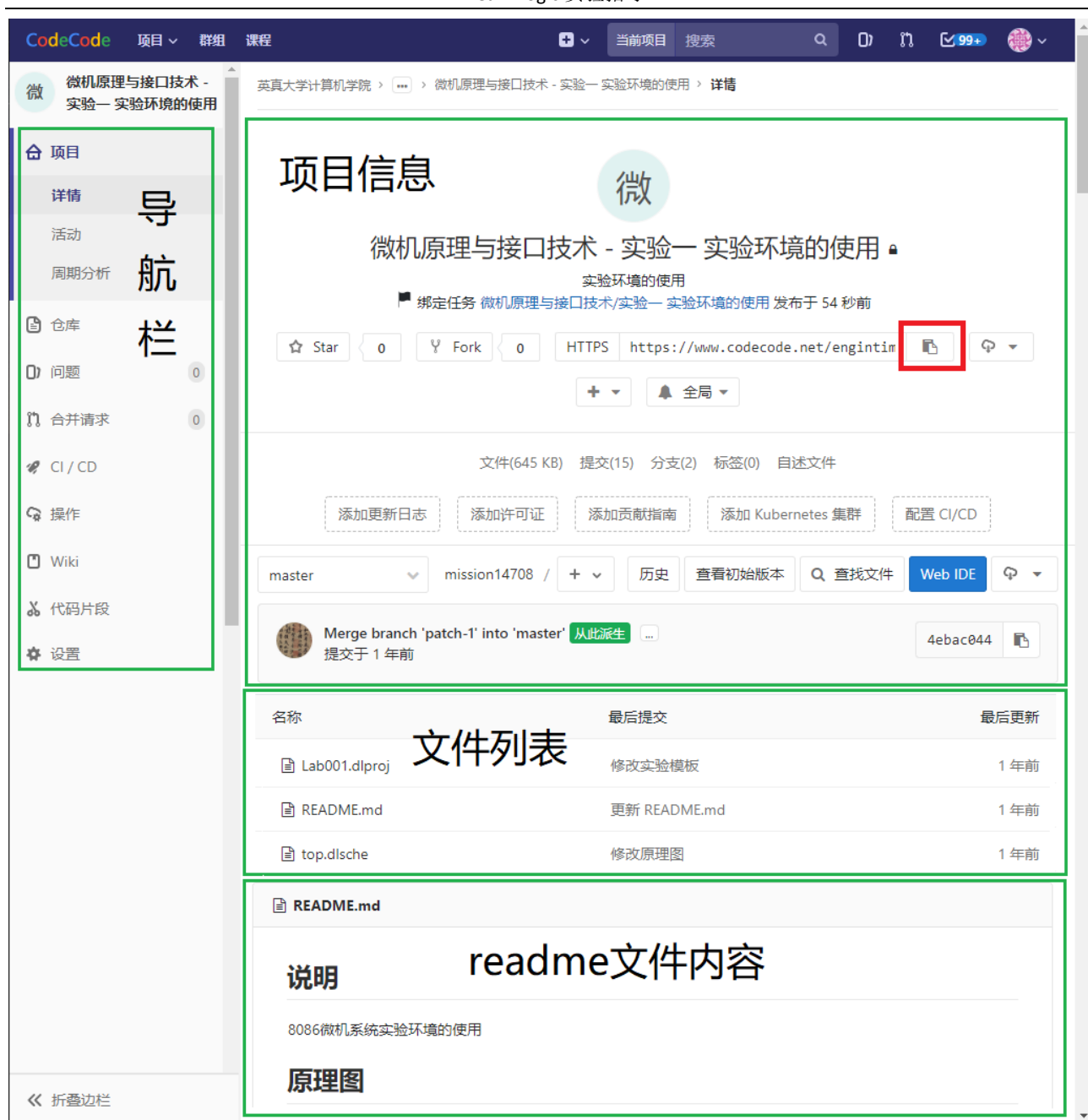


图 1-2：个人项目页面

### 3.2 启动 Dream Logic

在安装有 Dream Logic 的计算机上，可以使用两种方法来启动 Dream Logic：

- 在桌面上双击“Engintime Dream Logic”图标。
- 或者
- 点击“开始”菜单，在“所有程序”中的“Engintime Dream Logic”中选择“Engintime Dream Logic”即可启动程序。

启动 Dream Logic 后会首先打开“登录”窗口，可以使用两种方法完成登录。

- 如果读者在 CodeCode.net 平台上注册了帐号，在连接互联网的情况下，可以使用 CodeCode.net 平台中已注册的用户名和密码进行登录。
- 如果读者还没有 CodeCode.net 平台上的帐号，可以点击左下角“从加密锁获取授权”按钮，获取授权之后，完成登录。

### 3.3 将 CodeCode.net 平台上的项目克隆到本地



如果读者从 CodeCode.net 平台领取了任务,可以在领取任务后将读者的个人项目克隆到本地磁盘中;如果读者没有 CodeCode.net 平台账号无法领取任务,可以直接将实验模板项目克隆到本地磁盘中。

#### 将领取任务新建的个人项目克隆到本地

1. 选择 Dream Logic 菜单“文件->新建->从 Git 远程库新建项目”,打开“从 Git 远程库新建项目”对话框。
2. 在“Git 远程库 URL(G)”中填入读者个人项目的 Git 远程库的 URL 地址(在图 1-2 中,点击红色方框中的按钮,可以复制当前项目的 URL)。
3. “项目文件夹名称”填入“8086”。
4. “项目位置”选择新建项目需要保存到的磁盘目录。
5. 点击“确定”按钮后会弹出一个 Windows 控制台窗口,并开始运行 Git 克隆命令将 Git 远程库克隆到本地,一定要等克隆成功后再关闭该窗口。
6. 克隆项目成功后,在对话框中选择“克隆成功,打开新建项目”就可以打开新建的项目。

#### 直接将实验模板项目克隆到本地

由于 CodeCode.net 平台上提供的实验模板是开放的,所有的人都可以访问。所以, Dream Logic 可以直接将 DM1000 八位模型机实验模板克隆到本地磁盘。步骤如下:

1. 选择 Dream Logic 菜单“文件->新建->从 Git 远程库新建项目”,打开“从 Git 远程库新建项目”对话框。
2. 在“Git 远程库 URL(G)”中项目模板 Git 远程库的 URL 地址:  
<https://www.codecode.net/engintime/Dream-Logic/Project-Template/Microcomputer/Lab001.git>
3. “项目文件夹名称”填入“8086”。
4. “项目位置”选择新建项目需要保存到的磁盘目录。
5. 点击“确定”按钮后会弹出一个 Windows 控制台窗口,并开始运行 Git 克隆命令将 Git 远程库克隆到本地,一定要等克隆成功后再关闭该窗口。
6. 克隆项目成功后,在对话框中选择“克隆成功,打开新建项目”就可以打开新建的项目。

#### 注意:

1. 为了能正常使用从 Git 远程库新建项目功能,用户需要在本地安装 Git 客户端程序,并且在安装 Git 的过程中会有一个步骤询问是否设置 Windows 环境变量,此时一定要设置上 Windows 环境变量。
2. 文件路径中不允许包含中文,请读者在使用之初就养成使用英文命名项目或文件的习惯。

### 3.4 8086 微机系统项目

请读者通过以下步骤熟悉 8086 微机系统项目下的电路模块原理图以及其它的相关文件。

1. 打开新建的实验项目,在左侧的项目管理器中观察项目中包含的所有文件夹和文件。并根据表 1-1 了解每个文件的基本情况。

文件夹或文件名称	说明
8086	8086 文件夹包含下列文件,实现了一个简单的 8086 处理器。注意,实际的 Intel 8086 处理器是 16 位处理器,并且提供 20 位地址线。这里提供的是一个简化版本的 8 位 8086 处理器,数据线和地址线都是 8 位的。
8086.pdf	Intel 8086 官方手册。
ALU.dlsche	运算单元子模块原理图文件
PC.dlsche	程序计数器子模块原理图文件
REGS.dlsche	寄存器子模块原理图文件
CU.dlsche	控制单元子模块原理图文件
FLAG.dlsche	标志寄存器子模块原理图文件

uPC_NEXT.dlsche	微程序计数器子模块原理图文件
BRANCH.dlsche	程序跳转控制子模块原理图文件
8086.dlsche	8086 微处理器的顶层原理图文件
rom.masm	此文件中包含了 8086 的全部微指令源程序。
rom.bat	这是一个 Windows 批处理文件,运行此文件可以执行其中的命令让 microasm.exe 编译器将 rom.masm 中的微指令源程序编译为机器码,并写入存储器映射文件 rom.rxm 中,同时还会生成列表文件 rom.lst 和调试信息文件 rom.dbg。
rom.rxm	微指令存储器映射文件,其中包含了微指令源程序生成的机器码。当 8086 开始运行时,此文件中的内容会被加载到 CU 子模块的 ROM 中,从而允许 8086 将指令的机器码翻译为微指令的机器码并执行。
I/O	文件夹
I/O.dlsche	I/O 接口地址译码子模块原理图文件
MEM	文件夹
MEM.dlsche	主存储器子模块原理图文件
ram.asm	此文件中默认包含了一个简单的汇编语言源程序,作为示例程序提供给读者,读者可以修改此文件中的源代码,根据需要编写自己的汇编程序。在启动仿真后,8086 处理器会运行此文件中的汇编程序。
ram.bat	这是一个 Windows 批处理文件,运行此文件可以执行其中的命令让 dmasm.exe 编译器将 ram.asm 中的汇编代码编译为机器码,并写入存储器映射文件 ram.rxm 中,同时还会生成列表文件 ram.lst 和调试信息文件 ram.dbg。
ram.rxm	主存储器映射文件,其中包含了汇编程序生成的机器码。当 8086 开始运行时,此文件中的内容会被加载到 MEM 子模块的 RAM 中,从而允许 8086 执行其中的机器码。
top.dlsche	8086 微机系统的总图。使用地址总线 and 数据总线将 8086 处理器与存储器模块和 I/O 译码模块连接在一起。

表 1-1: 8086 微机系统项目中的文件

除了以上可以从项目管理器中看到的文件之外,在项目文件夹中还有一些其它的文件需要进行说明。读者可以在项目管理器窗口中右键点击项目节点(根节点),选择右键菜单中的“打开所在的文件夹”,Dream Logic 会使用 Windows 资源管理器打开项目所在的文件夹,需要说明的文件参见表 1-2。

文件名称	说明
项目名称.dlproj	Dream Logic 使用的项目文件
MEM/dmasm.exe	8086 汇编程序的编译器。
MEM/dmasm.c	dmasm.exe 程序的 C 源代码。
MEM/ram.dbg	8086 汇编程序的调试信息文件。在使用 dmasm.exe 编译 ram.asm 文件时,会生成此文件。
MEM/ram.lst	8086 汇编程序的列表文件。在使用 dmasm.exe 编译 ram.asm 文件时,会生成此文件。
8086/microasm.exe	8086 微指令程序的编译器。
8086/microasm.c	microasm.exe 程序的 C 源代码。
8086/rom.dbg	8086 微指令程序的调试信息文件。在使用 microasm.exe 编译 rom.masm 文件时,会生成此文件。

8086/rom.lst	8086 微指令程序的列表文件。在使用 microasm.exe 编译 rom.masm 文件时，会生成此文件。
--------------	--

表 1-2: 8086 微机系统项目中的其它文件

### 3.5 8086 微机系统中模块的连接方法以及时钟、复位控制

结合以下说明，熟悉总图 top.dlsche 中的模块连接方式，系统时钟以及系统复位控制。

1. 打开原理图 top.dlsche。
2. 原理图中通过总线来连接各个模块，突出各个模块之间的地址和数据通路。
3. 原理图中模块的接口，遵守左面输入，右面输出的约定，模块顶部或底部的总线为双向总线。
4. 一般情况下，名称相同的模块接口相连，从而不必通过网络连接，减少网络连线，使总图更加简洁。
5. 原理图中的单步时钟使用键盘上的 C 键控制，复位信号使用键盘上的 R 键控制。
6. 单步时钟与自动时钟可以切换，以便对程序执行过程进行控制。
7. 自动时钟模式下，微机自动执行程序，不需要手动干预；而单步时钟模式下，则需要手动输入时钟，一次执行一条微指令，没有时钟上升沿输入的话，微机系统停止取指执行，这时就可以详细查看系统中各个寄存器的值、数据总线以及地址总线的状态、主存储器的内容变化情况。通过这些信息，就可以分析每一条微指令的执行情况，便于从电路角度理解指令的执行原理。

**提示：**用鼠标左键双击总图中的模块，就会自动跳转到对应的子模块原理图文件，在子模块原理图文件中的空白处双击鼠标左键，就会返回到总图中，这样操作起来十分方便。

**提示：**如果读者是第一次使用 Dream Logic，请务必在本书第一部分的实验一中找到原理图常用操作的介绍内容，并自行练习，为后面的实验内容做好准备。

### 3.6 练习使用 8086 微机系统运行一个简单的汇编程序

接下来会引导读者练习使用 8086 运行一个简单的汇编程序，为后续的实验打下一个坚实的基础。

#### 8086 汇编指令源程序

8086 微机系统使用的是 8086 汇编指令集，读者可以使用熟悉的指令快速编写程序。

1. 打开项目下的汇编源程序文件 MEM/ram.asm，结合注释熟悉代码。

#### 汇编源程序的编译

2. 在项目管理器中右键点击 MEM/ram.bat 文件，在弹出的菜单中选择“运行批处理文件”，即可启动编译器 dmasm.exe，对默认的 ram.asm 源程序文件进行编译。
3. 若源程序编译成功，则会生成指令机器码文件 ram.rxm，列表文件 ram.lst，以及调试信息文件 ram.dbg。
4. 若代码有误，则会报告出错的代码行，以及错误类型，方便用户修改源代码。请读者将 ram.asm 中的指令“moval, 16”替换为“movall, 16”，保存文件后，运行批处理文件 ram.bat，观察编译窗口中的错误提示，根据错误提示修改代码，直到编译成功为止。

#### 指令机器码文件加载到存储器

编译生成的指令机器码文件 ram.rxm 需要加载到主存储器 RAM 中才能被执行。

5. 打开原理图文件 top.dlsche，双击 MEM 存储模块，选中存储器 RAM，在右侧属性栏中的“存储器映射文件”默认添加了 ram.rxm 文件。
6. 在 RAM 属性中的“调试”节点下，读者可以看到“调试信息文件”属性和“程序计数器名称”属性已经默认设置为 ram.dbg 和 IP。当 8086 启动仿真时，会使用 ram.dbg 文件提供的调试信息和 IP 寄存器的值，在“源代码”窗口中为读者实时显示 8086 将要执行的指令。

## 微指令程序

7. 打开项目下的微指令源程序文件 8086/rom.masm, 可以看到, 每一种汇编指令对应一段微程序。这是因为指令执行最终转换为执行指令对应的微程序。

## 微指令程序加载

8. 打开原理图 top.dlsche, 双击 8086 模块打开, 双击 CU 模块打开, 选择 ROM 只读存储器, 在右侧属性栏中可看到, rom.rxm 已经默认加载到了 ROM 中。读者可将其与之前观察的 RAM 中的属性进行对照。

在程序运行过程中, ROM 存储器中的微指令只能读出, 不能写入。而主存储器 RAM 既能读取指令或数据, 也能写入数据。

## 运行程序

通过前面的步骤, 我们分别将指令和微指令程序机器码加载到各自的存储器中, 接下来, 读者可以按照下面的步骤对 8086 微机系统进行一次完整仿真, 通过运行一个简单的汇编程序, 从而对其有一个动态的认识:

1. 首先, 确保当前打开的是 top.dlsche 原理图, 然后选择菜单“仿真”中的“启动仿真”(快捷键 F5)。
2. 通过按键 S 设置其数字信号源输出高电平, 选择单步运行指令的方式。
3. 按下 R 键将 PC 的值初始化为 0x10 (RAM 的 0~15 个存储单元是中断向量, 每个中断向量存储一个中断服务程序的入口地址, 因此代码段的起始地址为 16, 也就是 0x10), uPC 寄存器的值初始化为 0, 这就使得 8086 从汇编程序的第一条指令开始执行, 同时也是从微指令的第一条指令开始执行。
4. 选择“视图”菜单中的“寄存器”弹出“寄存器”窗口, 在寄存器窗口中找到 PC 和 uPC 的值, 确认 PC 为 0x10, 而 uPC 为 0, 其中 PC 是 8 位计数器, uPC 是 16 位寄存器。在寄存器窗口中灰色的项表示对应的寄存器没有使能或者计数器禁止计数, 黑色的则表示寄存器使能, 而红色表示寄存器发生了写入操作。
5. 在寄存器窗口中双击 PC 计数器对应的项目, 可以在原理图中迅速定位 PC 计数器的位置, 请读者确认 PC 计数器的 CTEN 管脚为高电平, 表示禁止计数; 同时, PC 计数器的 Q4 管脚输出高电平, 其余管脚输出低电平, 正好是第一条指令地址 0x10。用相同的方法定位 uPC 寄存器, 并观察其各个管脚的值。
6. 选择“视图”菜单中的“存储器”弹出“存储器”窗口, 选择此窗口工具栏下拉列表中的 RAM, 可以查看 RAM 存储器中的数据。由于此存储器的地址线和数据线都是 8 位的, 所以窗口中的每一行表示一个字节, 其中冒号左边的是 8 位地址值, 冒号右边的是 8 位数据值。RAM 存储器加载的数据是汇编程序生成的机器码, 所以其中的数据与 ram.rxm 存储器映射文件中的数据相同, 读者可以选择“视图”菜单中的“项目管理器”弹出“项目管理器”窗口, 然后双击 ram.rxm 文件, 使用二进制编辑器打开此文件, 确认 ram.rxm 文件中的数据与 RAM 中的数据相同。
7. 在“存储器”窗口中具有灰色底色的行, 表示该存储器当前地址线上的值, 以及数据线上的值。读者可以选择“存储器”窗口工具栏上的“定位存储器”按钮, 或者在窗口内单击右键, 在弹出的菜单中点击“定位存储器”在原理图中迅速定位 RAM 存储器, 并确认其地址线上的值为 0x10 (此时, RAM 地址线上的值是由 PC 寄存器提供的), 输出数据线上的值是 0x01。由于此时正在从 RAM 中读取数据, 所以其 WR 管脚为高电平。
8. 请读者按照前面步骤 5 和步骤 6 的方法, 在“存储器”窗口中查看 ROM 存储器的值, 并与微指令

源程序生成的 rom.rxm 文件中的内容进行比较, 确认其一致。需要注意的是, ROM 是微程序存储器, 它存储的是地址和内容均固定的微指令, 只能读出, 不能写入。ROM 地址线上的值是由微程序寄存器 uPC 提供的, ROM 输出的就是一条 32 位的微指令, 微指令中的每一位都有特定的含义, 有些位用于控制 8086 微处理器内部的电路, 而有些位输出到 8086 微处理器外部, 作为外部设备的控制信号。例如 ROM 的 Q30 管脚输出的信号 RD\_READ 就是用于读寄存器的, 属于内部控制信号。而 ROM 的 Q24 管脚输出的信号 MIO 用于区分 CPU 当前访问的是存储器还是输入输出设备, 属于外部控制信号。由于 ROM 的地址线是 12 根, 数据线是 32 根, 所以在“存储器”窗口中, ROM 的一行数据包括 4 个字节。

在仿真过程中通过“寄存器”窗口、“存储器”窗口和原理图窗口可以观察到 8086 在运行过程中的各种状态, 为了方便读者更好的理解 8086 的运行机制, Dream Logic 还提供了“源代码”窗口帮助读者从汇编指令和微指令的角度观察 8086 的运行过程, 从而将汇编指令与处理器的行为更加紧密的联系起来, 使读者可以在仿真的过程中同时从汇编指令和微指令的角度、从系统的角度(寄存器、存储器)、从数字电路的角度(时序逻辑、组合逻辑)加深理解。

请读者按照下面的步骤学习“源代码”窗口的用法:

1. 在 8086 启动仿真后, 默认会在左侧显示两个源代码窗口“源代码 1”和“源代码 2”。其中源代码 1 窗口默认显示 RAM 存储器对应的汇编代码, 也就是汇编程序 ram.asm 编译生成的列表文件 ram.lst 中的内容; 同样的, 源代码 2 窗口默认显示 ROM 存储器对应的微指令源代码, 也就是微指令源程序 rom.masm 编译生成的列表文件 rom.lst 中的内容。
2. 先来看源代码 1 窗口中的内容, 其一行源代码的内容可以分为四个部分:

```
0006    10    01 00 10    mov al, 16
```

- 最左侧的数字 0006, 表示此条指令在源代码文件 ram.asm 中第 6 行;
- 第二部分 10, 表示此条指令的机器码在存储器映射文件中的地址为 16;
- 第三部分 01 00 10, 表示此条指令的机器码, 可以是两字节或三个字节, `mov al, 16` 指令共三个字节, 01 表示操作码 mov, 00 表示目的操作数 al, 10 表示立即数 16;
- 第四部分是此条指令的汇编代码, 此条指令将立即数 16 写入 al 寄存器中。

源代码 1 窗口左侧的蓝色箭头表示 IP 寄存器指向的位置, 由于 8086 刚刚启动仿真时 IP 寄存器的值为 10, 所以蓝色箭头指向偏移为 10 的代码行。蓝色箭头之所以不表示 PC 计数器指向的位置, 是因为一条汇编指令的机器码通常有 2 到 3 个字节, 指令执行过程中 PC 的值会改变, 为了使指令从开始执行到执行完毕之间蓝色箭头总是指向该条指令, 于是使用 IP 寄存器的值, IP 寄存器总是指向汇编指令机器码的第一个字节, 直到指令执行完才会改变。请读者将源代码 1 窗口中指令的机器码与存储器窗口中 RAM 的内存进行比较, 确认其内容一致。

1. 源代码 2 窗口中的格式如下:

```
0011    00    C7 47 FF FF    fetch instruction
```

- 左侧的 0011, 表示微指令 `fetchinstruction` 在微程序源文件 rom.asm 中的第 11 行。
- 第二部分 00, 表示该条微指令在 ROM 存储器中的地址, 由于 8086 刚刚启动仿真时 uPC 寄存器的值为 0, 所以蓝色箭头指向偏移为 0 的微指令。
- 第三部分 C7 47FFFF 是该条微指令的机器码, 共 4 个字节。
- 第四部分 `fetchinstruction`, 表示微指令代码。

`fetch instruction` 是一条取指微指令, 它的功能是将当前 PC 指向的指令从主存储器 RAM 读出到数据总线 DBUS 上, 与此同时, CU 模块通过 DBUS 上的指令码得到指令对应微程序的入口地址, 并将该地址通过三态门输出到 uPC 输入端, 下个时钟上升沿到来时, uPC 加载该地址, 转去执行指定微程序。

取指微指令的具体实现过程如下, 请读者按下列步骤进行确认:

2. 打开 8086 模块内部的模块 PC，可看到 PC 向地址总线的输出三态门 PC\_A\_gate 使能打开，将 PC 计数器的值 0x10 传送到地址总线 ABUS 上。
3. 打开存储器模块 MEM，可看到 PC 输出到 ABUS 上的有效地址作为 RAM 存储器的输入地址，RAM 存储器的读写控制端输入高电平，于是 RAM 将当前地址 0x10 指向存储单元的内容 0x01（指令机器码）读出。主存储器输出门 MEM\_gate 使能打开，读出的指令输出到数据总线 DBUS 上。
4. 打开 8086 内部控制单元 CU，可看到，CU 将数据总线 DBUS 传送来的指令码 0x01 扩展为 0x20，这个 0x20 就是指令“mov al, 16”对应的微程序入口地址。该入口地址通过一个输出门传送到 uPC 输入端，而且此时 uPC 写使能，下个时钟上升沿到来时，0x20 将写入 uPC 中。
5. 现在，读者可以使用单步时钟发送一个上升沿来触发 8086 执行下一条微指令了！方法是：在原理图窗口中按下单步时钟对应的键盘上的 C 键再抬起，该步操作产生的时钟上升沿使得微程序寄存器 uPC 将输入端的地址写入，微指令存储器 ROM 于是输出新的微指令，控制处理器电路执行相应的操作。
6. 观察 8086 在第一个时钟上升沿触发后进入的状态：uPC 寄存器的值变为 0x20，在源代码 2 窗口中蓝色箭头跳转指向偏移为 0x20 处的微指令，从源代码 2 窗口中可以看到，蓝色箭头指向的微指令正是 `mov al, 16` 的第一条微指令 `inc pc`，它的功能是将 PC 加 1，为读出第二个指令字节码准备地址。后续每输入一个时钟上升沿，执行一条微指令，这就是单步运行的含义。

接下来读者就可以通过发送单步时钟的方式依次执行指令 `mov al, 16` 对应的微指令了，结合微指令注释以及取指微指令执行案例，理解各条微指令执行原理。

**注意**，在仿真过程中，将鼠标放置到源代码窗口中的寄存器上，会显示出当前寄存器的值。

**注意**，在仿真过程中，无论是在总图中，还是在子模块中，都可以使用键盘上的 C 按键发送单步时钟。

**注意**，在仿真过程中，无论是双击原理图中的子模块，还是双击子模块原理图的空白处，都是在同一个窗口中变换仿真层级，不会打开模块对应的原理图文件。

### 3.7 简单修改原理图

目前，在 `top.dlsche` 文件中，时钟信号 CLK 和复位信号 RESET 主要是通过网络标签与各个模块中的对应接口进行连接的，请读者使用“绘制”菜单中的“网络”功能，使用网络将 CLK 信号与各个模块中的对应接口连接起来，将 RESET 信号也与各个模块中的对应接口连接起来。

通常，网络标签用于将原理图中距离比较远的、比较分散的多个网络连接在一起，比使用网络画线连接要方便很多。但是，使用网络画线连接比较直观，容易辨认连接关系。所以，在绘制原理图时，应该综合使用网络标签和网络，充分发挥各自的优点，才能绘制出可读性好、便于修改的原理图。

### 3.8 提交作业

如果读者是通过从 CodeCode.net 平台领取任务创建的个人项目，并将个人项目克隆到本地进行实验，实验结束后可以将本地已更改的项目再推送到 CodeCode.net 平台的个人项目中，方便教师通过 CodeCode.net 平台查看读者提交的作业。步骤如下：

4. 在项目管理器窗口中，右键点击项目节点，在弹出的菜单中，选择“Git”中的“推送当前分支到 Git 远程库”菜单项，弹出“推送当前分支到 Git 远程库”的对话框。
5. 在“推送当前分支到 Git 远程库”对话框中，输入本次项目更改的提示信息。
6. 点击“推送”按钮，可以将当前项目推送到 CodeCode.net 平台的个人项目中。

可以按照下面的步骤查看推送后的项目：

7. 使用浏览器访问<https://www.codecode.net>，使用已注册的帐号登录 CodeCode.net 平台。
8. 在“课程”列表中选择对应的实验课程，打开实验课程的详细信息页面，点击左侧导航栏的“任务”链接，打开任务列表页面。

9. 在任务列表页面打开本次实验对应的任务，在任务详情中点击“查看项目详情”按钮，可以跳转到读者的个人项目页面。
10. 在个人项目页面中，可以查看当前项目的全部内容。
11. 在项目左侧的导航栏中点击“仓库”中的“提交”链接，可以打开提交列表页面。
12. 在提交列表页面中，点击最后一次提交，可以查看此次提交发生变更的文件。

**技巧：**在 Dream Logic 的项目管理器窗口中，选中项目节点，点击鼠标右键，在弹出的右键菜单中，选择“Git”中的“使用浏览器访问 Git Origin”菜单项，可以自动打开本地项目在 CodeCode.net 平台上对应的个人项目。



## 实验 2 可编程并行接口 8255

**实验性质：**验证+设计

**建议学时：**2 学时

### 一、实验目的

- 了解 8255 芯片的内部电路结构以及工作原理。
- 掌握 8255 与 8086 微机系统的连接方法。
- 掌握 8255 芯片编程。
- 结合微指令，从电路角度分析接口访问指令 IN 和 OUT 的实现原理。

### 二、实验内容

请读者按照下面方法之一在本地创建一个项目，用于完成本次实验：

#### 方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建个人项目，然后使用 Dream Logic 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

#### 方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，就需要使用 Dream Logic 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的 URL 为

<https://www.codecode.net/engintime/Dream-Logic/Project-Template/Microcomputer/Lab002.git>

#### 2.1 任务（一）

请读者按下列步骤熟悉 8255 芯片管脚功能和内部电路。

1. 打开项目下的原理图文件 8255/8255.dlsche。
2. 在 8255.dlsche 中找出下列各引脚对应的端口。
  - PA7~PA0: A 端口数据信号引脚。
  - PB7~PB0: B 端口数据信号引脚。
  - PC7~PC0: C 端口数据信号引脚。
  - D0~D7: 8255 的 8 位数据线，和 CPU 系统总线相连。
  - CS: 片选信号，CS 为低电平时，表示 8255 被选中。通常该信号的控制时通过译码电路的输出端提供。
  - RD: 读信号，低电平有效，与 IOR 相连。当 CPU 执行 IN 输入指令时，该信号有效，将数据信息或状态信息从 8255 读至 CPU。
  - WR: 写信号，低电平有效，与 IOW 相连。当 CPU 执行 OUT 输出指令时，该信号有效，将数据信息或控制字从 CPU 写入 8255。
  - A0, A1: 端口选择信号，用来指明哪一个端口被选中。8255 有三个数据端口和一个控制端口。
  - RESET: 复位信号，高电平有效。当 RESET 有效时，内部所有寄存器均被清零，A、B、C 口被自动设为输入端口。

3. 8255 的内部电路及主要模块。如表 2-1 所示：

模块名称	模块原理图文件	模块功能
WRC	wrc.dlsche	C 端口输出寄存器模块，负责 C 端口输出寄存器的置位、复位和写入。
AC	ac.dlsche	A 组控制部件，控制 A 口和 C 口高 5 位输入/输出。



BC	bc.dlsche	B 组控制部件，控制 B 口和 C 口低 3 位输入/输出。
ACH	ACh.dlsche	C 口高 5 位输入/输出
BCL	BCL.dlsche	C 口低 3 位输入/输出

表 2-1:8255 内部的主要模块

表 2-1 中仅列出了部分主要模块，请读者结合教材上的 8255 芯片功能介绍，理解 8255 内部电路原理。

## 2.2 任务（二）

在了解 8255 内部电路后，下面通过仿真，测试 8255 各个输入/输出管脚的功能，掌握 8255 工作方式的设置方法。

任务要求：设定 8255 的 A 口为方式 0 输入，B 口为方式 1 输出，PC7~PC4 为输出，PC3~PC0 为输入。按下列步骤进行实验。

1. 打开项目下的原理图文件 8255\_test.dlsche，启动仿真。
2. 双击 8255 打开内部电路，可以看到此时控制寄存器 CW 的值无效。
3. 按下按键 R 复位后，可以看到 CW 被初始化为 FF。
4. 双击原理图中的空白处，返回顶层电路。

8255 复位后，设置它的工作方式字 CW，步骤如下：

1. 根据 8255 工作要求可知，应将控制寄存器设置为 10010101，即十六进制数 0x95。选中原理图中的 8 位数字信号源器件，将其属性栏中“数字信号（十六进制）”的值修改为 95。
2. 通过键盘按键 W，将 8255 的输入管脚 WR 设置为低电平（写使能）。
3. 通过键盘按键 A，将 8255 的输入管脚 RD 设置为高电平（读禁止）。
4. 通过键盘按键 S，将 8255 的使能输入管脚 CS 输入设置为低电平（片选有效，选中芯片，允许读/写操作）。
5. 通过键盘按键 0 和 1，设置 A1=A0=1，使能控制寄存器 CW，到此即可对 CW 进行写操作了。

经过以上步骤，我们使得 8255 的控制寄存器可以处于可写入的状态，只需要给 CW 寄存器一个时钟上升沿就能写入数据了，步骤如下：

1. 双击 8255 进入其内部电路，可以看到此时 95H 已经到达 CW 的数据输入端，但是并没有写入 CW 控制寄存器中。此时控制寄存器 CW 的各个输入信号。
  - （1）CLR=1，高电平，异步清零信号无效。
  - （2）PR=0，低电平，异步置 1 信号无效。
  - （3）EN=0，CW 寄存器总是使能，允许写入。此时只需要有一个时钟上升沿就能将输入端的 8 位数字信号传送到输出端。
  - （4）CLK=0，低电平
2. 通过按键 W，将 WR 输入置为高电平，从而给 CW 寄存器的 CLK 端输入一个时钟上升沿，将 95H 写入 CW 寄存器中。
3. 双击空白处返回顶层电路。

通过上面的步骤，我们完成了 8255 控制寄存器 CW 的初始化。不要结束仿真，继续按照下面的步骤进行实验，测试 8255 的片选信号 CS 所起的作用。

1. 选中原理图中的 8 位数字信号源器件，在其属性栏中，将数字信号值修改为“0B”。
2. 通过键盘按键，分别设置 CS=0，WR=0，RD=1。
3. 双击 8255 进入内部电路，可看到控制寄存器 CW 的写使能端 EN 为高电平，禁止写入。
4. 设置 WR=1，给控制寄存器 CW 的时钟输入端 CLK 一个上升沿，CW 的值仍为 95H，说明 CW 没有

写入。

5. 双击返回顶层电路。

通过上述步骤可知，在 CS=1 时，芯片不使能，不能对 8255 的内部寄存器进行写操作。下面，我们通过设置 CS，使 8255 使能，然后再写入数据 0B。步骤如下：

1. 通过键盘按键，分别设置 CS=0（片选，芯片使能），WR=0。
2. 双击 8255 进入芯片内部，可以看到 CW 寄存器的使能端 EN=0，CLK=0。
3. 设置 WR=1，CW 寄存器的时钟输入端 CLK 输入一个上升沿。
4. 可以看到控制寄存器 CW 的输出变为了 0BH。说明新的值写入了控制寄存器中。

请读者结合 8255 芯片的读写控制逻辑电路，分析片选信号 CS 是如何控制寄存器的读/写的。控制逻辑中起主要作用的是一个 2-4 译码器，请查阅教材或相关资料，熟悉 2-4 译码器的功能和使用方法。

5. 结束仿真。不要关闭项目，后面将继续在本项目下开展实验。

通过以上实验，可以总结片选信号的作用：只有芯片的片选信号 CS 有效时，才能对芯片内部的寄存器进行读/写操作。几乎所有的可编程接口芯片都有片选信号输入端 CS，当 CPU 与多个接口芯片相接的时候，CPU 就是通过片选信号唯一指定一个芯片，这样 CPU 就可以对其进行单独的读/写访问了，而不会影响其它的芯片。芯片的片选信号通常是通过 I/O 地址译码器产生的，一个或多个 I/O 地址译码产生一个片选信号，选中唯一一个芯片。之所以会出现多个端口地址译码产生同样的片选信号，是因为一个芯片通常拥有多个端口，为了区分这些端口，往往需要分配不同的端口地址。比如，8255 并行接口芯片拥有端口 A、B、C 以及控制寄存器，它们对应的端口地址最低两位 A1、A0 不同，分别为 00、01、10、11。

一个芯片中究竟有多少个端口，最简单的方法是，芯片中有多少个寄存器可以被 CPU 进行读或者写访问：

- CPU 读接口芯片的端口寄存器的过程：CPU 将所读芯片寄存器的端口地址传送到 I/O 接口地址译码器，译码器产生对应芯片的使能信号，选中芯片，同时也指定了芯片中的唯一一个寄存器，芯片接收到 CPU 发送来的读信号 IOR，打开寄存器与总线之间的多通道三态门，寄存器的数据通过三态门传送到总线上，CPU 就可以从总线上获取数据了。
- CPU 写接口芯片的端口寄存器的过程：CPU 将所写芯片寄存器的端口地址传送到 I/O 接口地址译码器，译码器产生对应芯片的使能信号，选中芯片。而将需要写入寄存器的数据放到数据总线上，当芯片接收到 CPU 发送来的写信号 IOW 时，将数据总线上的数据写入了选中的寄存器中。

从电路图 8255.dlsche 中可以看出，只能将总线上的数据 D0~D7 写入控制寄存器 CW 中，而不能将控制寄存器的数据读出，因为控制寄存器的输出没有通过多通道三态门与 D0~D7 相连。

熟悉控制寄存器的写操作过程后，我们通过以下实验步骤，熟悉 8255 端口 A 的读写操作。我们将端口 A、B 设置为输出端口，将 C 的低四位、C 的高四位设置为输入端口，所有端口均工作在方式 0。C 端口作为输入端口，它的多通道输出三态门被关闭，不会干扰输出总线上的数据。

首先，初始化 8255 方式选择控制字，根据要求可知，方式选择控制字为 10001001，即 0x89。

1. 打开项目下的原理图 8255\_test.dlsche，启动仿真。
2. 在原理图中选中 8 为数字信号源器件 DS，在属性栏中，将数字信号输出值修改为 89。
3. 手动设置 8255 芯片输入端 A0、A1 的电平，由于控制寄存器的端口地址为 11，故将 A1、A0 都设置为高电平。
4. 设置 RD 输入为高电平，禁止读。
5. 设置 CS 输入为低电平，芯片使能。

6. 设置 WR 输入为低电平，写有效。
7. 双击模块 8255，进入内部，可以看到 A 寄存器的时钟端输入为低电平。
8. 将鼠标放在当前仿真原理图 8255 中，按下 WR 写信号对应的键盘按键 W，可以看到，CW 控制寄存器的时钟输入端变为高电平，89H 写入了寄存器中。

接下来，按实验步骤，将十六进制数 45H 写入端口 A 的输出缓冲器中，然后读出。在 8255\_test.dlsche 中添加两个十六进制 7 端数码管（在左侧“型号库”窗口的“数字信号显示”中可以找到此器件），用于显示端口输出的数据。

1. 返回仿真图层 8255\_test.dlsche，选中 8 位数字信号源器件 DS，在属性栏中，将数字信号输出值改为 45。
2. 双击模块 8255，进入内部，图中名称为 A 的边沿触发寄存器就是端口 A 的 8 位数据输出缓冲器，缓冲器输出通过一个 8 通道的三态门与 PA0~PA7 相连。结合高亮化显示的网络，可以看到此时的端口 A 寄存器 8 位输入数字信号已经就绪，且寄存器使能端 $\overline{EN}$ 始终为低电平，使能。只需时钟 CLK 端输入一个上升沿，就能就输入端 D0~D7 的数据写入寄存器中。
3. 鼠标双击图中空白处，返回上层电路图 8255\_test.dlsche 中。
4. 手动设置 8255 芯片输入端 A0、A1 的电平，由于 A 端口地址为 00，故将 A1、A0 都设置为低电平。
5. 设置 RD 输入为高电平，禁止读。
6. 设置 CS 输入为低电平，芯片使能。
7. 设置 WR 输入为低电平，写有效。
8. 双击模块 8255，进入内部，可以看到 A 寄存器的时钟端输入为低电平。
9. 将鼠标放在当前仿真原理图 8255 中，按下 WR 写信号对应的键盘按键 W，可以看到，A 寄存器的时钟输入端变为高电平，45H 写入了 A 寄存器中。
10. 设置 RD 输入为低电平，读出 A 端口数据。通过数码管显示器显示总线上的数据。
11. 将 A1、A0 分别设置为 0、1，此时读出的是端口 B 的数据，观察数码管显示器的值。
12. 鼠标双击图中空白处，返回上层原理图，可以看到 WR 的输入信号为高电平。我们注意到，在子模块仿真视图中，也能手动控制其他仿真图层的器件。需要说明的是，在所有仿真图层都能通过键盘按键控制键值相同的器件。

仿照上述方法，将数据 27H 写入端口 B 的输出缓冲器，然后读出，端口 B 的地址为 01。当 A、B 口都写入有效数据后，通过设置地址 A1、A0 的值，使得数码显示器分别显示 A、B 端口的数据。

### 2.3 任务（三）

通过上一个实验任务，读者已经对 8255 并行接口芯片的内部结构和工作原理有了基本的了解。下面将 8255 接口到 8086 微处理器中，掌握 8086 微处理器通过执行程序实现对 8255 芯片的控制的方法。

通过查询的方法读入端口数据。当端口作为输入端口，工作在方式 1 时，CPU 可以通过查询 C 端口的相关位，确定输入缓冲器是否有数据，从而决定是否读入端口数据。通过本次试验任务，掌握查询传送法。

任务要求：将 8255 的 A 口设定为输出数据，工作方式为方式 0；B 口设定为输入数据，工作方式为方式 1；C 口高四位输入，低四位输出。

通过编写程序，实现如下功能：

- 初始化 8255，设定 8255 的工作方式字。
- 通过 B 端口手动输入 8 位地址（使用 8 位数字信号源），8086 微处理器（CPU）从 B 端口获取地址，然后将地址指定存储单元（MEM）的内容通过 A 端口输出。

- 重复取 B 口地址，从主存储器取数，输出到 A 端口的过程。

按照下面的步骤开始实验：

1. 打开项目下的 MEM\ram.asm 汇编源程序文件，结合注释，理解代码。提示：本代码的功能是，首先，初始化 8255 芯片；然后，CPU 不停地查询端口 C 的低四位中的 PC1，若 PC1 为高电平，则说明 B 端口的输入缓冲器中写入了新的数据，CPU 可以取数据了。CPU 从 B 端口取数据后，将取到的数据作为地址，读出地址单元的内容并输出到 A 端口，通过 A 端口的数码管，显示输出的数据。重复上述过程。
2. 打开 top.dlsche 文件后启动仿真，按下复位按键 R 进行复位。
3. 通过手动控制按键 1，使仿真驱动时钟为自动时钟，程序进入自动执行状态。
4. 当初始化代码执行完成后，将与端口 B 中 PB0~PB7 相连的 8 位数字信号源的数字输入设置为 10。
5. 手动控制，使 8255 的 PC2 端输入一个负脉冲。当 B 端口工作在方式 1 时，PC2 是选通信号输入端，名称为 STB，STB 输入负脉冲将使端口数据写入端口缓冲寄存器中。数据写入缓冲器以后，8255 通过 PC1（IBF，高电平表示输入缓冲器数据满）向 CPU 发出状态信号，供 CPU 查询使用。当 PC2 输入负脉冲后，看以看到 PC1 输出为高电平。
6. 跟踪程序的执行，当程序执行完从 B 端口取数据然后从 A 端口输出数据后，观察 A 端口的输出。
7. 暂停仿真，打开存储器窗口，定位到 B 端口输入地址指定的存储单元，比较该单元的内容是否与 A 端口输出 7 段数码管显示的数据相同。若相同，说明 8086 微处理器（CPU）通过 B 端口输入的地址读出了存储器对应单元的内容并通过 A 端口成功输出了。
8. 继续仿真，将与端口 B 中 PB0~PB7 相连的 8 位数字信号源的数字输入设置为任意值，重复步骤 7~8。
9. 结束仿真。

### 三、思考与练习

1. 试说明并行输入接口芯片 8255 工作于方式 1，CPU 如何以中断方式将输入设备的数据读入？
2. 试说明 8255 的 A 口、B 口和 C 口一般在使用上有什么区别？
3. 试说明 8255 在工作方式 2 时如何进行数据输入和输出操作？
4. 如果需要 8255 的 PC3 输出连续方波，如何用 C 口的置位与复位控制命令字编程实现它？通过编程仿真进行测试和验证。
5. CPU 除了通过查询的方式使用 8255 传送数据外，还能使用中断请求的方式传送数据。下面，请读者使用一片 8255、一片 8259A 中断控制器芯片，设计实验电路，通过中断请求传送的方式传送数据。

提示：

1. 为 8255、8259A 分配端口地址。
2. 将 8255、8259A 与 8086 微机系统通过电路网络连接。
3. 使用 8086 指令编写 8255、8259A 的初始化程序，以及数据传送代码，实现 CPU 从 A 端口读入数据，将输入的数据加 1，然后再通过 B 端口输出的功能。

## 实验 3 可编程定时器/计数器 8253

实验性质：验证+设计

建议学时：2 学时

### 一、实验目的

- 了解 8253 芯片的功能和内部电路原理。
- 掌握 8253 芯片编程。

### 二、预备知识

8253 的内部结构如图 3-1 所示。它有 3 个独立结构完全相同的计数器，每个计数器包含一个 16 位计数器和一个 8 位的工作方式控制寄存器。

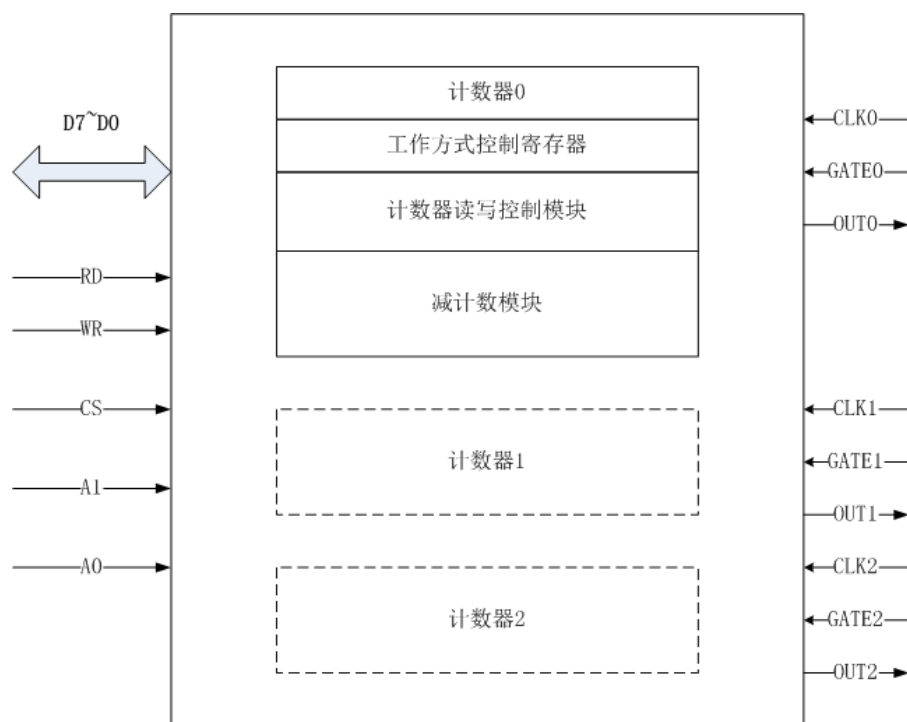


图 3-1: 8253 内部结构框图

### 三、实验内容

请读者按照下面方法之一在本地创建一个项目，用于完成本次实验：

#### 方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建个人项目，然后使用 Dream Logic 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

#### 方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，就需要使用 Dream Logic 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的 URL 为

<https://www.codecode.net/engintime/Dream-Logic/Project-Template/Microcomputer/Lab003.git>

### 3.1 任务（一）

熟悉 8253 内部电路及管脚功能。

1. 打开项目下的原理图文件 8253/8253.dlsche。
2. 在 8253.dlsche 中找出下列各引脚对应的端口。
  - D0~D7: 数据线, 双向, CPU 通过数据线输入计数器的方式控制字, 通过数据线读写计数器。
  - A0, A1: 地址线, 输入, 用于选择 3 个计数器中的一个及选择方式控制字寄存器。
  - RD: 读信号, 输入, 低电平有效, 有效时, 读出计数器的当前计数值。
  - WR: 写信号, 输入, 低电平有效, 有效时, 写入计数器的方式控制字或者计数器置初值。
  - CS: 片选端, 输入, 低电平有效, 有效时, 可对芯片进行读写操作, 否则禁止读写。
  - CLK0~CLK2: 计数器 0#, 1#, 2#的时钟输入端。
  - GATE0~GATE2: 计数器 0#, 1#, 2#的门控制脉冲输入端, 有外部设备输入门控脉冲。
  - OUT0~OUT2: 计数器 0#, 1#, 2#的输出端。

8253 的功能体现在两个方面, 即计数与定时。两者的工作原理在实质上是一样的, 都是利用计数器作减 1 计数, 减至 0 发信号。

1. 双击计数器 0 打开, 找到其内部的计数方式控制寄存器 CW, CW 编码 6 种计数方式, 以完成定时、计数或脉冲发生器等多种功能。
2. 打开计数器 0 的读写控制模块 RW, 模块 RW 用于该控制计数器的方式控制字的写入, 以及计数器的读写。
3. 打开计数器 0 的减计数模块 RE, 在输入时钟的驱动下, RE 按指定的工作方式对计数器数值的加载、减计数以及 OUT 信号的输出。

减计数模块下包含 6 种计数方式模块, 可结合每个计数方式模块电路图和注释了解每种计数方式的特点:

- 计数方式 0: 定时中断触发器
- 计数方式 1: 可编程单稳态触发器
- 计数方式 2: 分频器
- 计数方式 3: 方波发生器
- 计数方式 4: 软件触发选通脉冲
- 计数方式 5: 硬件触发选通脉冲

### 3.2 任务（二）

单独仿真 8253 芯片, 通过手动控制方式测试 8253 的引脚功能。

任务要求: 设 8253 的计数器 0 工作在方式 0, 采用 16 位计数, 先读写低字节, 后读写高字节。计数初值为 7, 减为 0 时输出一个正向跳变的上升沿。

#### 初始化计数器 0 的计数方式寄存器

1. 打开项目下的原理图文件 8253\_test.dlsche, 按下快捷键 F5, 启动仿真。
2. 手动设置计数器 0 的门控输入信号 GATE0, 使 GATE0 为低电平, 禁止计数器 0 的计数。
3. 根据任务要求, 设置 8253 的方式控制字为 0x30。首先, 选中图中的 8 位数字信号源器件, 在属性栏中将其数字信号值修改为 30。
4. 将地址输入端 A1、A0 均设置为高电平, 从而选中计数器 0 中的计数方式控制寄存器 CW。
5. 打开计数器 0, 观察计数方式控制寄存器 CW, 按下键盘按键 W 后抬起, 输入一个正向跳变的写脉冲信号, 将输入数据 D0~D7 写入 CW 中, 值为 0x30。

#### 初始化计数器 0 的计数初值

以上步骤将计数方式字写入计数器 0 的方式控制寄存器 CW 中, 接下来将计数初值写入计数器 0 的初

值寄存器 CH、CL 中。

1. 将 8 位数字信号源输出修改为 07，准备写入计数器 0，作为计数初值。
2. 将 A1、A0 都置为低电平，选中计数器 0。
3. 打开计数器 0 的减计数模块 RE，观察低字节计数初值寄存器 CL，按下按键 W 后抬起，输入一个正向跳变的上升沿，将 8 位初值 D0~D7 写入 CL 中，值为 0x07。
4. 返回顶层电路，将 8 位数字信号源的输出修改为 0，准备将其写入计数器 0 的高字节计数初值寄存器 CH 中。
5. 打开计数器 0 的减计数模块 RE，观察高字节计数初值寄存器 CH，按下按键 W 后抬起，输入一个正向跳变的上升沿，将 0 写入 CH 中。
6. 通过双击空白处，返回顶层电路。

### 计数器 0 启动计数

至此，计数器 0 被设置为工作方式 0，计数初始值被设置为 7。接下来，通过门控信号 GATE0 控制计数器 0 的计数。

1. 通过按键 G，使计数器 0 的门控信号 GATE0 为高电平，启动计数。
2. 在输入时钟的驱动下，计数器 0 做减 1 计数。
3. 当计数值减为 0 时，即 OUT0 输出高电平时，暂停仿真。
4. 双击逻辑分析仪打开，观察波形。
5. 通过水平滚动条定位到 OUT0 输出从高电平变为低电平，然后又变为高电平的波形段，结合 WR、GATE0、CLK0，分析计数器 0 的输出波形是否符合任务要求。

### 修改计数器 0 的计数初值，重新计数

修改计数器 0 的计数值，再进行仿真，观察 OUT0 输出波形。

1. 启动仿真，保持各个输入不变，将 8 位数字信号源输出修改为 0A。
2. 按下按键 W 后抬起，将 0A 写入计数器 0 的低字节计数初值寄存器。
3. 采用同样的方法，将 0 写入计数器 0 的高字节计数初值寄存器。
4. 当计数器 0 的计数值减为 0 时，暂停仿真，观察计数器 0 的输出波形 OUT0。
5. 结束仿真。

### 通过计数器 0 输出 8 分频方波

通过上面的步骤，我们模拟了计数器 0 工作在计数方式 0 时的情形。下面，我们使计数器 0 按方式 3 计数，输出 8 分频的方波，作为一个方波发生器。实验步骤如下：

1. 在原理图 8253\_test.dlschle 中，启动仿真。
2. 选中 8 位数字信号源器件，在属性栏中将其数字信号输出值改为 36。
3. 将 A0、A1、GATE 均设置为高电平。
4. 按下按键 W 后抬起，从 8253 芯片的 WR 端输入一个正向跳变沿，将 0x36 写入计数器 0 中的方式控制寄存器 CW 中，设置计数器 0 按计数方式 3 计数，输出方波。
5. 将 8 位数字信号源的数字输出值修改为 8。
6. 将 A0、A1 设置为低电平。
7. 按下按键 W 后抬起，将 08 写入计数器 0 的低字节计数初值寄存器 CL 中。
8. 将 8 位数字信号源的数字输出值修改为 0。
9. 按下按键 W 后抬起，将 0 写入计数器 0 的高字节计数初值寄存器 CH 中。
10. 双击逻辑分析仪打开，OUT0 输出 8 分频的方波（高低电平各占 4 个时钟周期）。
11. 将 GATE 置为低电平，停止计数，OUT0 输出立即变为高电平。观察相关波形。
12. 将 GATE 置为高电平，重新加载初值 8 并启动计数，继续输出 8 分频方波，观察波形。

## 13. 结束仿真。

**设置计数器 0 作为 5 分频器**

通过前面的两个任务，我们分别控制 8253 芯片的计数器 0 工作在方式 0 和方式 3。请读者通过手动控制的方法，使 8253 芯片的计数器 0 工作在方式 2，OUT0 输出 5 分频方波，并详细记录操作的过程。

提示：

- 设置方式控制字为 0x34，计数器 0 工作在方式 2。
- 设置低字节初值寄存器的值为 5，表示 5 分频。
- 设置高字节初值寄存器的值为 0，计数初值没有用到高 8 位。
- 计数过程中必须保持 GATE0 为高电平，否则计数停止。

**3.3 任务（三）**

将 8253 连接到 8086 微机系统上，通过编程控制 8253 的工作方式。

任务要求：通过编程初始化 8253，使计数器 0 的 OUT0 端输出 9 分频波形。

**9 分频器**

按照下面的步骤进行实验：

1. 打开项目下的 MEM/ram.asm 汇编源程序文件，结合注释，理解代码。
2. 打开项目下的原理图文件 top.dlsche。启动仿真。
3. 通过按键 R 进行复位。
4. 通过按键 s 设置数字信号源输出低电平，使用自动时钟，程序自动执行。
5. 当初始化代码执行完成后，打开逻辑分析仪，观察 OUT0 输出的 9 分频波形，该 9 分频波形的高电平占了 8 个时钟周期，低电平占一个时钟周期。
6. 结束仿真。

**根据输出波形判断计数器工作方式**

对代码进行适当的修改，从而改变计数器的工作方式。

1. 打开项目下的汇编源程序文件 MEM/ram.asm，将汇编指令“mov al, 0x34”替换为指令“mov al, 0x36”，保存文件，然后运行脚本文件 ram.bat，编译新的源程序代码。
2. 编译成功后，打开原理图文件 top.dlsche，启动仿真。
3. 当程序执行完成后，打开逻辑分析仪，观察 OUT0 输出的波形。根据波形判断此时，计数器 0 工作在何种方式？
4. 结束仿真。

**修改计数器初值，产生不同频率的波形。**

1. 打开项目下的汇编源程序文件 MEM/ram.asm，将汇编指令“mov al, 9”替换为指令“mov al, 4”，保存修改，然后运行脚本文件 ram.bat，编译新的源程序代码。
2. 编译成功后，打开原理图文件 top.dlsche，启动仿真。
3. 当程序执行完成后，打开逻辑分析仪，观察 OUT0 输出的波形，输出波形有什么变化？
4. 结束仿真。

**初始化计数器 1，使其输出 2 分频方波。**

1. 打开项目下的汇编源程序文件 MEM/ram.asm，在计数器 0 的初始化代码后添加计数器 1 的初始化代码，代码如下：

```
mov al, 0x76 ; 选择计数器 1，方式 3，方波发生器
```



```

out 0x83, al    ; 写计数方式控制字
mov al, 2       ; 2 分频
out 0x81, al    ; 先写低字节计数初值 2
mov al, 0       ;
out 0x81, al    ; 再写高字节计数初值 0

```

2. 保存汇编源文件 ram.asm。
3. 运行项目下的批处理文件 ram.bat 编译源程序。
4. 编译成功后，打开原理图文件 top.dlsche，启动仿真。
5. 程序执行完成后，打开逻辑分析仪，观察 OUT0 和 OUT1 输出的波形。
6. 结束仿真。

### 3.4 任务（四）

编写程序，使计数器 0、计数器 1、计数器 2 分别产生 2 分频、4 分频和 8 分频的方波。通过仿真验证你的设计。

**提示：**

1. 3 个计数器使用同一个计数时钟。
2. 在 top.dlsche 电路中，将计数 2 的门控信号 GATE2 以及输出信号 OUT2 连接到逻辑分析仪的剩余管脚，以便在一个窗口中观察所有计数器的波形。

## 四、思考与练习

1. 使用一片 8253 定时计数器和一片 8259A 中断控制器与 8086 微机系统构建一个定时中断系统。模拟计数器向系统日历时钟提供定时中断。

**要求：**

1. 设置 8253 的计数器 0 输出 30 分频的方波，即 30 分频器。利用 OUT0 输出方波的上升沿触发 8259A 的 0 号中断请求。
2. CPU 响应 8259A 的中断请求后运行中断服务程序，功能自行设定，但是要保证中断在下个中断请求到来之前返回。可以使计数器时钟和微机系统时钟一致，然后使中断服务程序的执行周期数小于 30 即可。

# 实验 4 可编程中断控制器 8259A

实验性质：验证+设计

建议学时：4 学时

## 一、实验目的

- 熟悉 8259A 芯片的内部电路原理。
- 掌握测试 8259A 芯片功能的方法。
- 掌握 8259A 与 8086 的电路连接方法。
- 掌握 8259A 芯片编程，并通过仿真熟悉 8086 处理硬中断的全过程。

## 二、预备知识

### 1. 8086 的中断类型

8086 的中断包含**硬件中断**和**软件中断**。

**软件中断**，也称内中断，是由执行某些指令引起的。软中断一般由下列 4 种情况引发：

- DIV 或 IDIV 指令：当执行这些除法指令时，若除数为 0 或商溢出，则一定会产生中断，这叫 0 型中断；
- INT 指令：当执行  $INTn$  时，则产生  $n$  型中断；
- INTO 指令：若指令序列执行过程中，前面指令的执行结果使溢出标志位  $OF=1$ ，接着若执行 INTO 指令，则会引起内部中断，称为 4 型中断；
- 单步执行。用于调试。

**硬件中断**，也称外中断，是由外部接口设备引起的。8086 通过可编程中断控制器 8259A 接受外部的可屏蔽中断请求，并按优先级顺序依次响应。8086（CPU）进入中断服务程序后，中断控制器 8259A 仍负责对外部中断请求的管理。

8259A 中断控制器的主要功能如下：

- 一片 8259A 可以接受并管理 8 级可屏蔽中断请求，最多 9 片级联可扩展至 64 级可屏蔽中断优先权控制。
- 每一级中断都可以通程序来屏蔽或允许。
- 在中断响应周期，可为 CPU 提供相应的中断类型号。
- 具有多种工作方式，可通过编程来选择。

### 2. 8259A 的内部结构

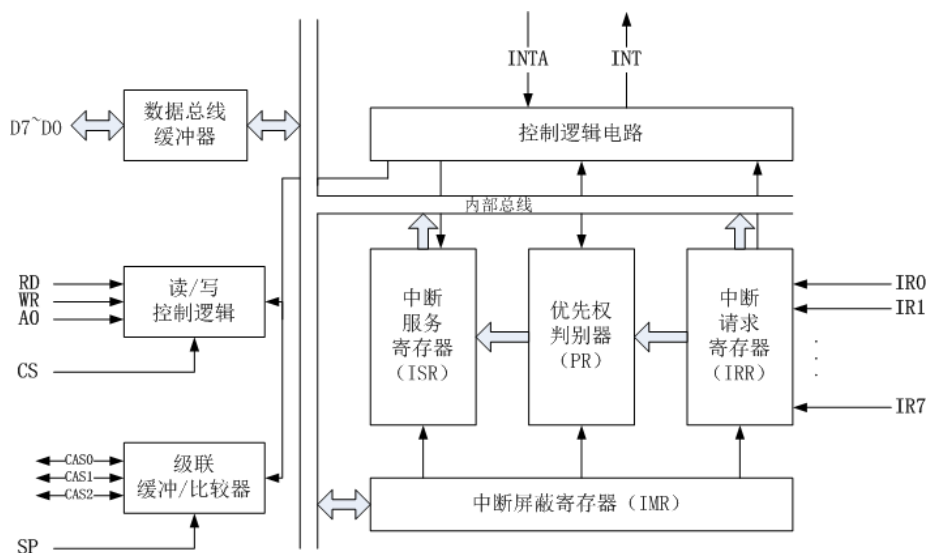


图 4-1: 8259A 内部结构

#### （1）中断请求寄存器（IRR）

中断请求寄存器 IRR 用于保存外设通过 IR0~IR7 提出的可屏蔽中断请求。当 IR7~IR0 的某端提出中断请求时，中断请求寄存器的相应位被置 1；中断请求被响应时，相应位清 0。

#### （2）中断服务寄存器（ISR）

中断服务寄存器用于保存所有正在被响应的中断。当 IR0~IR7 的某一中断请求被响应时，中断服务寄存器相应位被置 1。中断嵌套时，ISR 会有多位同时为 1。

#### （3）中断屏蔽寄存器（IMR）

中断屏蔽寄存器用于屏蔽或允许 IR0~IR7 提出的中断请求，某位置 1，相应的中断请求被屏蔽；置 0，中断允许。

#### （4）优先权判别器（PR）

多个中断源同时提出服务请求时，最高优先级的中断首先被响应。

### 3. 8259A 各引脚的定义如下

#### （1）与 CPU 的接口引脚

- D0~D7：三态数据输入输出引脚，与数据总线相连。
- RD：读控制信号输入引脚，与 CPU 的 IOR 相连，低电平有效，对 8259A 内部寄存器进行读操作。
- WR：写控制信号输入引脚，与 CPU 的 IOW 相连，正向跳变上升沿触发对 8259A 内部寄存器进行写操作。
- INT：中断请求信号输出引脚，与 CPU 的输入引脚 INTR 相连，向 CPU 发送中断请求信号。
- INTA：中断响应信号输入引脚，与 CPU 的输出引脚 INTA 相连，接收 CPU 的中断响应信号，低电平有效。
- CS：片选信号输入引脚，与地址译码电路相连。
- A0：片内端口选择信号输入引脚，一片 8259A 占用两个连续的端口地址。A0=0 时，选择偶地址端口；A0=1 时，选择奇地址端口。

#### （2）与外设的接口引脚

IR7~IR0：外设中断请求信号输入引脚，与外设的中断申请电路相连，向 CPU 提出中断请求。

#### （3）级联控制引脚

- CAS2~CAS0：级联信号引脚，单片 8259A 时无效；级联时，主片 8259A 的级联信号是输出引

脚, 从片 8259A 的级联信号是输入引脚。即主片的 CAS2~CAS0 输出作为所有从片的 CAS2~CAS0 端的输入。它的作用是, 在 CPU 响应主片的中断请求后, 需要获取对应的中断类型号, 如果被响应的中断请求来自于从片, 那么主片将当前响应的中断对应的 3 位标号(比如主片的 IR2 中断请求被响应, 那么标号就是 2, 对应 010, 即 CAS2=0, CAS1=1, CAS0=0)通过引脚 CAS2~CAS0 输出到从片, 对应的从片将其中断服务寄存器中的最高优先级中断请求类型号输出到总线, 供 CPU 读取, CPU 根据中断号找到对应的中断服务程序入口地址, 并转入执行服务程序。

- SP: 主从信号输入引脚。输入高电平时为主片; 输入低电平时为从片。

#### 4. 8259A 对外部中断的处理过程

8259A 中断控制器能处理外部设备的可屏蔽中断, 具体步骤如下:

##### (1) 接收外部的中断请求

外部中断请求通过输入引脚 IR0~IR7 输入, 中断请求寄存器 IRR 的相应位置 1, 接收并锁存外部中断请求。

##### (2) 判断中断请求是否被屏蔽

根据中断屏蔽寄存器 IMR 的相应位, 判断锁存的中断请求是否被屏蔽。对应位为 0 时, 相应的中断请求没有被屏蔽, 允许进入优先权判别器 PR; 否则, 该中断请求被屏蔽, 不允许进入优先权判别器 PR。

##### (3) 优先权裁决

优先权判别器 PR 对新进入的中断请求与中断服务寄存器 ISR 中正在处理的中断进行优先权裁决。

##### (4) 中断申请

中断处理期间, 如果 8259A 又接收到新的中断请求, 则首先与当前处理的中断请求的优先级进行比较。如果新的中断请求优先级高于当前正在处理的中断请求, 则 8259A 从 INT 引脚输出高电平, 向 CPU 提出中断申请, 处理新的中断请求; 否则将新的中断请求放入中断服务寄存器 ISR 中, 等待 CPU 响应。

##### (5) 中断响应

如果中断允许标志位 IF 为 0, 则 CPU 不会响应中断请求; 当 IF 为 1 时, CPU 执行完当前指令后, 查询中断请求输入引脚 INTR 是否为高电平, 若 INTR 为 1, 表示有中断请求。于是 CPU 跳转到一段固定的微程序处执行, 该段微程序首先完成保护现场的工作, 然后通过 INTA 引脚向 8259A 发出第一个负脉冲, 8259A 接收到该负脉冲后, 得知 CPU 可以响应它的中断请求了, 8259A 并将中断服务寄存器 ISR 中的最高优先级对应的标号暂存, 同时清除中断请求寄存器 IRR 中锁存的对应中断请求, 避免重复响应。CPU 通过 INTA 向 8259A 发送第二个负脉冲, 8259A 将暂存的标号组合成 8 位中断类型码通过数据总线 D0~D7 送个 CPU。CPU 通过获取的中断类型号, 得到对应的中断向量, 然后读出该中断向量中的地址, 该地址就是中断服务程序的入口地址, CPU 加载该地址就跳转到对应的中断服务程序执行。中断服务程序执行完成后, 8259A 的中断服务寄存器 ISR 中对应位清 0, 表示中断服务结束。

中断返回时, 通过出栈恢复现场, 从断点处继续执行指令。

#### 5. 8259A 的工作方式

8259A 的工作方式很多, 包括引入中断请求方式、中断屏蔽方式、设置优先级的方式、结束处理的方式及连接系统总线的方式。这些控制方式都可以通过编程来设置。

##### (1) 中断触发方式

- 边沿触发方式: 边沿触发方式是中断申请 IR7~IR0 出现由低电平向高电平的跳变时表示有中断请求信号。该方式的优点是申请端的 IRi 端可以一直保持高电平不会误判为又发生一次中断申请。应注意的是在 CPU 响应中断, 发回第一个 INTA 回答信号前, 已申请过中断的 IRi 端不会发生又一次的由低电平到高电平的跳变。也就是说, 同一个设备不能在旧的中断请求未被响应的情况下, 又发出一个新的中断申请。该方式的实现方法是使初始化命令字 ICW1 的 D3 位置 0。
- 电平触发方式: 电平触发方式是 IR7~IR0 的中断申请端出现高电平, 作为中断申请信号。该方式应注意的是中断申请信号需要保持到第一个 INTA 信号到来, 若时间过短会丢失该信号; 另外,

在 CPU 响应中断后, ISR 的相应位置位后, 必须撤除中断申请信号, 即将申请输入端置为低电平, 否则会一直发生中断申请。

## (2) 结束中断处理的方式

当某个中断源得到 CPU 的中断服务后, 要使中断服务寄存器 (ISR) 的相应位置 1, 表示 CPU 正在为该中断源服务。如果 CPU 正在为某个中断源服务时, 产生了比该中断源级别更高的中断申请, CPU 暂时中止地级别的中断服务, 转到高级别的中断服务, 同时也使高级别的中断源在中断服务寄存器的相应位置 1, 所以中断服务寄存器的状态反应出当前 CPU 正在为哪些中断源服务。当中断处理结束时, 必须将中断服务寄存器的相应位置 0, 表示该中断源的中断服务已经结束, 这个使中断服务寄存器的相应位置 0 的动作叫做中断结束处理。

结束中断处理的方式有:

### ● 中断自动结束方式

当某一级中断被 CPU 响应后, CPU 送回第一个 INTA 中断回答信号, 该信号使中断服务寄存器 ISR 的相应位置 1, 当第二个 INTA 负脉冲结束时, 自动将 ISR 的相应位置 0。

中断自动结束方式只适用于单片 8259A, 并且各中断不会发生嵌套的情况。因为 CPU 响应中断后第二个 INTA 信号使中断服务寄存器 ISR 的相应位置 0, 但该中断源的中断服务程序才开始执行, 在此时如果有同级或低级的中断申请, 只要 CPU 的中断是开放的, 就会打断该高优先级中断服务程序的执行, 这是不允许的。

使用方法: 通过初始化命令字 ICW4 的 D1 (AEOI) 设置为 1 实现。

### ● 一般中断结束方式

该方式是通过软件方法发送一个中断结束命令, 使当前中断服务寄存器 ISR 中级别最高的置 1 位清 0。当 CPU 用输出指令向 8259A 发送普通的 EOI 命令时, 8259A 会把 ISR 中最高优先级位复位。因为最高 ISR 位对应于最后一次被响应 (优先级最高) 的中断, 也就是当前正在处理的中断, 所以最高 ISR 位复位相当于结束了当前正在处理的中断。

使用方法: 首先设置初始化命令字 ICW4 的 D1 (AEOI) 位为 0, 定为正常中断结束, 要求 CPU 发送 EOI 命令复位 ISR。然后 CPU 执行向 8259A 写 OCW2 的指令, 写入字的 D7 (R, R=0, 固定优先级方式, 最高优先级始终为 0), D6 (SL, SL=0 一般中断结束方式), D5 (EOI, EOI=1 使当前 ISR 中的对应位复位) 位为 001。

### ● 特殊的中断结束方式

该方式也是通过软件方法发一中断结束命令, 与一般的中断结束方式不同的是, 特殊中断结束方式 EOI 命令中给出 ISR 需要复位的是哪一位。

使用方法: 首先设置初始化命令字 ICW4 的 D1 位为 0, 设为正常中断结束方式。然后通过设置控制命令字 OCW2 的 D7、D6、D5 位为 011 或 111, D2、D1、D0 位给出结束中断处理的中断源号, 使该中断源在中断服务寄存器中的相应位清零。

## (3) 屏蔽中断源的方式

8259A 内部有一个屏蔽寄存器 IMR, 它的每一位对应于一个中断请求输入。通过编程可以使 IMR 任意一位或几位置 0 或 1, 从而禁止或允许相应的中断。中断屏蔽方式分为普通屏蔽方式和特殊屏蔽方式。

### ● 普通屏蔽方式

当 CPU 执行主程序时不希望某几个中断源申请中断, 或在执行某一个中断源的中断服务程序时, 不希望比该中断源级别更高的中断源申请中断时, 可对此中断源进行屏蔽。

使用方法: 操作命令字 OCW1 的相应位置 1 或置 0, 置 1 表示该位的中断源被屏蔽, 置 0 表示该位的中断源没有被屏蔽。

### ● 特殊屏蔽方式

在某些情况下, 希望能在中断处理过程中动态改变系统的优先级结构。例如, 在执行中断服务程序的某个部分时, 允许比本身优先级低的中断。为达到此目的, 可执行写命令字 OCW1 指令, 从而使屏蔽寄存器中对应位置 1, 为开放较低的中断请求提供可能。但 8259A 有这样一个问题, 当

一个中断请求被响应时，就会使 ISR 对应位置 1，只要中断没有结束处理，8259A 就会据此禁止所有优先级比它低的中断，于是引进特殊屏蔽方式。设置了特殊屏蔽方式后，再变成使 IMR 某位置位 1，就会同时使 ISR 的对应位复位。这样，不只屏蔽了当前正在处理的这级中断，而且真正开放了其他较低级的中断。当然未屏蔽的更高级中断也可以得到响应。

使用方法：在某级中断服务程序中首先设置命令字 OCW3 的 D6、D5 位为 11，进入特殊屏蔽方式，然后通过设置命令字 OCW1 使该级的中断申请被屏蔽。

#### (4) 优先级设置方式

按照优先级设置方式分类，8259A 有普通全嵌套方式、特殊全嵌套方式、优先级自动循环方式、优先级特殊循环方式四种。

##### ● 普通全嵌套方式

普通全嵌套方式是 8259A 最常用的方式，简称全嵌套方式。在该方式中，中断源 IR7~IR0 的优先级别顺序是 IR0 最高，IR7 最低。当中断被响应时，8259A 就把请求中断优先级最高的中断类型号送上数据总线，ISR 中相应位置 1。在 ISR 相应位置位期间，仅允许比该级更高的中断源申请中断，不允许比该级别低或同级的中断源申请中断。

在普通全嵌套方式以及特殊全嵌套方式中，一定要预先设置 AE0I=0，使中断结束处于正常结束方式（要求 CPU 发 EOI 命令取复位 ISR）。这是因为中断结束命令使中断服务寄存器的相应位置 0，代表 CPU 对该级中断源的中断服务程序已经结束，允许比该级别低的中断源申请中断；若 AE0I=1，则为中断自动结束方式，当某一级中断申请得到 CPU 响应后，在第二个中断响应脉冲 INTA 到来时，8259A 会自动结束当前中断服务寄存器 ISR 中的对应位，这时，实际上该级中断源的中断服务程序才开始执行，尚未结束，中断服务寄存器的对应位却是 0，若此时出现比该级中断源优先级低的中断申请就可能得到响应，使中断优先级次序发生混乱。因而使用自动结束中断方式时要特别引起注意。

##### ● 特殊全嵌套方式

特殊全嵌套方式和普通全嵌套方式基本相同，只有一点不同，特殊全嵌套方式不但响应比本级高的中断申请，而且响应同级的中断申请，从而实现一种对同级中断请求的特殊嵌套。

特殊全嵌套方式一般用于 8259A 级连工作方式中的主片。在级连方式中，从片 8259A 的中断请求通过主片发送到 CPU，如果当前正在执行的某一中断服务程序是从片的 IRi 中断源引起的，这时有可能发生该从片的另一 IRi-1 端的中断源申请中断，若从片 8259A 的 IRi-1 中断源比 IRi 中断源的优先级别高，主片 8259A 予以响应。从主片 8259A 来看，一个从片 8259A 中的所有中断源都是同级别优先级，但是从片 8259A 中多个中断源优先级是不同的，主片为了能响应同一级中断，主 8259A 就需要设为特殊全嵌套方式，从 8259A 设为其他优先级方式。

实现方法：工作方式控制字 ICW4 中的 D4 为置 1 为特殊的全嵌套方式，置 0 为普通全嵌套方式。另外，在特殊全嵌套方式中，对中断结束的操作硬注意软件必须检查岗结束的中断是否是主片的唯一中断。其方法是：先向从片发一正常结束中断命令 EOI，然后读从片 ISR 的内容。若为 0 表示从片只有一个中断服务，这时再向主片发一个 EOI 命令；否则，说明该从片还有未结束的中断服务，不应给主片发 EOI 命令。

##### ● 优先级自动循环方式

优先权自动循环方式下，优先级队列是在变化的。初始化优先级顺序为 IR0~IR7，当某一个中断源受到服务后，它的优先级别自动降为最低，而降最高优先级赋给比它低一级的中断源。若 IR0~IR7 端引入的中断源有相同的优先级，使用自动循环方式，使得每个中断源有同等的机会得到 CPU 的服务。

实现方法：设置操作命令字 OCW2 的 D7、D6 位分别为 1、0。

##### ● 优先级特殊循环方式

该方法同优先级自动循环方式相比，不同点仅在于可以根据用户要求通过编程将初始化优先级赋

予某一中断源。

### 三、实验内容

请读者按照下面方法之一在本地创建一个项目，用于完成本次实验：

#### 方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建个人项目，然后使用 Dream Logic 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

#### 方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，就需要使用 Dream Logic 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的 URL 为

<https://www.codecode.net/engintime/Dream-Logic/Project-Template/Microcomputer/Lab004.git>

#### 3.1 任务（一）

8259A 中断控制器芯片功能多样，内部电路结构复杂，为了简化电路，以便理解，本实验中提供的 8259A 芯片只保留了常用的核心功能。

在使用 8259A 进行实验前，请读者打开实验项目文件夹 8259A 中的原理图文件 8259A.dlsche，首先熟悉 8259A 的内部电路及主要模块。如表 4-1 所示：

模块名称	模块原理图文件	模块功能
INIT	Initial.dlsche	初始化模块，负责 8259A 内部命令字寄存器 ICW1~ICW4，OCW1~OCW3 的初始化。
IRR	IRR.dlsche	中断请求模块，保存外设通过 IR0~IR7 提出的可屏蔽中断请求。当 IR7~IR0 的某端提出中断请求时，IRR 的相应位被置 1；中断请求被响应时，相应位清 0。
ISR	ISR.dlsche	中断服务模块，保存所有正在被响应的中断。当 IR0~IR7 的某一中断请求被响应时，ISR 的相应位被置 1。中断嵌套时，ISR 会有多位同时为 1。
PR	PR.dlsche	优先级判别模块，将 IRR 中新进入的所有中断请求与 ISR 中正在服务的中断请求进行优先级比较，若 IRR 中存在优先级高于当前正在服务的中断请求，那么 PR 通过端口 INT 发出中断申请，通知 CPU 有更高优先级的中断，CPU 于是暂停正在运行的中断服务程序，响应新进入的高优先级中断请求。
NUMER	Acs.dlsche	CPU 从 NUMER 模块读取被响应中断对应的中断类型号（中断向量），通过中断类型号获取中断服务程序入口地址，加载中断服务程序入口地址到程序计数器 PC 中，即可转去执行中断服务程序。

表 4-1:8259A 内部的主要模块

#### 3.2 任务（二）

本任务是单片 8259A 中断系统实验，通过本次实验任务，希望读者掌握以下几点：

1. 单片 8259A 中断控制器与 8086 通过网络连接，构成一个可处理 8 级中断的微机系统。
2. 编程初始化 8259A 中断控制器。
3. 8259A 向 8086 申请中断服务的工作原理。

#### 8259A 与 CPU 连接构成一个可处理中断的系统

1. 打开项目下的原理图文件 top.dlsche。

2. 找到图中的 8259A 芯片模块，可以看到 8259A 的右侧接口 IR0~IR7 接的是单周期时钟，单周期时钟在按下和抬起操作过程中会产生一个正向的跳变沿，即上升沿“↑”，该上升沿将触发对应的中断请求。
3. 8259A 的接口 CS 与 IO 译码器的输出 CS0 连接。CS 是 8259A 的片选信号，当 CPU 通过端口地址访问外设时，端口地址通过 IO 模块进行译码，从而产生对应的芯片片选信号。CS0=0 有效时，表示 CPU 访问的端口地址在 0x80~0x87 范围内。8259A 的端口只需要一位地址 A0，因此，8259A 只使用到了两个端口地址 0x80 和 0x81。
4. 8259A 的接口 WR 和 RD 分别与 IO 译码器的 IOW 和 IOR 连接。当 CPU 访问 8259A 的端口时，一种是读，一种是写，是通过 IOR 和 IOW 来控制的。
5. 8259A 的接口 SP 始终为高电平，表示该片没有级联其他的 8259A 芯片。
6. 8259A 的接口 CAS0~CAS2 悬空，默认为低电平，对单片 8259A 来说没有作用。
7. 8259A 的接口 INT 输出中断请求，它与 8086 的中断请求输入端 INTR 连接，从而向 CPU 发出中断请求。CPU 在每条指令执行最后总是查询管脚 INTR 的状态，若 INTR=1，则说明有中断请求，CPU 即可进行相应的处理。
8. 8259A 的接口 INTA 输入接的是 8086 的中断应答输出信号 INTA，当 CPU 通过 INTA 发出低电平时表示 CPU 响应 8259A 发出的中断请求。
9. 8259A 的接口 D0~D7 分别与数据总线 DBUS0~DBUS7 相接。这样，CPU 就可以通过数据总线将 8259A 的初始化命令字传送到 8259A 的相关寄存器，完成对 8259A 的初始化。CPU 也能通过 8259A 的 D0~D7 读取中断向量，以决定该运行哪个中断的服务程序。

中断微机系统构建完成后，就可以通过编程初始化中断控制器，使得 CPU 能够响应中断请求了。请读者打开项目下的 MEM/ram.asm 汇编源程序，结合注释，理解代码。

### CPU 中断过程自动仿真

在 CPU 自动运行主程序的过程中，向 CPU 申请中断服务。

1. 在原理图文件 top.dlsche 中启动仿真。
2. 打开 RAM 存储器窗口，0~7 号存储单元是中断向量，它们存储的地址分别指向中断服务程序 int0~int7 的入口（起始指令）。比如，0 号单元的内容是 37，在源代码窗口 1 中，中断服务程序 int0 的第一条指令地址正好是 37；1 号单元的内容是 47，指向中断服务程序 int1 的第一条指令等。
3. 复位后通过 S 键将时钟切换为自动时钟。
4. 等待 8259A 初始化结束，通过按键 1 输入一个上升沿，触发 IR1 中断请求。可以看到 8259A 的输出 INT=1，等待 CPU 响应。
5. 在源代码窗口 2 中观察硬中断处理微程序的执行过程。
6. 在源代码窗口 1 中观察中断服务程序 int1 的执行过程。
7. 中断服务程序执行完成后，可以看到 CPU 继续运行后续程序。
8. 结束仿真。

通过上述仿真，读者对中断全过程有了一个大概的了解。

### CPU 中断过程单步仿真

接下来，我们通过单步仿真，详细分析中断过程中的主要步骤。

1. 在原理图文件 top.dlsche 中启动仿真，切换到单步时钟后复位。
2. 首先运行的是关中断指令 cli。
3. 通过按键 C，输入一个单步时钟，源代码窗口 2 中的箭头指向微指令 cli。
4. 在寄存器窗口中找到 uPC，双击定位该寄存器，在仿真视图中可以看到，此时 ROM 输出的所有微命令（控制信号）中，只有 CLI 为低电平（有效）。



5. 在寄存器窗口中找到标志寄存器 FLAG，双击定位该器件，在该仿真视图中，我们可以看到，CLI 信号经过非门，输出高电平，从而清零了存储 IF 标志的 D 触发器，即 IF=0。
6. IF 标志位为 0，为什么能屏蔽中断请求？

在寄存器窗口中双击 uPC，打开控制单元 CU 的仿真视图，从图中左下角可以看到，IF=0，经过与非门和或门输出的 INT\_RESPOND 信号为高电平，而与中断请求信号 INTR 无关。INT\_RESPOND=1，那么它直接控制的三态门 4 关闭，因此硬中断处理微程序地址 0xEC0 就不会传送到 uPC，也就不会响应中断请求信号 INTR 了。

7. 继续单步运行，下一条指令为 movsp, 0。为什么要将栈寄存器 SP 初始化为 0 呢？

栈寄存器 SP 始终指向栈顶，每次入栈前，都需要先将 SP 减 1，再将数据入栈，因此将栈顶初始化为 0，那么第一个入栈的地址单元为  $0-1=-1=0xff$ ，后续入栈的依次为 0xfe, 0xfd, ...。这样，栈空间分配了 RAM 的最后一段存储空间，从而将有限的存储空间节省下来分配给指令或数据。

接下来是 8259A 的初始化程序。

1. 单步运行，每执行完一条写 8259A 端口的 out 指令，就打开 8259A 内部初始化模块 INIT，观察对应寄存器的写入值，保证写入值的正确性。
2. 开中断指令 STI 与 CLI 正好相反，STI 指令是将 IF 置位为 1。执行完 STI 指令后观察标志寄存器 FLAG 模块内的标志位 IF。

然后运行主程序，我们先触发中断请求，再单步运行，观察 CPU 是如何查询中断并进行响应的。

1. 通过按下并抬起按键 0 输入一个上升沿，触发 0 号中断请求，可看到 8259A 的输出 INT=1。
2. 打开源代码窗口 2，注意蓝色箭头的位置。
3. 通过按键 C 输入单步时钟，当运行到查询中断的微指令 checkirq 时停止输入时钟。
4. 打开 8086 内部的控制单元 CU。在图的左下侧可以看到，INT\_RESPOND=0，单向总线收发器 4 打开，uPC 输入端地址为 0xEC0。
5. 输入单步时钟，uPC 被置位为 0xEC0，在源代码窗口 2 中可以看到，蓝色箭头指向了硬中断处理微程序。

通过上述操作，我们不难看出，CPU 在每次执行完一条指令后都要查询一下中断请求输入管脚 INTR 的状态，若 INTR=1，且 IF=1，表示允许中断且有中断请求，那么 CPU 就会使 uPC 跳转到固定的硬中断处理微程序处执行，开始响应中断。

1. 硬中断处理微程序首先将标志寄存器和中断返回地址（下一条指令地址）入栈，完成现场保护，请读者结合注释，单步运行，了解入栈过程。
2. 当运行到微指令 inta 时，CPU 发出第一个中断应答信号。
3. 当运行到微指令 path inta, mar 时，CPU 发出第二个中断应答信号。这时，打开 8259A 的内部模块 NUMER，可以看到中断向量输出 D[7..0]=0。
4. 打开 CPU 可看到，地址寄存器 mar 写使能。
5. 输入单步时钟，mar 写入 0。该步完成的是将 0 号中断对应的中断向量转移到 mar。
6. 运行到微指令 path [mar], pc，该条微指令是将 mar 中的中断向量指向的中断服务程序入口地址加载到 PC，实现中断服务调用。
7. 输入单步时钟，在寄存器窗口中，可看到 PC=0x37，这正是中断服务程序 int0 的首地址。
8. 继续输入单步时钟，直到源代码窗口 1 中蓝色箭头指向 int0 的首条指令。

通过上述步骤，硬中断处理微程序实际完成了两件事：其一是，保护中断现场，将标志寄存器和返回地址依次入栈；其二是，从 8259A 读取中断类型号并通过中断类型号获取中断服务程序入口地址，将中断服务入口地址加载到 PC 中，转去执行中断服务程序。

1. 继续单步运行程序，当运行到 0 号中断服务程序的指令 `out 0x80, al` 时，打开 8259A 内部的中断服务寄存器 ISR，注意输出 ISR0 的状态变化。该条指令是中断结束 EOI 命令，该指令的执行会清除 ISR 中的相关位，表示该位对应的中断服务程序已经执行完毕。
2. 继续单步运行中断返回指令 `iret`，结合微指令的注释，理解出栈过程。
3. 中断返回后，继续运行后续指令。
4. 结束仿真。

通过上述步骤，可以看出中断服务程序需要完成三件事：

- ① 实现中断服务功能；
- ② 中断服务完成后通过写端口指令 OUT 向 8259A 发出中断结束命令 EOI；
- ③ 标志寄存器和返回地址出栈，恢复中断现场，实现中断返回。

### 多中断申请与中断嵌套仿真

通过前面的仿真，相信读者对中断全过程有了深刻的理解。接下来，请读者仿真以下情形，以掌握中断判优和中断嵌套原理。

1. 多个中断源同时向 CPU 提出中断请求的情形。请读者根据下列提示，带着相关问题进行仿真：
  - ① 8259A 初始化完成后，切换为单步仿真。
  - ② 通过按键向 8259A 提出两个中断申请，观察 8259A 的中断请求模块 IRR 中的相关位，然后打开中断判优模块 PR，观察多中断判优后输出的最高优先级编号  $IRNM2 \sim IRNM0$  的值， $IRNM2$  是高位， $IRNM2 \sim IRNM0$  的值代表最高优先级中断请求，例如 0 表示 0 号中断请求，7 表示 7 号中断请求。
  - ③ 单步运行，当 CPU 准备发出第一个应答信号  $INTA$  时，打开 8259A 的中断服务寄存器 ISR 模块，注意  $T0 \sim T2$  以及  $ISR0 \sim ISR7$  的状态。输入一个单步时钟，CPU 发出第一个  $INTA$  信号，再观察  $T0 \sim T2$  以及  $ISR0 \sim ISR7$  的状态，有什么变化？有何含义？此时打开 PR 模块，中断请求端口  $INT$  输出什么电平，为什么？
  - ④ 当高优先级中断服务程序执行完，CPU 发出中断结束命令时，打开 8259A 的 ISR 模块，观察  $ISR0 \sim ISR7$  中，哪一位被清 0？为什么？清零后，打开 PR 模块，此时  $INT$  和  $IRNM2 \sim IRNM0$  处于何种状态，为什么？
  - ⑤ 当 CPU 运行高优先级中断服务程序时，为什么 CPU 的中断请求输入管脚  $INTR$  始终处于低电平？
2. CPU 执行某个中断服务程序时，8259A 接收到一个更高优先级中断请求的情形。  
**提示：**高优先级中断请求打断低优先级中断服务，其实质是，当 CPU 运行低优先级中断服务程序时，查询到  $INTR$  为高电平，于是运行固定的硬中断处理微程序，从而转去运行新的中断服务程序。

### 3.3 任务（三）

通过前面的任务，我们完成了单片 8259A 中断控制器微机系统中断实验。接下来，请读者按步骤完成以下任务要求：

1. 打开项目下的原理图 `top1.dlsche`，通过电路连接，实现两片 8259A 级联，完成一个包含 15 级中断的微机系统。要求如下：
  - ① 8259A\_M 为主片，8259A\_S 为从片，从片的中断申请  $INT$  与主片的  $IR1$  级联。
  - ② 主片端口地址为  $0x80$  和  $0x81$ ，从片的端口地址为  $0x90$  和  $0x91$ 。
  - ③ 主从片的中断请求均为边沿触发。
  - ④ 主片使用特殊全嵌套方式，从片使用普通嵌套方式。
  - ⑤ 主从片均采用正常中断结束方式，即通过 CPU 发 EOI 命令复位 ISR。
  - ⑥ 主片的中断向量类型为 0，从片的中断向量类型为 2。
2. 将下列代码复制到 `ram.asm` 文件中，并填充中断向量表，补充主从片 8259A 的初始化程序以及各个中

断服务程序。

```
.int_table
```

```
;根据电路中的中断源填写中断向量表
```

```
.text
```

```
cli ; 关中断
```

```
; 初始化栈顶地址
```

```
mov sp, 0
```

```
; 初始化主片 8259A
```

```
; 写 ICW1, 边沿触发, 级连方式, 需要设置 ICW4
```

```
; 写 ICW2, 设定 IRQ0 的中断号为 0
```

```
; 写 ICW3, 设定主片 IRQ1 级联从片
```

```
;写入 ICW4, 设定特殊全嵌套方式, 普通 EOI 方式 (正常中断结束)
```

```
; 写 OCW1, 允许 0, 1, 2 号中断请求
```

```
;初始化从片 8259A
```

```
; 写入 ICW1, 设定边沿触发, 级联方式
```

```
;写入 ICW2. 设定从片 IRQ0 的中断向量号为 2
```

```
; 写入 ICW3, 设定从片级联于主片 IRQ1
```

```
; 写入 ICW4, 普通全嵌套方式, 普通 EOI 方式
```

```
; 写 OCW1, 允许全部 8 级中断请求
```

```
sti ; 开中断
```

```
BEGIN:
```

```
mov bl, 4
```

```
movcl, 2
```

```
shl bl, cl
```

```
rcl cl, bl
```

```
jmp BEGIN
```

```
;=====
```

```
; 主片的中断服务程序
```

```
Int_master0:
```

```
mov bl, 1
```

```
add bl, 8
```

```
mov al, 0x20
```

```
out 0x80, al
```

```
iret
```

```
;=====
```

; 从片的中断服务程序

```
Int_slave0:
mov dl, 8
addc dl, 2
mov al, 0x20
out 0x90, al
out 0x80, al
iret
```

3. 分别仿真以下情形

- ① 从片发出 1 号中断请求的中断处理过程。
- ② 主片发出 1 号中断请求，从片发出 2 号中断请求，CPU 处理两个中断的过程。
- ③ CPU 正在运行主片 2 号中断服务程序时，主片发出 0 号中断请求的情形。
- ④ CPU 正在运行主片 2 号中断服务程序时，从片发出 0 号中断请求的情形。
- ⑤ CPU 正在运行从片的 1 号中断服务程序时，从片发出 0 号中断请求的情形。若从片发出的是 2 号中断请求呢？

## 四、 思考与练习

1. 8259A 的中断请求是如何触发的？
2. 8259A 是如何从多个中断请求中选出最高优先级中断请求的？
3. CPU 在什么时候查询中断请求输入信号 INTR？CPU 查询到中断请求后是如何转到硬中断微程序处执行的？若没有中断请求，CPU 是如何继续取指，执行下一条指令的？
4. CPU 在运行硬中断微程序时，前后发出的两个中断应答信号 INTA 有何作用？
5. CPU 是如何获取响应中断的中断向量以及对应中断服务程序的入口地址的？
6. CPU 响应中断后，把标志寄存器和中断返回地址入栈，这是如何实现的？
7. 中断返回后，CPU 是如何返回断点处继续执行程序的？
8. 中断返回后，需要发送 EOI 结束命令给 8259A，该命令是为了清除 8259A 中断服务寄存器 ISR 中的对应标志位，从电路角度分析，这是如何实现的？
9. 多片 8259A 级联的中断实验中，主片的 CAS0~CAS1 是输出还是输入？从片呢？CAS0~CAS2 有何作用？请结合 CPU 响应从片中断请求时获取从片中断向量的过程加以分析。
10. 主片通常设置为特殊全嵌套模式，而从片设置为普通全嵌套模式，这样设置有何用意？

# 实验 5 串行通信接口 8250

实验性质：验证+设计

建议学时：2 学时

## 一、实验目的

- 了解 8250 芯片的内部电路结构以及工作原理。
- 熟练掌握 8250 芯片初始化编程，通过程序控制其工作方式。

## 二、实验内容

请读者按照下面方法之一在本地创建一个项目，用于完成本次实验：

### 方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建个人项目，然后使用 Dream Logic 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

### 方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，就需要使用 Dream Logic 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的 URL 为

<https://www.codecode.net/engintime/Dream-Logic/Project-Template/Microcomputer/Lab005.git>

### 2.1 任务（一）

由于实际的 8250 芯片功能复杂，不便于理解，因此本实验中使用的 8250 芯片是简化后的芯片，它保留了核心的串行收发功能以及中断请求功能。请读者按下列步骤熟悉 8250 的管脚和主要功能模块。

1. 打开项目下的原理图 8250\_test.dlsche。
2. 从图中找出以下 8250 芯片的模块接口（芯片引脚）：
  - 1) D7~D0：CPU 和 8250 通过此 8 位双向数据线传送数据或命令。
  - 2) CS：片选输入引脚，低电平有效，有效时选中 8250。
  - 3) A2,A1,A0：地址选择线，用来选择 8250 的内部寄存器。它们通常接地址总线的 ABUS2,ABUS1,ABUS0。
  - 4) DISTR：数据输入选通引脚。当 DISTR 为高电平时，CPU 就能从选中的 8250 寄存器中读出状态字或数据信息。DISTR 接系统总线上的 IOR。
  - 5) DOSTR：数据输出选通引脚。当 DOSTR 为高电平时，CPU 就能将数据或命令写入 8250。DOSTR 接系统总线上的 IOW。
  - 6) MR：复位信号输入引脚。一般接系统复位信号线 RESET。
  - 7) INTRPT：中断输出请求引脚，高电平有效。当 8250 允许中断时，接收数据寄存器满、发送数据寄存器空的状态均能产生有效的 INTRPT 信号。
  - 8) SIN：串行数据输入引脚。外设或其他系统送来的串行数据由此端进入 8250。
  - 9) SOUT：串行数据输出引脚。
  - 10) RCLK：接收和发送时钟输入引脚。
  - 11) RTS：请求发送输出引脚。发送数据寄存器满，向接收端请求发送允许。
  - 12) CTS：允许发送输入引脚。接收端接收寄存器空，通知发送端可以发送数据了。

3. 双击 8250 打开模块，熟悉 8250 的内部电路及主要模块。如表 5-1 所示：

模块名称	模块原理图文件	模块功能
reg_wr	reg_wr.dlsche	寄存器读写控制逻辑，产生 8250 内部各个寄存器的读写控制信号。
TCL	TCL.dlsche	串行数据发送模块，将发送寄存器 THR 中的数据通过 SOUT 串行输出。
RCL	RCL.dlsche	串行数据接收模块，接收 SIN 输出的串行数据，并将接收到的 8 位有效数据写入接收寄存器 RBR 中。
lsr	lsr.dlsche	通信线路状态寄存器，记录数据传输的状态信息，例如，发送寄存器 THR 满，接收寄存器 RBR 空等。
ier	ier.dlsche	中断允许寄存器，低 2 位允许 8250 设置 2 种类型的中断（将相应位置 1 即可）。
iir	iir.dlsche	中断识别寄存器，用来判断有无中断并判断中断是哪一类中断请求：发送寄存器 THR 空中断请求，通知 CPU 写入新的发送数据；接收寄存器 RBR 满中断请求，通知 CPU 读取接收数据。

表 5-1:8250 内部的主要模块

## 2.2 任务（二）

使用单片 8250 芯片完成自发自收实验，即手动将发送的字节写入 8250 的发送器，在收发时钟的驱动下，8250 将发送字节通过串行输出口 SOUT 逐位输出，通过串行输入口 SIN 接收 SOUT 的输出，最终完成一个字节的接收，将接收到的字节保存到 8250 的接收寄存器中。实验步骤如下：

1. 打开项目下的原理图文件 8250\_test.dlsche，熟悉 8250 的内部电路和各个模块的功能。
2. 启动仿真。
3. 通过按键 M，使 MR 输入高电平，复位 8250。复位后分别进入以下模块，观察复位信号的作用：
  - ① 8250 的模块 lsr 内部的 DR 和 THRE 分别被清零和置位，表示接收器空，发送器也为空。
  - ② 8250 的模块 RCL 内部接收器的有限状态机被初始化为起始状态。
  - ③ 8250 的模块 TCL 内部发送器的有限状态机被初始化为起始状态。
  - ④ 8250 的模块 ier 内部的中断允许寄存器被清零，表示禁止中断。
  - ⑤ 8250 的模块 iir 内部的中断识别触发器被清零，表示没有中断请求。
4. 通过按键 M 将 MR 设置为低电平，撤销复位信号。

复位后，就可以开始发送数据了。首先，我们需要将发送的字节写入发送器 TCL 中的寄存器 THR 中，而 THR 的时钟接的是信号 WR\_THR，只有 WR\_THR 从低电平变为高电平，才能将发送数据写入 THR 中。请读者按下列步骤，将发送数据 0xc7 写入 THR 中。

1. 返回顶层电路，将 D0~D7 的输入修改为 0xc7。
2. 通过按键 R 将 8250 的输入 DISTR 设为高电平，表示读无效。
3. 通过按键 W 将 8250 的输入 DOSTR 设为低电平，表示写有效。
4. 打开 8250 内部的读写控制模块 reg\_wr，可以看到 WR\_THR 为低电平，通过按键 W 将 8250 的输入 DOSTR 设为高电平，可以看到 WR\_THR 升为高电平，从而产生一个上升沿。该上升沿将输入数据 0xc7 写入了 THR 寄存器中，打开 TCL 观察 THR 寄存器，确实写入了 0xc7。
5. 打开 lsr 模块，可以看到 THRE 标志被清零，表示发送器数据满，可以发送数据了。

下面，我们分析发送器 TCL 是如何逐位发送寄存器 THR 中的字节的。

1. 打开发送器 TCL，可以看到此时有限状态机处于起始状态，即 S0~S3 均为低电平，其中 S0~S3 是当前

状态的编码。

2. 发送器使用 2 个 8 选 1 数据选择器，它们的输入端接收将要发送的每一位，它们的输出就是被选择用来发送的位。由于当前状态为 0，所以被选择发送的位为第一个 8 选 1 选择器 U3 的 D0 位，它是高电平，所以串行输出 SOUT=1。
3. 通过按键 C 输入一个发送时钟，可以看到 S0=1，该状态下选择发送的是第二位，它是一个低电平，输出 SOUT=0，标志从这一位开始，后面的位是有效数据位。
4. 继续输入发送时钟，依次发送每一位有效数据。
5. 我们注意到，最后发送的位是高电平，是字节结束的标志位。
6. 当状态回到起始态时，字节发送过程结束，状态保持不变，输出 SOUT=1 不变。
7. 打开 RCL 接收器，可以看到接收寄存器 RBR 接收到了字节 0xC7，打开 1sr 模块还可看到，接收就绪标志 DR=1。

接收器 RCL 的工作原理是，当接收寄存器 RBR 为空时，若串行输入 SIN=0，则表示 SIN 之后的位是一个有效字节，使用移位寄存器逐位采样，当接收完成后，接收寄存器 RBR 管脚 CLK 端输入一个时钟上升沿，从而将移位寄存器接收的字节写入 RBR，同时设置接收字节就绪标志 DR。若允许中断，还可以发出接收字节就绪中断请求，使 CPU 取走已经接收到的字节。

在此，请读者重复上述整个过程，完成字节 0x28 的自发自收，并着重分析接收器 RCL 的工作原理。

### 2.3 任务（三）

使用两片 8250 芯片和一片 8259A 中断控制器芯片，搭建一个微机系统。实现一个 8250 芯片向另一个 8250 发送字节，字节接收完成后触发中断，在中断服务程序中，CPU 读取接收到的字节，并将其移动到寄存器 b1 中。

实验步骤如下：

1. 打开项目下的原理图文件 top.dlsche，熟悉电路图，着重理解 8250 与 8259A 是如何与微机系统建立连接关系的。
2. 打开项目下的汇编源程序文件 MEM/ram.asm，结合代码注释，理解代码功能。
3. 启动仿真。
4. 复位后，使用单步或者自动时钟运行程序，理解各条指令实现的功能。
5. 当程序运行到标号 BEGIN 处时，将时钟切换为单步时钟。

所有芯片初始化完成后，CPU 开始运行主程序，这时使用 8250 进行字节传送。步骤如下：

1. 将 8250\_T 的 8 位数字信号源修改为 7，该字节将作为发送字节，由 8250\_T 发送给 8250\_R。
2. 通过按键 W 给 8250 输入一个低电平和电平，将 7 写入 8250\_T 的 TCL 模块中的发送寄存器 THR 中。
3. 通过按键 A 输入发送时钟，将字节 7 逐位发送给 8250\_R。
4. 打开 8250\_R 的接收器 RCL，在发送过程中，注意接收寄存器 RBR 的值，当其显示 7 时，再输入一个时钟，使状态机恢复为起始态。
5. 此时可以看到 8250\_R 中的 1sr 模块内的接收就绪标志 DR=1。8250\_R 中的 iir 模块内的中断请求输出端 INTRPT 变为高电平，从而触发接收数据满中断请求。返回顶层电路，打开 8259A 芯片，可看到模块 IRR 内部的 0 号中断请求 IRR0 变为高电平，且 8259A 的中断请求输出 INT=1，INT 与 8086 的中断输入 INTR 相连，从而通过中断的方式使得 8086 读取 8250\_R 接收到的字节。
6. 继续单步运行，直到 8086 响应中断并运行中断服务程序，将 8250\_R 接收到的字节读入 al 寄存器中，再由 al 移动到 b1 寄存器中。完成字节 0x07 从 8250\_T 串行发送到 8250\_R，由 8250\_R 的接收寄存器传送到 8086 的目的寄存器中整个过程。
7. 复位后，使用自动时钟运行程序，当 8259A 芯片初始化完成后，开始手动控制 8250 进行字节传送。这样，当字节传送结束后，CPU 自动查询中断并响应中断。
8. 中断程序返回后，查看 b1 寄存器的结果。

9. 结束仿真。

#### 2.4 任务（四）

请读者在任务三的基础上，连续传送两个字节，分别将 0x05、0x12 传送到 a1 中，然后将两个数相加，结果写回 b1 中。

**提示：**该任务中需要修改 ram 源程序，首先需要在中断服务程序外部将 b1 初始化为 0，在中断服务程序中，将每次通过 a1 接收到的数据与 b1 相加，写回 b1 中即可。



# 实验 6 DMA 控制器 8237A

实验性质：验证+设计

建议学时：2 学时

## 一、实验目的

- 了解 8237A 芯片的内部电路结构以及工作原理。
- 熟练掌握 8237A 芯片初始化编程，通过程序控制其工作方式。

## 二、实验内容

请读者按照下面方法之一在本地创建一个项目，用于完成本次实验：

### 方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建个人项目，然后使用 Dream Logic 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

### 方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，就需要使用 Dream Logic 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的 URL 为

<https://www.codecode.net/engintime/Dream-Logic/Project-Template/Microcomputer/Lab006.git>

### 2.1 任务（一）

由于实际的 8237A 芯片功能复杂，不便于理解，因此本实验中使用的 8237A 芯片是简化后的芯片，它保留了两个 DMA 传送通道，通道 0 和通道 1，以及核心的功能。请读者按下列步骤熟悉 8237A 的管脚和主要功能模块。

1. 打开项目下的原理图 8237/8237A.dlsche。
2. 从图中找出以下模块接口（芯片引脚）：
  - 1) A7~A0: 8 位双向地址线。A7~A0 的作用有两种：第一种是在 8237A 空闲时，提供 CPU 访问 8237A 寄存器的地址通道；第二种是在进行 DMA 操作时，读周期提供源存储器的数据地址，在写周期提供目的存储器地址。
  - 2) DB7~DB0: 8 位双向数据线。DB7~DB0 的作用有两种：第一种是在 8237A 空闲时，提供 CPU 访问 8237A 寄存器的数据通道；第二种是在进行 DMA 操作时，读周期经 DB7~DB0 线把源存储器的数送入暂存寄存器，在写周期再把暂存寄存器中的数据经 DB7~DB0 传送到目的存储器。
  - 3) DACK1: DMA 响应信号，输出，高电平有效。该信号是由 8237A 控制器发给通道 1 的回答信号，也就是说，简化后的 8237A 只有通道 1 可以申请 DMA 服务。
  - 4) MEMR: 存储器读信号，输出，三态，低电平有效。在 DMA 操作时，作为从选定的存储单元读出数据的控制信号。
  - 5) MEMW: 存储器写信号，输出，三态，低电平有效。在 DMA 操作时，作为向选定的存储单元写入数据的控制信号。
  - 6) IOR: I/O 读信号，双向，三态，低电平有效。在 DMA 控制器空闲时，由 CPU 发来，表示 CPU 读取 8237A 内部寄存器，该信号为输入。在 DMA 传送时，DMA 控制器处于工作状态，由 8237A 发给 I/O 设备的读控制信号，表示从 IO 设备读数据送到内存单元，该信号为输出。
  - 7) IOW: I/O 写信号，双向，三态，低电平有效。在 DMA 控制器空闲时，由 CPU 向 8237A 发来的写信号，数据从 CPU 写入 8237A 内部寄存器，该信号为输入。在 DMA 控制器工作时，是 8237A

控制器发给 I/O 设备的写信号，数据从内存写入 I/O 设备，该信号为输出。

- 8) DREQ1: DMA 请求信号, 输入, 上升沿有效。它是连接到通道 1 的外设, 向 DMA 控制器请求 DMA 服务的请求信号, 上升沿将触发通道 1 的 DMA 传送请求。
- 9) HRQ: 请求占用总线信号, 输出, 高电平有效。该信号是 DMA 控制器接到通道 1 的请求后, 且通道 1 请求未被屏蔽的情况下, DMA 控制器向 CPU 发出请求占用总线的信号。
- 10) HLDA: 同意占用总线信号, 输入, 高电平有效, 是 CPU 发给 DMA 控制器, 同意 DMA 控制器占用总线控制请求的回答信号。8237A 接收到 HLDA 后, 即可进行 DMA 传送。
- 11) CS: 片选信号, 输入, 低电平有效。当该信号有效时, CPU 可向 8237A 输出工作方式字和命令字, 或读入状态。
- 12) RESET: 复位信号, 输入, 高电平有效。当芯片被复位时, 操作方式命令字寄存器被清零, 8237A 进入空闲状态, 即不作为 DMA 控制器, 仅作为一个 I/O 设备。
- 13) CLK: 时钟信号, 输入, 用于控制芯片内部定时和数据传送速率。

3. 打开 8237A 模块, 熟悉 8237A 的内部电路及主要模块。如表 6-1 所示:

模块名称	模块原理图文件	模块功能
write_read	Write_read.dlsche	寄存器读写控制模块, 产生 8237A 内部各个寄存器的读写控制信号。
regs	regs.dlsche	公共寄存器模块, 包含 8237A 的操作方式命令字寄存器和暂存寄存器。
channel_0	channel.dlsche	通道 0, 该通道只有在存储器与存储器之间的数据传送时才使用到, 用于存放源地址。而目的地址和传送字节数存入通道 1 中。
channel_1	channel.dlsche	通道 1, 简化后的 8237A 只有通道 1 可以申请 DMA 传送服务。
fsm	fsm.dlsche	DMA 周期状态转换模块, 提供芯片处于不同周期的控制信号。

表 6-1:8237A 内部的主要模块

## 2.2 任务 (二)

使用 8237ADMA 控制器, 将存储器 RAM 中的代码块从地址 B 处复制到地址 A 处, 也就是存储器到存储器之间的 DMA 传送。实验步骤如下:

1. 打开项目下的微指令源程序文件 8086/rom.asm。
2. 在文件的第 642 行, 微指令 holdDMA 就是用来查询 8237A 发出的 DMA 请求的。8237A 发出的 DMA 请求信号通过 8086 的管脚 HOLD 输入。8086 微处理在执行完一条指令后, 通过微指令 hold DMA 查询 HOLD 管脚输入信号, 若 HOLD=1, 高电平, 则表示有 DMA 请求, 那么 8086 的 uPC 计数器停止计数, 保持 ROM 输出控制信号不变, 停止微指令的执行, 让出数据总线和地址总线, 同时发出 HLDA 信号, 通知 8237A 可以进行 DMA 传送操作了。DMA 传送结束后, HOLD 变为低电平, uPC 计数器继续计数, 从而继续运行程序。8086 停止运行的含义是, PC 和 uPC 均不随时钟改变, 直到 HOLD 变为无效为止, 从而保证 DMA 传送结束后仍然能继续运行被停止的指令。8086 让出总线的含义是, 在 DMA 传送期间, 8086 通过控制信号, 使得所有向总线输出数据或地址的三态门关闭, 将 CPU 的内部寄存器与总线隔离, 使得 8237A 能使用总线进行数据传送。请读者结合 CU 控制单元的内部电路加以理解。

注意: 这里有两点需要说明。首先, 为了简化 8086 的设计, 仅仅在一处添加了微指令 hold DMA, 用于检测 DMA 请求, 而在实际的处理器中, 应该在每条指令的末尾都检测 DMA 请求, 并且检测 DMA 请求都应该在检测硬中断之前。其次, 为了简化 8086 的设计, 当检测到 DMA 时, 8086 停止运行了, PC 和 uPC 均不随时钟改变, 而在实际的处理器中, 处理器是不会停止运行的, 而是会转去做其它的工作。

3. 打开项目下的汇编程序源文件 MEM/ram.asm, 结合注释, 理解程序功能。
4. 打开原理图文件 top.dlsche, 分析 8237A 是如何与 8086 微机系统连接起来的。
5. 打开 MEM 模块, 为了实现存储器到存储器之间的 DMA 传送, MEM 模块增加的 MEMW 和 MEMR 输入信号。在进行 DMA 传送时, MEMW=0 使能 RAM 的写, 并将数据写入 RAM, 而 MEMR=0 打开存储器输出门, 将存储数据输出到指定端口。
6. 由于 8237A 使用低四位地址 A0~A3 编码端口, 可使用 0x80~0x8F 作为 8237A 的端口地址, 因此 8237A 的片选信号由 CS0 与 CS1 组合得到。

下面, 通过仿真, 实现存储器到存储器之间的数据传送。步骤如下:

1. 打开原理图文件 top.dlsche, 启动仿真。
2. 通过按键 R 进行复位, 通过按键 S 切换为手动单步时钟运行模式。
3. 程序开始部分的作用是为了初始化 8237A 控制器。请读者依次定位相关寄存器或计数器, 然后通过按键 C 单步运行, 观察它们是如何写入数据的。各个寄存器和计数器定位如下:
  - ① 通道 0 的工作方式寄存器 MR 存在于 8237A 的内部模块 channel\_0 中;
  - ② Channel\_0 中的器件 C 和 A 分别是通道 0 的当前字节计数器和当前地址计数器。
  - ③ 打开 8237A 内部模块 channel\_1, 可以分别找到通道 1 的工作方式寄存器 MR、当前字节计数器 C 和当前地址计数器 A。
  - ④ 操作方式命令字寄存器 CW 存在于 8237A 的内部模块 regs 中, 它属于所有通道的公用寄存器。

当 8237A 初始化完成后, 开始执行主程序的第一条指令 cmpal, 0x15。在这条指令的最后, CPU 通过微指令 hold DMA 查询当前是否有 DMA 传送请求。下面, 我们测试没有 DMA 请求的情况下, 程序的执行情况:

1. 观察此时的 8237A 输出接口 HRQ 为低电平, 它与 8086 的 DMA 请求输入端 HOLD 相连, 说明此时没有 DMA 请求。8086 执行完 holdDMA 微指令后不会停止, 而应该继续执行下一条微指令。
2. 单步运行 cmpal, 0x15 对应的微指令, 运行到 holdDMA 时停下。双击打开 8086 内部的控制单元 CU, 我们可以看到 uPC 微程序计数器, uPC 实际上是一个 16 位寄存器, 它的输入使能端 EN=0, 说明 uPC 会随着时钟上升沿的到来继续写入微指令地址, 从 ROM 取出后续微指令继续执行, 即 CPU 不会停止指令执行。进一步可以发现, uPC 的使能输入 EN 来自于 HOLD 和 CHECK\_HODL 组合得到, 由于 HODL 为低电平, 即没有 DMA 请求, 所以 EN 使能, 继续运行程序。
3. 为了验证上述的分析, 我们通过按键 C 输入单步时钟, 观察 uCP 的变化, 从源代码窗口 2 可以看到, uCP 指向 holdDMA 的后一条微指令。证明, CPU 没有停止运行。
4. 继续单步运行下一条指令。
5. 结束仿真。

下面, 我们测试有 DMA 请求时 CPU 的运行情况。

1. 打开原理图文件 top.dlsche, 启动仿真, 复位后将时钟切换为自动时钟, 自动运行 8237A 初始化程序。当运行到主程序指令 cmpal, 0x15 时, 在查询 DMA 前切换为单步时钟。可以看到, 此时 8237A 的 DMA 请求输出端 HRQ 为低电平。
2. 双击打开 8237A 内部电路, 再双击 channel\_1 打开通道 1 模块, 找到 DMA 请求触发器 D。可以看到这是一个 SR 触发器, 它的置位端 S 始终为高电平, 时钟输入端 CLK 接外部输入 DREQ, 这是外部的请求输入。
3. 通过键盘按键 A 输入一个上升沿, 触发器 D 输出置位为高电平。通道 1 于是发出 DMA 请求。从这里可以看到, 通常是由外设向 8237A 申请 DMA 传送服务, 再由 8237A 通过 HOLD 管脚向 CPU 申请总线占用。
4. 返回顶层电路, 可以看到按键 A 控制的信号源与通道 1 的 DMA 请求输入 DREQ1 相连。

## 5. 观察 CPU 的 HOLD 输入高电平，说明有 DMA 请求。

到此，我们使 CPU 在查询 HOLD 管脚之前将 HOLD 置位为高电平。

1. 下面继续单步运行，当运行到微指令 hold DMA 时停下。
2. 打开 8086 的内部控制单元 CU，可以看到，uPC 的使能端 EN=1，使能无效，禁止 uPC 写入，从而使 CPU 运行停止。同时，可看到 HLDA 输出高电平，以响应 DMA 占用总线的请求。
3. 观察此时所有由 CPU 控制的向总线输出数据或地址的三态门，它们均被关闭，即 CPU 让出数据总线 DBUS 和地址总线 ABUS。
4. CPU 让出总线后，接下来就是 DMA 传送过程了。从 ram.asm 程序可知，8237A 将通过总线，将程序块 block2 传送到程序块 block1 所占的存储空间，总共传送 6 个字节，即两条指令。
5. 手动输入一个单步时钟，在源代码窗口可以看到蓝色箭头都没有移动，说明 CPU 停止运行程序了。

经过以上步骤，8237A 开始控制总线进行 DMA 传送。首先，我们需要知道存储器到存储器传送的原理。存储器到存储器之间的数据传送并不是指两个不同存储器之间的数据传送，而是指一个存储器内不同地址单元之间的数据传送。它的实现原理是，将源地址保存到通道 0 的当前地址计数器中，而将目的地址保存到通道 1 的当前地址计数器中，传送的字节数保存到通道 1 的当前字节计数器中。在传送时，首先输出通道 0 的源地址到地址总线 ABUS 上，8237A 发出读存储器的信号 MEMR，将存储器 RAM 指定地址单元的内容读出并传送到数据总线 DBUS，与此同时，8237A 的 regs 模块内的暂存寄存器 STR 写使能，下个时钟上升沿到来时，就将从存储器读出的内容写入 STR 暂存。下个时钟到来时，8237A 将通道 1 的目的地址传送到地址总线 ABUS 上，同时将 STR 暂存的内容传送到数据总线 DBUS 上，并发出写存储器 RAM 的信号 MEMW，当下个时钟上升沿到来时，将数据写入 RAM 的指定单元中，从而完成一个字节的传送。然后，通道 0 的源地址加/减 1，指向下一个传送字节，通道 0 的目的地址加/减 1，指向下一个接收单元，传送字节数减 1。依次类推，逐个字节传送，当所有字节数传送完毕后，即通道 1 的当前字节计数器减为-1 时，清除 DMA 请求信号 RHQ，释放总线，CPU 获取总线使用权，继续运行程序。

1. 打开 8237A 内部的模块 fsm，可以看到此时状态机处于状态 S3 (S3=0)。对应的有效输出信号为：
  - ① A0\_EN=0，通道 0 地址输出门打开，源地址传送到地址总线 ABUS 上。打开通道 0 的内部电路，可以看到 AB[7..0] 端口输出当前地址，返回 8237A 可以看到，A0~A7 上的地址就是来自于通道 0 的 AB[7..0]。我们注意到，ABUS 上的地址为 0x3D，它正是 ram.asm 中需要传送的代码块 block2 的第一条指令的第一个字节地址。在存储器 RAM 窗口可以看到当前地址。
  - ② MEMR=0，存储器输出门打开，从而将地址指定单元数据传送到数据总线 DBUS 上。返回顶层电路，打开存储器模块 MEM，可以看到在 MEMR 的使能下，存储器 RAM 读出数据传送到数据总线 DBUS 上了。我们注意到，存储器输出的字节内容为 0x01，这正是 block2 的第一条指令的第一个字节。在存储器 RAM 窗口可以看到当前字节内容。
  - ③ WR\_TR=0。该信号使能 8237A 内部模块 regs 内部的暂存寄存器 STR，目的是将从存储器读出的字节暂时写入该寄存器，以便将其在写回传送目的地址指定的存储器单元中。打开 8237A 内部的 regs 模块，可以看到暂存寄存器 STR 的时钟输入端接的正是 WR\_TR 信号，而其输入数据来源于 DBUS，内容为 0x01，当 WR\_TR 变为高电平后，就写入。
2. 输入单步时钟 C，可以看到 STR 写入了 0x01。打开 fsm 模块，可以看到状态机处于 S4 状态，输出的有效控制信号为：
  - ① A1\_EN=0，通道 1 地址输出门打开，目的地址传送到地址总线 ABUS 上。打开通道 1 的内部电路，可以看到 AB[7..0] 端口输出当前地址，返回 8237A 可以看到，A0~A7 上的地址就是来自于通道 1 的 AB[7..0]。我们注意到，ABUS 上的地址为 0x37，它正是 ram.asm 中代码块 block2 需要传送到的第一个目的地址。在存储器 RAM 窗口可以看到当前地址。
  - ② RD\_TR=0，读出暂存寄存器 STR 的字节，输出到数据总线 DBUS 上。打开 8237A 内部的 regs 模块，

可以看到暂存寄存器 STR 的输出三态门使能，字节 0x01 输出到 DBUS 上。

- ③ MEMW=0，存储器 RAM 写使能，从而将 DBUS 上的数据写入目的地址指定单元。在存储器 RAM 窗口可以看到从暂存寄存器读出的字节 0x01 确实写入了目的地址指定单元 0x37 中。写存储器的操作在进入 S4 状态就完成了，这是因为按键 C 输入的单步时钟先有下降沿，然后有一个上升沿，在下降沿时，状态从 S3 进入 S4，同时输出有效信号 MEMW，它使能存储器 RAM 的写控制，在按键抬起的时候，来一个上升沿，触发存储器写入数据。

- ④ 从存储器 RAM 窗口可以看到写入后的结果。

- 经过以上步骤，实现了存储器 RAM 的源地址 0x3D 指定字节传送到目的地址 0x37 指定位置。打开 8237A 内部模块 fsm，注意到当前 CP 输出为低电平。分别打开通道 0 和通道 1，可以看到，CP 是通道内部当前字节数和当前地址计数器的时钟，当 CP 输入上升沿时，当前字节数减 1，而当前源地址和目的地址均加 1。
- 打开 fsm，观察 CP，通过按键 C 输入单步时钟后，可以看到，从状态 S4 进入状态 S3，同时 CP 变为高电平。分别打开通道 0 和通道 1，可以看到通道 0 的源地址由 0x3D 加 1 变为 0x3E。而通道 1 中的当前字节数由 5 减为 4，目的地址由 0x37 加 1 变为 0x38。
- 到此，其实已经开始下一个字节的传送了，传送过程与上一个字节传送一致。请读者依次输入单步时钟，同时观察通道 1 中的当前字节数变化情况，以及存储器 RAM 窗口显示的写入字节。当通道 1 的传送字节减为 -1 时，表示所有字节传送完成。

所有字节传送结束后，也就是 block2 处的两条指令传送到了 block1 处。那么，我们从存储器 RAM 窗口，比较源地址块 0x3D~0x42 与目的地址块 0x37~0x3C 的内容，它们完全一致，说明传送成功了。DMA 传送结束后，RHQ 请求被清除，总线使用权上交给 CPU，CPU 从停机开始运行程序。需要注意的是，CPU 是在运行 block1 处的程序停机的，通过 DMA 传送操作，block1 处的两条指令已经被替换了。我们观察替换后，CPU 运行的情况。

- 通过按键 C 输入单步时钟，逐步运行，可以看到虽然源代码窗口 1 中蓝色指向的是 block1 处的第一条指令 movbl, 0。但是源代码窗口 2 中蓝色箭头指向的确是 subreg, imm 对应的微程序，这正是 block2 处的指令 subal, 3 对应的微程序。
- 查看此时 al=0x15，单步运行，直到该条指令执行完毕，可以看到 al 由 0x15 减 3 后变为 0x12。
- 继续执行，下一条指令正是 block2 处的第二条指令，adcal, 2，执行结果是 al+2=0x14。
- 到此，可继续运行后续指令或结束仿真。

通过上述所有步骤，我们通过 8237A 将存储器 ram 的代码块 block2 移到到代码块 block1 处，并完成了 DMA 请求响应和结束的整个过程。相信读者已经对 DMA 工作原理有了深刻的理解。

### 2.3 任务（三）

在任务二中，我们完成了存储器到存储器之间的 DMA 传送，下面请读者设计一个写传送或读传送电路。

- 读传送，将存储器 RAM 中地址单元 0x3D 处的字节传送到一个外部寄存器 r 中。
- 写传送，将外部寄存器 r 中的字节传送到 RAM 地址单元 0x3D 中。
- 使用通道 1，通过 DREQ1 发出 DMA 传送请求。

# 实验 7 数/模转换器 DAC0832

实验性质：验证+设计

建议学时：2 学时

## 一、实验目的

- 熟悉 DAC0832 芯片的内部电路结构以及工作原理。
- 掌握 DAC0832 芯片编程与应用。

## 二、预备知识

DAC0832 数/模转换器的内部结构如图 7-1 所示。

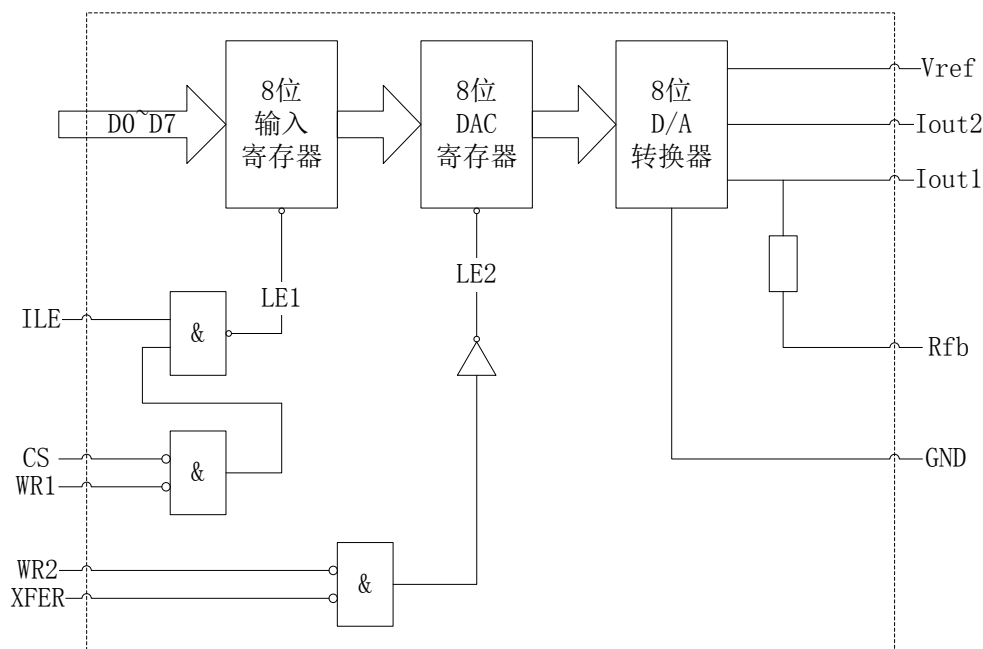


图 7-1：DAC0832 内部结构示意图

DAC0832 内部有两级锁存器：

第一级锁存器是一个 8 位的输入寄存器，由锁存控制信号 ILE、CS、WR<sub>i</sub> 共同控制。当 ILE=1，CS=WR<sub>i</sub>=0（由 OUT 指令产生）时，LE1=0，输入寄存器的输出随输入而变化。接着，WR<sub>i</sub> 由低电平变为高电平时，LE1=1，则数据被锁存到输入寄存器，其输出不再随输入变化；

第二级锁存器是一个 8 位的 DAC 寄存器，它的锁存控制信号为 XFER，同理，LE2=0 时，第一级锁存器的数据传送到 DAC 寄存器的输出端，当 LE2=1 时，输入寄存器的信息被锁存到 DAC 寄存器中。同时，转换器开始工作，I<sub>OUT1</sub> 和 I<sub>OUT2</sub> 端输出电流。

## 三、实验内容

请读者按照下面方法之一在本地创建一个项目，用于完成本次实验：

### 方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建个人项目，然后使用 Dream Logic 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

### 方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台,就需要使用 Dream Logic 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中,实验模板的 URL 为

<https://www.codecode.net/engintime/Dream-Logic/Project-Template/Microcomputer/Lab007.git>

### 3.1 任务(一)

请读者按下列步骤熟悉 DAC0832 的管脚和内部电路。

1. 打开项目下的原理图 DAC0832/DAC0832.dlsche。
2. 从图中找出以下端口(引脚):
  - 1) D7~D0: 8 位输入数据线。
  - 2) ILE: 输入锁存允许信号,高电平有效。用来控制 8 位输入寄存器的数据是否能被锁存的控制信号之一。
  - 3) CS: 片选信号,低电平有效。它和 ILE 信号一起用于控制 WR1 信号是否起作用。
  - 4) WR<sub>1</sub>: 写信号 1,低电平有效。在 ILE 和 CS 有效时,用它将输入的 8 位数字锁存于输入寄存器中。ILE、CS、WR<sub>1</sub> 是 8 位输入寄存器工作的三个控制信号。
  - 5) WR<sub>2</sub>: 写信号 2,低电平有效。在 XFER 有效时,用它将输入寄存器中的数字传送到 8 位 DAC 寄存器中。
  - 6) XFER: 传送控制信号,低电平有效。用它和 WR<sub>2</sub> 一起控制 8 位 DAC 寄存器是否工作。
  - 7) I<sub>OUT1</sub>: DAC 电流输出 1,它是逻辑电平为 1 的各位输入电流之和。
  - 8) I<sub>OUT2</sub>: DAC 电流输出 2,它是逻辑电平为 0 的各位输入电流之和。
  - 9) R<sub>fb</sub>: 反馈电阻,该电阻被制作在芯片内,用作运算放大器的反馈电阻。
  - 10) V<sub>REF</sub>: 基准电压输入。一般在 -10V~+10V 范围内,有外电路提供。
  - 11) VCC: 逻辑电源。实验中可悬空。
  - 12) AGND: 模拟地。为芯片模拟电路接地点。

从图中可以看出,从 D0~D7 输入的数字信号需要经过两个锁存器 A 和 B,才能输入 DAC 进行数模转换,最终将转换后的电流通过 Iout1 和 Iout2 输出。根据通过两个锁存器的方法,形成 DAC0832 的三种工作方式:

1. 双缓冲方式: 数据先后通过两个寄存器锁存后再送入 D/A 转换电路,执行两次写操作才能完成一次 D/A 转换。这种方式特别适用于要求同时输出多个模拟量的场合。
2. 单缓冲方式: 此时两个寄存器之一处于直通状态,输入数据只经过一次锁存,然后送入 D/A 转换电路。这种方式下,只需要执行一次写操作,即可完成 D/A 转换,可以提高 DAC 的数据吞吐量。
3. 直通方式: 此时两个寄存器都处于开通状态,即 ILE、CS、WR<sub>1</sub>、WR<sub>2</sub> 和 XFER 满足有效电平状态,输入数据直接送入 D/A 转换电路进行转换。这种方式可用于一些不采用微机的控制系统中。

### 3.2 任务(二)

通过仿真,熟悉 DAC0832 的管脚功能。实验步骤如下:

1. 打开项目下的原理图 DAC0832\_test.dlsche,并熟悉 DAC0832 的内部电路。
2. 启动仿真。
3. 设置 D0~D7 的输入为 0。
4. 通过按键 3,将 ILE 设为高电平,输入锁存允许。
5. 通过按键 1、2、4、5 将 CS、WR<sub>1</sub>、WR<sub>2</sub> 和 XFER 设为低电平。
6. 双击 DAC0832 进入内部,可看到 A、B 两个锁存器均使能,输入数据 D0~D7 均写入了 A、B 中。
7. 返回顶层电路,观察电压表的值为 0,这里显示的就是将输入的数字信号转换后的模拟电压值。
8. 将 D0~D7 依次修改为 01,0f,5f,af,ff,观察转换后的电压值。通过数模转换公式检验结果的正确性,公式如下所示。其中,d 表示输入数字信号 D7~D0 的十进制值,V<sub>ref</sub> 为基准电压值,V 为数

模转换输出电压值。

$$V = \frac{V_{\text{ref}} * d}{256}$$

9. 将 ILE 设为低电平, 修改 D0~D7 的输入, DAC0832 内部的锁存器 A、B 会随 D0~D7 改变吗? 电压表显示值 V 会改变吗? 为什么?
10. 仿照测试 ILE 管脚功能的方法, 分别测试 CS、WR1、WR2 和 XFER 对数模转换的控制作用。
11. 选中电压源 V1, 在属性窗口中将其电压值修改为-5V, 数字信号输入为 5f, 观察转换输出电压值。
12. 结束仿真。

### 3.3 任务（三）

通过编程, 控制 DAC0832 输出正向锯齿波形。实验步骤如下:

1. 打开项目下的原理图 top.dlsche, 注意观察 DAC0832 是如何与微机系统连接的。
2. 打开汇编源程序文件 MEM/ram.asm, 理解代码功能。
3. 打开 top.dlsche, 启动仿真。
4. 通过按键 S, 将时钟切换为单步时钟。
5. 通过按键 R, 进行复位。
6. 单步运行程序, 当执行“outdl, al”指令的微指令“pathrs, <mar>”时, 可看到 DAC0832 的片选信号 CS=0, 写信号 WR1=0, 其内部锁存器 A、B 均写入了数据 D7~D0。此时电压表显示的值就是数模转换后的输出电压。
7. 继续仿真, 将仿真结果填入下表 7-1 中。

数模转换的数字信号值 al	转换输出模拟电压值 V
0	
0x40	
0x80	
0xc0	
0	
0x40	
0x80	
...	...

表 7-1: 数模转换表

8. 结束仿真。
9. 以 AL 为横坐标, V 为纵坐标, 绘制数模转换波形图。这是一个正向锯齿波形。

### 3.4 任务（四）

修改任务三中的程序, 实现将存储器 RAM 的 0x10~0x15 地址单元的内容转换为模拟电压输出。

要求:

1. 修改项目下的 MEM/ram.asm 程序, 实现将存储器 RAM 的 0x10~0x15 地址单元的内容转换为模拟电压输出。。
2. 将 DAC0832 的输入基准电压 VREF 分别设置为 2V、4V、7V。记录每一种基准电压下的转换结果。基准电压对转换结果有影响吗? 为什么?

## 四、思考与练习

1. DAC0832 有哪几种工作方式? 每种工作方式适用于什么场合? 每种方式是用什么方法产生的?



# 实验 8 模/数转换器 ADC0809

实验性质：验证+设计

建议学时：2 学时

## 一、实验目的

- 了解 ADC0809 芯片的内部电路结构以及工作原理。
- 掌握 ADC0809 芯片与 CPU 的连接方法，并能对其编程，通过程序来使用其 A/D 转换功能。

## 二、预备知识

### 2.1 逐次逼近型 A/D 转换原理

逐次逼近型 A/D 转换器采用的是一种反馈比较型电路结构。它的构思是这样的：取一个数字量加到 D/A 转换器上，于是得到一个对应的输出模拟电压。将这个模拟电压和输入的模拟电压信号相比较。如果两者不相等，则调整所取的数字量，直到两个模拟电压最相近为止，最后所取的这个数字量就是所求的转换结果。

在进行逐次逼近型 A/D 转换时，先选取 8 位数字信号的最高位置 1，即 1000 0000，若转换后的模拟电压大于输入电压，说明数字过大，则最高位的 1 应去掉；若转换后的模拟电压小于输入电压，则说明数字还不够大，这个最高位的 1 应予保留。然后，再按同样的方法将次高位置 1，通过比较决定这一位的 1 是否保留。这样逐位比较下去，直到最低位比较完为止。这时，比较后的数码就是所求的数字量。

上述比较过程类似于使用天平称量一个未知重量的物体，使用的砝码一个比一个重量少一半。

### 2.2 ADC0809 的内部结构

ADC0809 是 CMOS 单片逐次逼近型 A/D 转换器，它的内部结构如下图 8-1 所示。

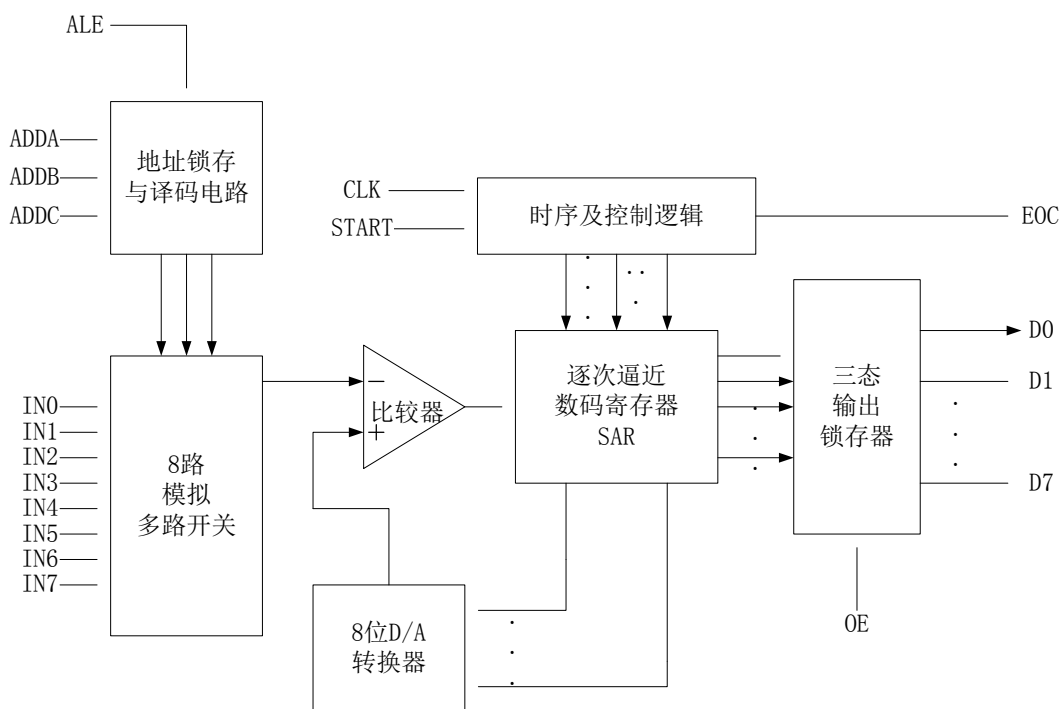


图 8-1：ADC0809 的内部结构图

### ◆ 8 路模拟多路开关

输入 8 路模拟信号，根据通道地址译码选择任意一路模拟输入进行 A/D 转换。

### ◆ 地址锁存与译码电路

用来选择模拟通道。通过 ADDA、ADDB、ADDC 三个地址译码，控制通道选择开关，接通某一路的模拟信号，输入到 DAC0809 比较器的输入端。

### ◆ 逐次逼近 A/D 转换器

该部分包括比较器、8 位倒 T 型 D/A 转换器、逐次逼近寄存器。把输入的模拟信号进行逐次逼近式 A/D 转换，转换结束后把转换数据从逐次逼近寄存器传送到 8 位锁存器。

### ◆ 三态输出锁存器

经 A/D 转换的数字量保存在 8 位锁存器中，当输出允许信号 OE 有效时（高电平有效），打开三态门，转换后的数字信号通过数据总线读入 CPU。

## 三、 实验内容

请读者按照下面方法之一在本地创建一个项目，用于完成本次实验：

### 方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建个人项目，然后使用 Dream Logic 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

### 方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，就需要使用 Dream Logic 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的 URL 为

<https://www.codecode.net/engintime/Dream-Logic/Project-Template/Microcomputer/Lab008.git>

### 3.1 任务（一）

请读者按下列步骤熟悉 ADC0809 的管脚和内部电路。

1. 打开项目下的原理图文件 ADC0809/ADC0809.dlsche。
2. 从原理图中找出以下端口（引脚）：
  - 1) D7~D0：输出数据线。
  - 2) IN0~IN7：8 通道模拟电压输入端，可连接 8 路模拟量输入。
  - 3) ADDA、ADDB、ADDC：通道地址选择，用于选择 8 路中的一路模拟输入电压进行 A/D 转换。ADDA 为最低位，ADDC 为最高位，这 3 个引脚上所加电平的编码为 000~111，分别对应于选通通道 IN0~IN7。
  - 4) ALE：通道地址锁存信号，用于锁存 ADDA~ADDC 端的地址输入，上升沿有效。
  - 5) START：启动转换信号输入端，下降沿有效。在启动信号的下降沿，启动 A/D 转换。
  - 6) EOC：转换结束状态信号。平时为高电平，当其正在转换时为低电平，转换结束时，它变为高电平。此信号可用于查询或者作为中断申请。
  - 7) OE：输出（读）允许（打开输出三态门）信号，高电平有效。在其有效期间，打开输出三态门，CPU 将转换后的 8 位数字量读入。
  - 8) CLK：时钟输入。驱动 A/D 转换。
  - 9) V<sub>REF</sub>：基准参考电压输入端。
3. 图中与通道地址选择 ADDA~ADDC 连接的是一个 8 位寄存器，可看到当 ALE 输入上升沿时，ADDA~ADDC 端的地址输入就会写入该寄存器，即通道地址锁存。
4. 锁存的通道地址通过 3-8 译码器输出 8 个选择信号，其中唯一有效的信号可以控制一路模拟信号输入，其他模拟输入信号被断开。此处使用到了 8 个数字开关，如下图 8-2 所示：

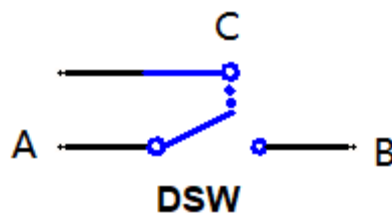


图 8-2: 数字开关

数字开关的工作原理是：当 C 端输入高电平时，数字开关闭合，A 和 B 导通，模拟信号即可从 A 传输到 B，否则，数字开关断开，A 和 B 不导通。

5. 被选择的模拟信号与 DAC 输出的模拟电压通过电压比较器 C 进行比较，如图 8-3 所示：

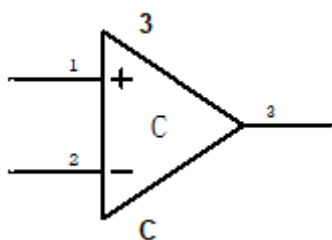


图 8-3: 电压比较器

电压比较器的工作原理是：当管脚 1 端输入的电压大于 2 端输入电压时，管脚 3 输出高电平；否则，管脚 3 输出低电平。

6. 打开模块 DAC，DAC 是一个倒 T 型电阻网络 D/A 转换器，负责将输入的 8 位数字信号转换为模拟电压输出。这个 DAC 模块与 DAC0832 中的 DAC 模块的原理是一样的。

### 3.2 任务（二）

通过仿真，测试 ADC0809 的功能。步骤如下：

1. 打开项目下的原理图 ADC0809\_test.dlsche，基准电压值使用 V1 提供的 5V，转换精度为  $5/256 = 0.01953125V$ 。
2. 启动仿真。
3. 设置 ADC0809 通道地址选择 ADDC~ADDA 的值为 0，选择模拟输入 IN0 进行模数转换。
4. 双击 ADC0809 进入内部电路，可以看到通道地址锁存器 ADR 还没有将 ADDC~ADDA 锁存。通过键盘按键 A，给管脚 ALE 输入一个上升沿，ADR 将 ADDC~ADDA 锁存，锁存地址经过译码后产生有效控制信号 S0（高电平），S0 控制数字开关 1 闭合，从而输入 IN0 信号。
5. 通过按键 S，给管脚 START 先输入一个高电平，再输入低电平，启动模数转换。
6. 通过按键 E，设置管脚 OE 输入高电平，将转换后的数字量输出。
7. 为了便于观察，在电路中有一个数模转换器 DAC0832，它将 ADC0809 转换后的数字量转换为模拟电压，通过电压表，观察电压值，将其与 IN0 端的电压值比较，它们之间的差值就是转换误差，记录下来。
8. 重复步骤 3~7，依次将模拟通道 IN1~IN7 的输入转换为数字量输出，并将输出的数字量转换成的模拟电压 V 与原始输入电压 INx 比较，计算模数转换误差的平均值。

修改基准电压，再次进行模数转换。

9. 将 ADC0809 的基准输入电压修改为 3V，再次将通道 IN0~IN7 的模拟电压转换为数字量，再将数字量转换为模拟电压，计算转换误差的平均值，将其与基准电压为 5V 时的转换误差平均值比较，哪一个误差小？为什么？
10. 结束仿真。

### 3.3 任务（三）

使用一片 ADC0809 模数转换芯片和一片 8255 并行接口芯片，通过与微机系统连接，实现一个自动采集模数转换器转换后的数字量的系统。

要求如下：

1. 打开项目下的原理图文件 top.dlsche，完成芯片电路连接。
2. 实现功能：依次将 ADC0809 的 8 路模拟输入信号 IN0~IN7 转换为数字量，并将数字量保存到存储器 RAM 的 0x90~0x97 单元中。
3. 功能测试代码已经给出，读者可直接打开项目下的源程序文件 MEM/ram.asm 查看。
4. 测试代码中包含了 ADC0809 和 8255 的端口地址信息，在连接芯片的时候，需要使用这些端口地址。
5. 当程序进入死循环后，说明转换结束。打开存储器 RAM 窗口查看 0x90~0x97 单元中的值。

## 四、思考与练习

1. ADC 中的转换结束信号（EOC）起什么作用？
2. 如果 ADC0809 与微机接口采用中断方式，试问 EOC 应如何与微处理器连接？

## 附录 A DM1000 八位模型机规格说明

DM1000 每一条微指令就是一组 32 位控制信号的编码。下表中列出了每一位控制信号名称及其含义。

表 A-1 DM1000 八位模型机中控制信号详情表

位置	缩写	含义
31		保留位
30		保留位
29	UPC_RESET_EN	微程序计数器 uPC（微程序地址寄存器）的复位信号，低电平有效。有效时，打开取指微指令地址输出门 IR_Fetch_gate，0 地址传送到 uPC 的输入端，当下个时钟上升沿到来时，0 写入 uPC，从而读出 ROM 中的取指微指令。
28	PC_ADD	程序计数器 PC 的加计数使能信号，高电平有效。有效时，时钟上升沿到来时，PC+1，指向下一个字节。 以 PC 为地址从存储器读取字节，若该字节是指令的第一个字节，它将被写入指令寄存器 IR 中；若该字节是指令的第二个字节，那么它是指令的操作数，不会写入指令寄存器 IR 中。
27	CSP_U\D	栈计数器 CSP 的加/减控制信号，CSP_U\D 为低电平时栈计数器做加法，对应于出栈；为高电平时栈计数器做减法，对应于入栈。
26	IA_REG_EN	中断向量地址寄存器 IA 的写使能信号，低电平有效。有效条件下，时钟上升沿到来时，数据总线上的中断号与固定偏移地址组合后的向量地址写入寄存器 IA 中。IA 中存放的是一个地址，该地址指向的存储单元存放的是一个中断服务程序的入口地址（第一条指令的地址）。
25	MEM_WR	主存储器 RAM 的写使能信号，低电平有效。写使能时，时钟上升沿到来时，主存储器 RAM 的输入数据 DBUS[7..0] 写入地址 ABUS[7..0] 指定存储单元中。
24	ROUT_REG_EN	输出寄存器 ROUT 的写使能信号，低电平有效。有效时，时钟上升沿到来时，数据总线上的数据写入输出寄存器 ROUT 中。
23	PC_A_GATE_EN	地址 PC 输出门 PC_gate 的使能信号，低电平有效。有效时，打开输出门 PC_gate，PC 值输出至地址总线 ABUS 上。
22	IREN	指令寄存器 IR 的写使能信号，低电平有效。有效条件下，时钟上升沿到来时，数据总线上的指令（指令的第一个字节）写入 IR 中。 在一条指令执行期间，指令寄存器 IR 保持不变，这样，指令执行期间就可以随时利用 IR 保存的指令信息。比如指令寄存器中的 IR0 和 IR1 两位用来寻址通用寄存器 R0~R3，而 IR2 和 IR3 两位用来编码跳转指令类型。在 DM1000 中，跳转指令分为无条件跳转指令 jmp，条件转移指令 jc 和 jz。
21		保留位
20	PC_LOAD_EN	PC_LOAD_EN 为低电平时，表示当前指令是跳转指令，结合指令的 IR2、IR3 以及进位标志 CF、零标志 ZF，得到 PC 置数信号 LOAD。 若 LOAD=0，则允许 PC 同步置数，当 PC 的时钟上升沿到来时，输入数据 DBUS[7..0] 加载到 PC 中，从而跳转到指定地址处执行程序。 若 LOAD=1，PC 不会加载输入端的数据，也就不会发生跳转。
19	MAR_REG_EN	地址寄存器 MAR 的写使能信号，低电平有效。有效时，时钟上升沿到来时，数据总线上的数据（地址）写入 MAR。MAR 是数据总线向地址总线传输地址的唯一单向通道。
18	MAR_GATE_EN	地址寄存器 MAR 的输出门 MAR_gate 的使能信号，低电平有效。有效时，

		MAR_gate 打开, 地址寄存器 MAR 保存的地址即可输出到地址总线 ABUS 上。
17	ASR_REG_EN	栈辅助寄存器 ASR 的写使能信号, 低电平有效。有效条件下, 时钟上升沿到来时, 数据总线上的数据写入 ASR 中。
16	SP_REG_EN	栈顶指针寄存器 SP 写使能信号, 低电平有效。有效条件下, 时钟上升沿到来时, 数据总线上的数据写入 SP 中。
15	CSP_LOAD	栈顶地址计数器 CSP 的预置信号, 低电平有效。有效时, 将栈顶指针寄存器 SP 的内容加载到 CSP 中, 用来对栈顶指针 SP 加 1 或减 1。 出栈时, SP 通过 CSP 做加 1 运算, 结果写回 SP 中; 入栈时, SP 通过 CSP 做减 1 运算, 结果写回 SP 中。因此, CSP 的作用就是为了出入栈时, 更新栈顶指针 SP。
14	REG_WR	通用寄存器 R0~R3 的写信号, 低电平有效。有效时, 与指令寄存器中的位 IRO 和 IR1 配合, 通过 2-4 译码器, 选择 R0~R3 中的某一个寄存器写使能, 当下个时钟上升沿到来时, 就可以将数据写入被使能的寄存器中。  寄存器 R 的选择由指令的最低两位 IR <sub>2</sub> IR <sub>1</sub> 决定, IR <sub>2</sub> IR <sub>1</sub> 作为 2-4 译码器的输入, 当 IR <sub>2</sub> IR <sub>1</sub> 为 00~11 时, 分别选中 R0~R3, 从而可对选中的寄存器进行读或写操作。
13	CN	决定运算器 ALU 是否带进位移位, CN=1 带进位, CN=0 不带进位
12	FLAG_REG_EN	标志寄存器 FLAG 的写使能信号, 低电平有效。有效条件下, 时钟上升沿到来时, 将运算结果标志位写入 FLAG 中。DM1000 中的运算标志主要包含进位标志 CF, 零标志 ZF, 这两个标志分别为进位转移指令 jc 和为零则转移指令 jz 的跳转条件。
11	X3	四位组合译码来产生三态输出门的使能信号, 这些三态门的输出端与数据总线相连, 它们是互斥的, 即任何时刻只能有一个三态门打开, 其余的关闭, 从而允许该三态门输出数据到总线上。避免多部件同时向总线发送数据造成的信号冲突。
10	X2	
9	X1	
8	X0	
7	W_REG_EN	工作寄存器 W 的写使能信号, 低电平有效。有效条件下, 时钟上升沿到来时, 数据总线上的数据写入 W 中。
6	A_REG_EN	累加寄存器 A 的写使能信号, 低电平有效。有效条件下, 数据总线上的数据写入 A 中。
5	M	运算类型控制信号: M=0, ALU 做算术运算; M=1, ALU 做逻辑运算
4	C	ALU 进位输入信号, 加运算时, 若 C=0, 则表示最低位有进位; 减运算时, 若 C=1, 表示最低位有借位
3	S3	ALU 的运算方式控制信号, 决定 ALU 的运算类型
2	S2	
1	S1	
0	S0	

所有向数据总线 DBUS 输出数据的部件, 都需要通过三态门与总线连接, 任何时刻只允许一个三态门打开, 其余三态门关闭。打开的三态门对应的部件就可以将其数据通过三态门输出到总线上了。下表列出了数据总线上的所有三态输入门:

表 A-2 数据总线上的输入三态门

X3	X2	X1	X0	使能信号	三态门名称	说明
0	0	0	0	RIN_GATE_EN	RIN_gate	当 RIN_GATE_EN 为低电平时, 输出门 RIN_gate 打开, 输入寄存器 RIN 的内容通

						过该三态门传送到数据总线 DBUS 上。 当 RIN_GATE_EN 为高电平时,三态门关闭。
0	0	0	1	IA_GATE_EN	IA_gate	中断向量地址寄存器输出门, 使能时 IA 的内容输出到 DBUS 上
0	0	1	0	SP_GATE_EN	SP_gate	栈指针寄存器输出门, 使能时 SP 的内容输出到 DBUS 上
0	0	1	1	PC_D_GATE_EN	PC_D_gate	PC 向数据总线输出门, 使能时, PC 的内容输出到 DBUS 上
0	1	0	0	D_GATE_EN	D_gate	ALU 运算结果输出门, 使能时, ALU 运算结果输出到 DBUS 上
0	1	0	1	R_GATE_EN	R_gate	逻辑右移一位输出门, ALU 右移一位后的结果输出到 DBUS 上
0	1	1	0	L_GATE_EN	L_gate	逻辑左移一位输出门, 使能时, ALU 左移一位后的结果输出到 DBUS 上
0	1	1	1	ASR_GATE_EN	ASR_gate	辅助寄存器输出门, 使能时, ASR 的内容输出到 DBUS 上
1	0	0	0	CSP_GATE_EN	CSP_gate	栈顶地址计数器输出门, 使能时, CSP 的内容输出到 DBUS 上
1	0	0	1	MEM_GATE_EN	MEM_gate	存储器输出门, 使能时, 从主存储器读出的数据输出到 DBUS 上
1	0	1	0	REG_READ	IR <sub>i</sub> IR <sub>0</sub> 为 00~11 时, 分别对应: R0_gate, R1_gate, R2_gate, R3_gate	被选中寄存器的内容输出到 DBUS 上
1	0	1	1	—	保留	—
1	1	0	0	—	保留	—
1	1	0	1	—	保留	—
1	1	1	0	—	保留	—
1	1	1	1	—	保留	—

表 A-3 DM1000 中的寄存器及其控制信号

名称	说明	控制信号说明	输出三态门
A	累加寄存器	A_REG_EN: A 累加寄存器写入使能信号, 低电平有效	无
W	工作寄存器	W_REG_EN: W 工作寄存器写入使能信号, 低电平有效	无
R0~R3	通用寄存器 R0~R3	REG_READ: 寄存器读信号, 低电平有效; 有效时, 被选中寄存器的内容通过打开的三态门输出到数据总线 DBUS 上  REG_WR: 寄存器写信号, 低电平有效; 有效时, 数据总线上的数据写入被选中的寄存器  IR1, IR0 是指令的最后两位, 作为通用寄存器的选择信号, 当 IR1IR0 为 00~11 时分别选中寄存器 R0~R3	R0_gate R1_gate R2_gate R3_gate
PC	Program Counter 程序计数器	PC_A_GATE_EN: PC 向地址总线输出三态门使能信号, 低电平有效; 有效时, 打开三态门 PC_gate, PC 的内容输出到地址总线 ABUS 上。  PC_D_GATE_EN: PC 向数据总线输出三态门使能信号, 低电平有效; 有效时, 打开三态门 PC_D_gate, PC 内容输出到数据总线 DBUS 上。	PC_D_gate PC_gate
MAR	Memory address register 主存地址寄存器	MAR_REG_EN: MAR 寄存器写入使能信号, 低电平有效; 用于接收和保存数据总线传输的地址数据  MAR_GATE_EN: MAR 输出三态门使能信号, 低电平有效; 有效时, 打开三态门 MAR_gate, 将 MAR 内容输出到 ABUS 上, 作为访问主存地址	MAR_gate
SP	栈指针寄存器	SP_REG_EN: SP 寄存器写入使能信号, 低电平有效  SP_GATE_EN: SP 寄存器输出三态门使能信号, 低电平有效; 有效时, 打开三态门 SP_gate, SP 中的内容 (主存地址) 输出到数据总线上	SP_gate
IR	Instruction Register 指令寄存器	IREN: IR 寄存器写入使能信号, 低电平有效	
RIN	输入寄存器	RIN_GATE_EN: RIN 寄存器输出三态门使能信号, 低电平有效; 有效时, 打开三态门 RIN_gate, RIN 的内容输出到数据总线 DBUS 上	RIN_gate
ROUT	输出寄存器	ROUT_REG_EN: ROUT 寄存器写入使能信号, 低电平有效	
IA	中断向量寄存器	IA_REG_EN: IA 寄存器写入使能信号, 低电平有效 IA_GATE_EN: IA 寄存器输出三态门使能信号, 低电	IA_gate



		平有效；有效时，打开三态门 IA_gate，IA 中的中断向量输出到数据总线上	
ASR	栈辅助寄存器，用于暂存数据或地址	ASR_REG_EN: ASR 寄存器写入使能信号，低电平有效  ASR_GATE_EN: ASR 寄存器输出三态门使能信号，低电平有效，有效时，打开三态门 ASR_gate，ASR 的内容输出到数据总线上	ASR_gate
CSP	栈加减计数器，将栈顶指针寄存器 SP 中的地址加 1 或减 1	CSP_LOAD: 将 SP 中的内容加载到 CSP 中  CSP_UD: 低电平，计数器做加法；高电平，计数器做减法  CSP_GATE_EN: CSP 输出三态门使能，低电平有效；有效时，打开三态门 CSP_gate，CSP 的内容输出到数据总线 DBUS 上	CSP_gate

表 A-4 DM1000 指令集

指令形式	机器码 1	机器码 2	简要说明
ADD A, R?	000100xx		将寄存器 R? 的值加入累加寄存器 A 中
ADD A, [R?]	000101xx		将寄存器 R? 中地址指向的主存储器单元中的值加入累加器 A 中
ADD A, MM	000110xx	MM	将地址 MM 指向的存储器单元中的值加入累加器 A 中
ADD A, immediate	000111xx	immediate	将立即数 immediate 加入累加器 A 中
ADC A, R?	001000xx		将寄存器 R? 的值加入累加寄存器 A 中，带进位
ADC A, [R?]	001001xx		将寄存器 R? 中地址指向的主存储器单元中的值加入累加器 A 中，带进位
ADC A, MM	001010xx	MM	将地址 MM 指向的存储器单元中的值加入累加器 A 中，带进位
ADC A, immediate	001011xx	immediate	将立即数 immediate 加入累加器 A 中，带进位
SUB A, R?	001100xx		从累加器 A 中减去寄存器 R? 的值
SUB A, [R?]	001101xx		从累加器 A 中减去寄存器 R? 指向的存储单元的值
SUB A, MM	001110xx	MM	从累加器 A 中减去 MM 指向的存储单元的值
SUB A, immediate	001111xx	immediate	从累加器 A 中减去立即数 immediate
SBB A, R?	010000xx		从累加器 A 中减去寄存器 R? 的值，减借位
SBB A, [R?]	010001xx		从累加器 A 中寄存器 R? 指向的存储单元的值，减借位
SBB A, MM	010010xx	MM	从累加器 A 中减去 MM 指向存储单元的值

			值, 减借位
SBB A, immediate	010011xx	immediate	从累加器 A 中减去立即数 immediate, 减借位
AND A, R?	010100xx		累加器 A “与” 寄存器 R? 的值
AND A, [R?]	010101xx		累加器 A “与” 寄存器 R? 指向存储单元的值
AND A, MM	010110xx	MM	累加器 A “与” 地址 MM 指向的存储单元的值
AND A, immediate	010111xx	immediate	累加器 A “与” 立即数 immediate
OR A, R?	011000xx		累加器 A “或” 寄存器 R? 的值
OR A, [R?]	011001xx		累加器 A “或” 寄存器 R? 指向的存储单元的值
OR A, MM	011010xx	MM	累加器 A “或” 地址 MM 指向存储单元的值
OR A, immediate	011011xx	immediate	累加器 A “或” 立即数 immediate
MOV A, R?	011100xx		将寄存器 R? 的值送到累加器 A 中
MOV A, [R?]	011101xx		将寄存器 R? 指向存储单元的值送到累加器 A 中
MOV A, MM	011110xx	MM	将地址 MM 指向存储单元的值送到累加器 A 中
MOV A, immediate	011111xx	immediate	将立即数 immediate 送到累加器 A 中
MOV R?, A	100000xx		将累加器 A 的值送到寄存器 R? 中
MOV [R?], A	100001xx		将累加器 A 的值送到寄存器 R? 指向的存储单元中
MOV MM, A	100010xx	MM	将累加器的值送到地址 MM 指向的存储单元中
MOV R?, immediate	100011xx	immediate	将立即数 immediate 送到寄存器 R? 中
LEAA, MM	1001 10xx	MM	将地址 MM 送到累加器 a 中
MOV SP, immediate	1001 11xx	Immediate	将立即数 immediate 送到栈指针寄存器 SP 中
JC MM	101000xx	MM	若进位标志是 1, 跳转到 MM 地址
JZ MM	101001xx	MM	若零标志是 1, 跳转到 MM 地址
JMP MM	101011xx	MM	跳转到 MM 地址
IN	101100xx		从输入寄存器 RIN 读入数据到累加器 A 中
OUT	101101xx		将累加器 A 的值输出到寄存器 ROUT
INT immediate	101110xx	immediate	中断, 中断号为立即数 immediate
RET	110010xx		子程序调用返回
SHR A	110100xx		累加器 A 逻辑右移一位
SHL A	110101xx		累加器 A 逻辑左移一位
RCR A	110110xx		累加器 A 带进位循环右移一位
RCL A	110111xx		累加器 A 带进位循环左移一位
NOP	111000xx		空指令
NOT A	111001xx		累加器 A 取反, 结果存入累加器 A

CALL MM	111010xx	MM	调用 MM 地址的子程序
IRET	1111 10xx		中断返回

提示：机器码 1 中的符号“x”表示该位可以为 0，也可以为 1，它由具体的指令决定。例如指令 adda, r0 和指令 adda, r1 的机器码 1 分别为 00010000 和 00010001。

## 附录 BMIPS 指令(32 位)

本附录总结的是 32 位 MIPS 指令。表 B1~表 B3 定义了每条指令的 opcode 和 funct 字段，并对指令的功能做了简单的说明。使用以下符号：

- [reg]: 寄存器的内容
- Imm: I 类型指令的 16 位立即数字段
- Addr: J 类型指令的 26 位地址字段
- SignImm: 32 位符号扩展的立即数 = { {16{imm[15]}}, imm }
- ZeroImm: 32 位 0 扩展的立即数 = { 16' b0, imm }
- Address: [rs] + SignImm
- [Address]: 存储单元 Address 的内容
- BTA: 分支目标地址 = PC + 4 + (SignImm << 2)
- JTA: 跳转目标地址 = { ( PC + 4 ) [31:28], addr, 2' b0 }
- lable: 指定指令地址的文本

表 B-1 按 opcode 字段排列的指令

Opcode	名称	描述	操作
000000(0)	R 类型	所有 R 类型指令	见表 B-2
000001(1) (rt = 0/1)	bltz rs, lable / bgez rs, lable	小于 0 转移/大于等于 0 转移	If ([rs] < 0) PC=BTA/ If ([rs] >= 0) PC=BTA
000010(2)	J lable	跳转	PC=JTA
000011(3)	Jal lable	跳转并链接	\$ra = PC+4, PC=JTA
000100(4)	Beq rs, rt, lable	如果相等则转移	If([rs] == [rt]), PC=BTA
000101(5)	Bne rs, rt, lable	如果不相等则转移	If([rs] != [rt]), PC=BTA
000110(6)	Blez rs, lable	如果小于或等于 0 则转移	If([rs] ≤ 0), PC=BTA
000111(7)	Bgtz rs, lable	如果大于 0 则转移	If([rs] > 0), PC=BTA
001000(8)	Addi rt, rs, imm	立即数加法	[rt] = [rs] + SignImm (有符号立即数扩展)
001001(9)	Addiu rt, rs, imm	无符号立即数加法	[rt] = [rs] + SignImm (无符号立即数扩展)
001010(10)	slti rt, rs, imm	设置小于立即数	[rs] < SignImm ? [rt] = 1 : [rt] = 0
001011(11)	sltiu rt, rs, imm	设置小于无符号立即数	[rs] < SignImm ? [rt] = 1 : [rt] = 0
001100(12)	andi rt, rs, imm	立即数与	[rt] = [rs] & ZeroImm
001101(13)	ori rt, rs, imm	立即数或	[rt] = [rs]   ZeroImm
001110(14)	xori rt, rs, imm	立即数异或	[rt] = [rs] ^ ZeroImm
001111(15)	lui rt, imm	装入立即数的高 16 位	[rt] = {imm, 16' b0}
010000(16) (rs=0/4)	mfc0 rt, rd / mtc0 rt, rd	从协处理器 0 移出/移入协处理器 0	[rt] = [rd] / [rd] = [rt] (rd 是协处理器 0 中的寄存器)
010001 (17)	F-type	fop = 16/17:F 类型指令	浮点运算类型指令，见表 B-3

010001(17) (rt=0/1)	bclft lable / bclt lable	Fop = 8: 如果 fpcond 为 FALSE/TRUE 则转移	If (fpcond == 0) PC=BTA If (fpcond==1) PC=BTA
011100(28) (func=2)	mul rd, rs, rt	乘法 (32 位结果)	[rd] = [rs] × [rt]
100000(32)	lb rt, imm(rs)	装入字节	[rt] = SignExt([Address]7:0)
100001(33)	lh rt, imm(rs)	装入半字	[rt] = SignExt([Address]15:0)
100011(35)	lw rt, imm(rs)	装入字	[rt] = [Address]15:0
100100(36)	lbu rt, imm(rs)	装入无符号字节	[rt] = ZeroExt([Address]7:0)
100101(37)	lhu rt, imm(rs)	装入无符号半字	[rt] = ZeroExt([Address]15:0)
101000(40)	sb rt, imm(rs)	存储字节	[Address]7:0 = [rt]7:0
101001(41)	sh rt, imm(rs)	存储半字	[Address]15:0 = [rt]15:0
101011(43)	sw rt, imm(rs)	存储字	[Address] = [rt]
110001(49)	Lwcl ft, imm(rs)	将字装入 FP 协处理器 1 中	[ft] = [Address]
111001(56)	Swcl ft, imm(rs)	将字存储到协处理器 1 中	[Address] = [ft]

表 B-2 按 funct 字段排列的 R 类型指令

Funct	指令名称	指令描述	操作
000000(0)	sll rd, rt, shamt	逻辑左移	[rd] = [rt] << shamt
000010(2)	srl rd, rt, shamt	逻辑右移	[rd] = [rt] >> shamt
000011(3)	sra rd, rt, shamt	算术右移	[rd] = [rt] >>> shamt
000100(4)	sllv rd, rt, rs	带变量的逻辑左移	[rd] = [rt] << [rs]4:0
000110(6)	srlv rd, rt, rs	带变量的逻辑右移	[rd] = [rt] >>> [rs]4:0
000111(7)	srav rd, rt, rs	带变量的算术右移	[rd] = [rt] >>>> [rs]4:0
001000(8)	jr rs	跳转寄存器	PC = [rs]
001001(9)	jalr rs	跳转和链接寄存器	\$ra = PC+4, PC = [rs]
001100(12)	syscall	系统调用	System call exception
001101(13)	break	中断	Break exception
010000(16)	mfhi rd	从 hi 寄存器中移出	[rd] = [hi]
010001 (17)	mthi rs	移入 hi 寄存器	[hi] = [rs]
010010 (18)	mflo rd	从 lo 寄存器中移出	[rd] = [lo]
010011 (19)	mtlo rs	移入 lo 寄存器	[lo] = [rs]
011000(24)	mult rs, rt	乘法	{[hi], [lo]} = [rs] × [rt]
011001(25)	multu rs, rt	无符号乘法	{[hi], [lo]} = [rs] × [rt]
011010(26)	div rs, rt	除法	[lo] = [rs] / [rt], [hi] = [rs] % [rt]
011011(27)	divu rs, rt	无符号除法	[lo] = [rs] / [rt], [hi] = [rs] % [rt]

100000(32)	add rd, rs, rt	加法	$[rd] = [rs] + [rt]$
100001(33)	addu rd, rs, rt	无符号加法	$[rd] = [rs] + [rt]$
100010(34)	sub rd, rs, rt	减法	$[rd] = [rs] - [rt]$
100011(35)	subu rd, rs, rt	无符号减法	$[rd] = [rs] - [rt]$
100100(36)	and rd, rs, rt	与	$[rd] = [rs] \& [rt]$
100101(37)	or rd, rs, rt	或	$[rd] = [rs] \mid [rt]$
100110(38)	xor rd, rs, rt	异或	$[rd] = [rs] \wedge [rt]$
100111(39)	nor rd, rs, rt	或非	$[rd] = \sim([rs] \mid [rt])$
101010(42)	slt rd, rs, rt	设置小于	$[rs] < [rt] ? [rd] = 1 : [rd] = 0$
101011(43)	sltu rd, rs, rt	设置小于无符号	$[rs] < [rt] ? [rd] = 1 : [rd] = 0$

表 B-3 F 类型指令

Funct	指令名称	指令描述	操作
000000(0)	add.s fd, fs, ft / add.d fd, fs, ft	FP 加法	$[fd] = [fs] + [ft]$
000001(1)	sub.s fd, fs, ft / sub.d fd, fs, ft	FP 减法	$[fd] = [fs] - [ft]$
000010(2)	mul.s fd, fs, ft / mul.d fd, fs, ft	FP 乘法	$[fd] = [fs] \times [ft]$
000011(3)	div.s fd, fs, ft / div.d fd, fs, ft	FP 除法	$[fd] = [fs] / [ft]$
000101(5)	abs.s fd, fs / abs.d fd, fs	FP 取绝对值	$[fd] = ([fs] < 0) ? [-fs] : [fs]$
000111(7)	neg.s fd, fs / neg.d fd, fs	FP 取反	$[fd] = [-fs]$
111010(58)	c.seq.s fs, ft / c.seq.d fs, ft	FP 相等比较	$fpcond = ([fs] == [ft])$
111100(60)	c.lt.s fs, ft / c.lt.d fs, ft	FP 小于比较	$fpcond = ([fs] < [ft])$
111110(62)	c.le.s fs, ft / c.le.d fs, ft	FP 小于或等于比较	$fpcond = ([fs] \leq [ft])$

表 B-4 常用 MIPS 指令集及格式

MIPS 指令集（共 31 条）									
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-型	op	rs	rt	rd	shamt	func			
add	000000	rs	rt	rd	00000	100000	Add \$1, \$2, \$3	$\$1 = \$2 + \$3$	rd←rs+rt; 有符号加法
addu	000000	rs	rt	rd	00000	100001	Addu\$1, \$2, \$3	$\$1 = \$2 + \$3$	rd←rs+rt; 无符号加法
sub	000000	rs	rt	rd	00000	100010	sub \$1, \$2, \$3	$\$1 = \$2 - \$3$	rd←rs-rt; 有符号减法
subu	000000	rs	rt	rd	00000	100011	Subu\$1, \$2, \$3	$\$1 = \$2 - \$3$	rd←rs-rt; 无符号减法
and	000000	rs	rt	rd	00000	100100	and \$1, \$2, \$3	$\$1 = \$2 \& \$3$	rd←rs&rt; 逻辑与
or	000000	rs	rt	rd	00000	100101	or \$1, \$2, \$3	$\$1 = \$2   \$3$	rd←rs rt; 逻辑或
xor	000000	rs	rt	rd	00000	100110	xor \$1, \$2, \$3	$\$1 = \$2 \oplus \$3$	rd←rs xor rt; 异或
nor	000000	rs	rt	rd	00000	100111	nor \$1, \$2, \$3	$\$1 = \sim(\$2   \$3)$	rd←not(rs rt); 或非
slt	000000	rs	rt	rd	00000	101010	Slt\$1, \$2, \$3	if(\$2<\$3) \$1=1 else \$1=0	if(rs<rt)rd=1 else rd=0 ; 有符号数小于则置 1
sltu	000000	rs	rt	rd	00000	101011	Sltu\$1, \$2, \$3	if(\$2<\$3) \$1=1 else \$1=0	if(rs<rt)rd=1 else rd=0 ; 无符号数小于则置 1
sll	000000	00000	rt	rd	shamt	000000	sll \$1, \$2, 10	$\$1 = \$2 \ll 10$	rd ←- rt << shamt;

									shamt 存放移位的位数，也是指令中的立即数。 逻辑左移
srl	000000	00000	rt	rd	shamt	000010	srl \$1, \$2, 10	\$1=\$2>>10	逻辑右移
sra	000000	00000	rt	rd	shamt	000011	sra \$1, \$2, 10	\$1=\$2>>10	算术右移， 保留符号位
sllv	000000	rs	rt	rd	00000	000100	sllv \$1, \$2, \$3	\$1=\$2<<\$3	逻辑左移
srlv	000000	rs	rt	rd	00000	000110	srlv \$1, \$2, \$3	\$1=\$2>>\$3	逻辑右移
srav	000000	rs	rt	rd	00000	000111	srav \$1, \$2, \$3	\$1=\$2>>\$3	算术右移， 保留符号位
jr	000000	rs	00000	00000	00000	001000	jr \$31	goto \$31	PC <-rs
<b>I 型</b>	op	rs	rt	immediate					
addi	001000	rs	rt	immediate			addi \$1, \$2, 100	\$1=\$2+100	rt<-rs+immediate; 有符号扩展 立即数加法
addiu	001001	rs	rt	immediate			addiu \$1, \$2, 100	\$1=\$2+100	无符号扩展 立即数加法
andi	001100	rs	rt	immediate			andi \$1, \$2, 10	\$1=\$2 & 10	立即数与，无符号
ori	001101	rs	rt	immediate			andi \$1, \$2, 10	\$1=\$2   10	立即数或，无符号
xori	001110	rs	rt	immediate			andi \$1, \$2, 10	\$1=\$2 ⊕ 10	立即数异或，无符号
lui	001111	00000	rt	immediate			lui \$1, 100	\$1=100*65536	rt<-immediate* 65536; 将 16 位立即数 放到目标寄存器 高 16 位， 低 16 位填 0
lw	100011	rs	rt	immediate			lw \$1, 10(\$2)	\$1=memory[\$2+10]	立即数有符号扩展



sw	101011	rs	rt	immediate	sw \$1, 10(\$2)	memory[\$2+10] =\$1	立即数有符号扩展
beq	000100	rs	rt	immediate	beq \$1, \$2, 10	if(\$1==\$2) goto PC+4+40	立即数有符号扩展
bne	000101	rs	rt	immediate	bne \$1, \$2, 10	if(\$1!=\$2) goto PC+4+40	立即数有符号扩展
slti	001010	rs	rt	immediate	slti \$1, \$2, 10	if(\$2<10) \$1=1 else \$1=0	小于有符号立即数 则置 1
sltiu	001011	rs	rt	immediate	sltiu \$1, \$2, 10	if(\$2<10) \$1=1 else \$1=0	小于无符号立即数 则置 1
<b>J 型</b>	op	address					
j	000010	address			j 10000	goto 10000	PC← (PC+4) [31..28], address, 0, 0; address=10000/4
jal	000011	address			jal 10000	\$31←-PC+4; goto 10000	\$31←-PC+4; PC←-(PC+4) [31..28], address, 0, 0; address=10000/4

# 附录 C 数字电路实验芯片功能说明

在 DreamLogic 中，每一个型号的管脚样式都代表一定的含义，熟悉这些代表性的符号，便于直观识别器件的控制信号输入管脚的功能。下面以同步置数/异步清零十六进制加法计数器 74LS161 为例加以说明。

74LS161 如下图 1 所示：

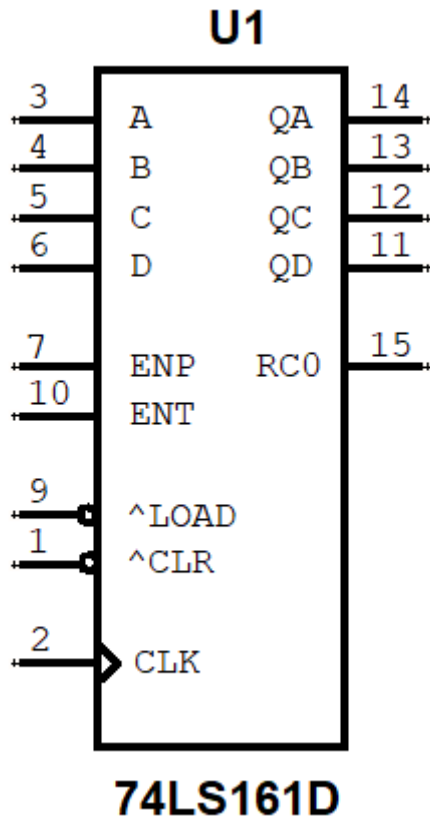


图 1:74LS161 器件型号

如图 1 所示 74LS161 器件图，器件的管脚编号与实际芯片的管脚编号相同，但是管脚的位置不同。例如，CLK 管脚编号为 2，实际芯片的时钟输入管脚 CLK 的编号也为 2，只是它们在芯片中的位置不同。

在实际芯片中，管脚是从芯片开口处逆时针进行编号的。

在图 1 中，74LS161 的控制管脚有 ENP、ENT、LOAD、CLR 以及 CLK，其中 ENP 和 ENT 管脚表示输入信号为高电平时有效；LOAD 和 CLR 管脚末端有一个圆圈，表示低电平有效；CLK 管脚末端有一个三角形，表示时钟上升沿有效，若在 CLK 管脚末端增加一个圆圈，则表示下降沿有效。管脚末端的圆圈通常表示输入信号经过一个非门后加载到输入端。

管脚编号	管脚名称	有效电平	功能描述
1	CLR	低电平	异步清零端，信号优先级最高，一旦 CLR=0，立即将输出 QA~QD 清零。
9	LOAD	低电平	异步置数端，优先级次于 CLR，当清零信号 CLR 无效时（高电平），若 LOAD=0，则立即将输入数据 A~D 传送到输出端 QA~QD。
7	ENP	高电平	ENP 和 ENT 是计数使能端，当它们均为有效电平时，计数器才能做加计数。若其中一个为无效电平（低电平），那么计数器输出 QA~QD 保持不变。
10	ENT		
2	CLK	↑	计数器只对输入时钟的上升沿进行计数，在计数时，CLK 端每

			输入一个上升沿，计数器输出 QA~QD 的数字值加 1。
2~6	A~D	-	计数器置数数据输入端，在 LOAD 有效时，A~D 并行传送到输出端 QA~QD。
11~14	QA~QD	-	计数值输出端，在计数状态下，随着 CLK 上升沿的输入，输出计数值逐次加 1
15	RCO	高电平有效	RCO 是计数器发生进位输出端，也就是当计数值 QA~QD=1111=15 时，RCO=1，表示计数已达最大值，下个时钟上升沿将发生进位

表 1:74LS161 管脚说明

## 优先编码器 74LS148

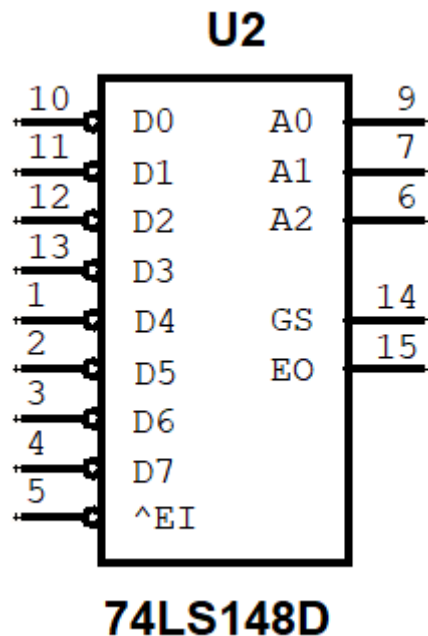


图 2:优先编码器 74LS148

管脚名称	有效电平	功能描述
EI	0	编码输入使能端，只有当 EI=0 使能时，才能对输入的 D0~D7 最高优先级位进行编码。否则，编码输出 A0~A2=111 且 GS=1，表示编码无效
D0~D7	0	编码输入端，D0 优先级最低，D7 优先级最高
A0~A2	-	编码输出端，A0 是最低位，A2 是最高位，A2A1A0 表示的十进制数对应于输入编码 D0~D7 中最高有效编码位的标号。例如，A2A1A0=4，则表示 D7~D0=1110 xxxx，也就是最高有效位 D4=0
GS	-	片选优先编码输出端，当 GS=0 时，表示有效编码，且 D0~D7 中至少有一位有效（为低电平）
EO	-	使能输出端，当 EO=1 时，唯一表示编码输入 EI=0 使能，且

	输入编码 D0~D7 均为高电平，即没有有效编码位的情况
--	------------------------------

表 2: 74LS148 优先编码器管脚说明

### 3-8 译码器 74LS138

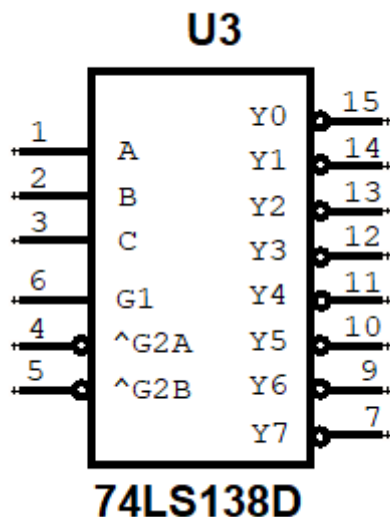


图 3: 3-8 译码器

管脚名称	有效电平	功能描述
A~C	-	地址输入端，A 为最低位，C 为最高位。当译码器工作时，可以看成是一个分配器，将 G1=1 分配给由 CBA 地址唯一确定的输出端。例如，当 CBA=101 时，Y5 端接收 G1，经过一个非门输出低电平，而其余输出均为高电平。
G1	高电平	片选输入端，当 G1=1，G2A+G2B=0 时，译码器处于工作状态。否则，译码器被禁止，所有的输出端被封锁在高电平。利用片选的作用可以将多片连接起来以扩展译码器的功能。
G2A	低电平	
G2B	低电平	
Y0~Y7	低电平	译码器工作时，有且只有一位输出低电平。哪一位为低电平是由当前输入地址 CBA 决定的。

表 3: 74LS138 译码器管脚说明

## 7 段数码显示器 74LS48

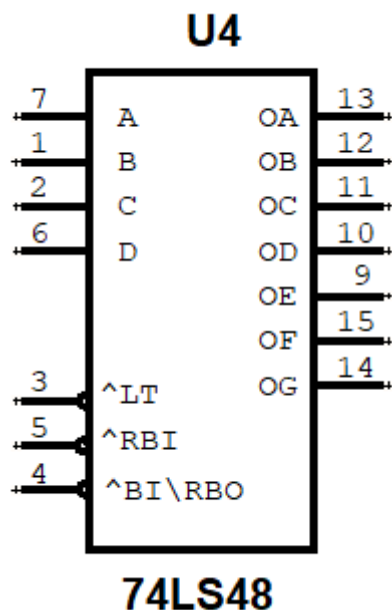


图 4：共阴极七段显示译码器

管脚名称	有效电平	功能描述
LT	低电平	LT 是灯测试输入信号。 当 LT=0 的信号输入时，被驱动数码管的七段同时点亮，以检查该数码管各段能否正常发光。工作时应使 LT 为高电平。 译码时，LT 输入高电平。
RBI	低电平	RBI 是灭零输入。 设置灭零输入的目的是为了能把不希望显示的零熄灭。当被驱动的数码管显示的不是 0 时，灭零输入 RBI 不起作用。 译码时，RBI 输入高电平。
BI/RBO	低电平	灭灯输入/灭零输出信号。 当 BI/RBO 灭灯输入低电平时，被驱动数码管的七段同时熄灭。 译码时，BI/RBO 输入高电平。
A~D	-	数字码输入端，A 为最低位，D 为最高位。 译码时，被驱动的七段数码管显示 DCBA 表示的十六进制数值。
OA~OG	高电平	译码时，OA~OG 中的所有高电平驱动七段数码管显示输入数字。 若是共阳极七段显示译码器 74LS47，则是使用 OA~OG 中的所有低电平驱动七段数码管显示数字。这就是共阴极和共阳极的区别。

表 4：共阴极七段显示译码器 74LS48 管脚说明

## 4 位超前进位加法器 74LS283

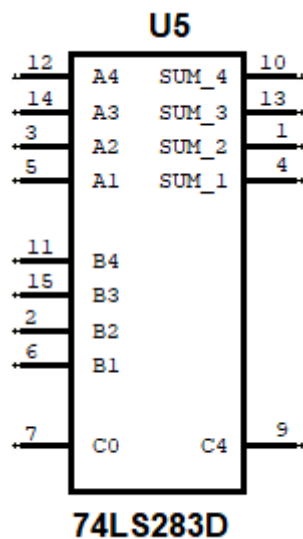


图 5:四位超前进位加法器 74LS283

管脚名称	有效电平	功能描述
A1~A4	-	四位二进制加数，A1 为最低位，A4 为最高位。
B1~B4	-	四位二进制加数，B1 为最低位，B4 为最高位。
SUM_1~SUM_4	-	加运算结果。 Sum=A+B+C0 的低四位
C0	高电平	来自低位的进位
C4	高电平	进位输出。 当加运算结果超出 SUM_4~SUM_1 所能表示的最大数 15 时，输出进位 C4=1。

表 5: 超前进位加法器 74LS283 的管脚描述

## 4 位算术逻辑运算器 74LS181

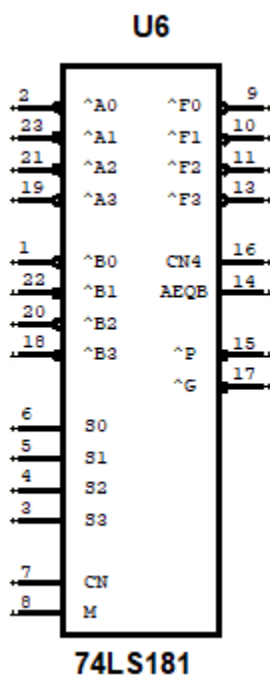


图 6：算术逻辑运算器 74LS181

管脚名称	功能描述
A0~A3	四位二进制运算操作数，A3 为最高位
B0~B3	四位二进制运算操作数，B3 为最高位
F0~F3	四位二进制运算结果，F3 为最高位
S0~S3	运算方式，4 位二进制可编码 $2^4=16$ 种不同的运算方式
M	运算类型。 当 M=1 时，A 与 B 进行逻辑运算。逻辑运算包含与、或、非等。 当 M=0 时，A 与 B 进行算术运算。算术运算包含加、减以及逻辑与算术的混合运算
CN	CN 表示最低位进位输入。 CN=0，属于有进位型运算，共十六种； CN=1，属于无进位型运算，也包含十六种。
CN4	最高位进位输出，低电平有效。
AEQB	当 F3~F0=1111 时，AEQB=1，否则，AEQB=0
G	进位发生输出。
P	进位传送输出。

表 6：算术运算器 74LS181 的管脚说明

触发器的异步置数 PR 和异步清零 CLR 优先级相同，均为最高优先级。在触发器的其它信号起作用时，需要将 PR 和 CLR 置为 0，使其无效。

## D 触发器

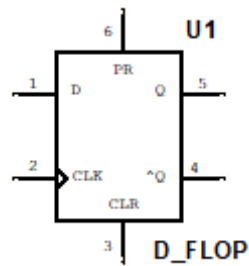


图 7: D 触发器

管脚名称	有效电平	功能描述
D	-	1 位输入数据
PR	高电平	异步置数端。 当 PR=1, CLR=0 时, 将输出 Q 置位为 1
CLR	高电平	异步清零端。 当 CLR=1, PR=0 时, 将输出 Q 清零
CLK	↑	时钟上升沿有效。 当 PR=0 且 CLR=0 时, 时钟上升沿到来时, 输入数据 D 传送到输出端 Q, Q 取反传送到输出端 ^Q

表 7: D 触发器管脚描述

## JK 触发器

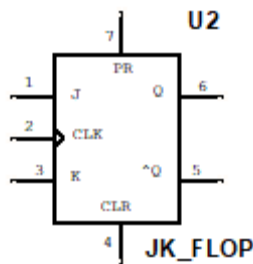


图 8: JK 触发器

管脚名称	有效电平	功能描述
D	-	1 位输入数据
PR	高电平	异步置数端。 当 PR=1, CLR=0 时, 将输出 Q 置位为 1
CLR	高电平	异步清零端。 当 CLR=1, PR=0 时, 将输出 Q 清零
J	高电平	同步置 1 端。 当 J=1, K=0, 时钟上升沿到来时, 输出端 Q 置 1



K	高电平	同步置 0 端。 当 $k=1, J=0$ ，时钟上升沿到来时，输出端 Q 置 0
CLK	↑	时钟上升沿有效。 当 $J=K=0$ 时，Q 保持不变； 当 $J=K=1$ ，时钟上升沿到来时，输出端 Q 翻转，即 $Q=\neg Q$ 。利用这一特性，将 JK 连起来就构成了 T 触发器。

表 8: JK 触发器管脚描述

## SR 触发器

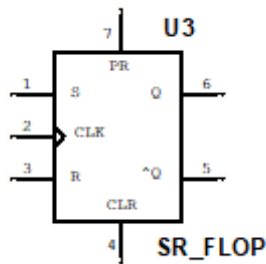


图 9: SR 触发器

管脚名称	有效电平	功能描述
D	-	1 位输入数据
PR	高电平	异步置数端。 当 $PR=1, CLR=0$ 时，将输出 Q 置位为 1
CLR	高电平	异步清零端。 当 $CLR=1, PR=0$ 时，将输出 Q 清零
S	高电平	同步置位端。 当 $S=1, R=0$ ，时钟上升沿到来时，输出端 Q 被置位为 1
R	高电平	同步清零端。 当 $R=1, S=0$ ，时钟上升沿到来时，输出端 Q 被清零
CLK	↑	时钟上升沿触发同步置位或清零

表 9: SR 触发器管脚描述

## T 触发器

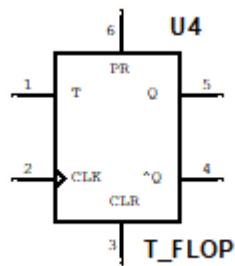


图 10: T 触发器

管脚名称	有效电平	功能描述
D	-	1 位输入数据
PR	高电平	异步置数端。 当 PR=1, CLR=0 时, 将输出 Q 置位为 1
CLR	高电平	异步清零端。 当 CLR=1, PR=0 时, 将输出 Q 清零
T	高电平	当 T=1, 时钟上升沿到来时, 输出端 Q 发生翻转, 即 $Q = \neg Q$ 。 当 T=0 时, 输出 Q 保持不变
CLK	↑	时钟上升沿有效。

表 10: T 触发器管脚描述

## 四位双向移位寄存器 74LS194

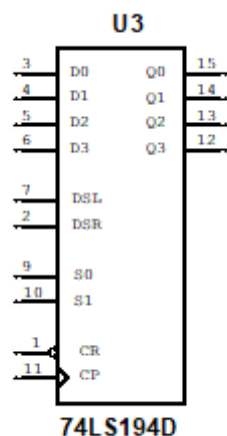


图 11: 四位双向移位寄存器

管脚名称	有效电平	功能描述
D0~D3	-	移位数据输入端
Q0~Q3	-	移位数据输出端
DSL		左移串行数据输入端

DSR		右移串行数据输入端
S0~S1		移位方式控制端。 S1S0=00, 输出保持; S1S0=01, 右移; S1S0=10, 左移; S1S0=11, 并行传送, 输出 Q0~Q3=D0~D3
CR	低电平	异步清零端, 优先级最高, 与时钟无关。 当 CR=0 时, 输出 Q0~Q3 均为低电平
CP	↑	时钟输入端, 上升沿有效。 触发移位或并行输出。

表 11: 四位双向移位寄存器管脚说明

## 8 选 1 数据选择器 74LS151D

74LS151D 为互补输出的 8 选 1 数据选择器, 引脚排列如图 12 所示, 功能见表 12。选择控制端 (地址端) 为 C~A, 按二进制译码, 从 8 个输入数据 D0~D7 中, 选择一个需要的数据送到输出端 Y, G 为使能端, 低电平有效。

- (1) 使能端 G=1 时, 不论 C~A 状态如何, 均无输出 (Y=0, W=1), 多路开关被禁止。
- (2) 使能端 G=0 时, 多路开关正常工作, 根据地址码 C、B、A 的状态选择 D0~D7 中某一个通道的数据输送到输出端 Y。如: CBA=000, 则选择 D0 数据到输出端, 即 Y=D0。CBA=001, 则选择 D1 数据到输出端, 即 Y=D1, 依此类推。

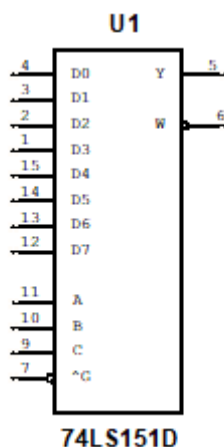


图 12:8 选 1 数据选择器

管脚名称	功能描述
D0~D7	8 位数据输入端
A~C	数据选择端, CBA 表示的二进制数决定将哪一位输入数据作为输出。例如: CBA=101, 那么选择 D5 输出到 Y 端
G	当 G=1 时, 无论输入数据以及数据选择信号如何变化, 输出端 Y 始终为低电平, W 始终为高电平。当 G=0 时, 多路开关正常工作, 根据地址码 C、B、A 的状态选择 D0~D7 中某一个通道的数据输送到输出端 Y。输出端 W 的值为 Y 取反。
W 和 Y	互补的数据输出端

表 12: 8 选 1 数据选择器管脚说明

## 二-五进制计数器 74LS90

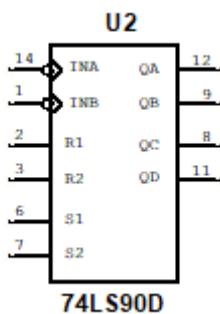


图 13：二-五进制计数器

复位输入				输出			
R1	R2	S1	S2	Q3	Q2	Q1	Q0
1	1	0	x	0	0	0	0
1	1	x	0	0	0	0	0
x	x	1	1	1	0	0	1
x	0	x	0	计数			
0	x	0	x	计数			
0	x	x	0	计数			
x	0	0	x	计数			

表 13：二-五进制计数器功能表

74LS90 是二-五进制异步计数器，该芯片包含 4 个主从触发器和附加门，其中一级 Q0 组成二分频计数器，计数器输入端是 INA；另外三级（Q3、Q2、Q1）组成五分频计数器，计数输入端是 INB。四级的共同清零输入是 R1、R2，而 S1、S2 则为预置 9 输入端。利用该异步计数器，根据“计数到 N 时置 0”的原则（也叫反馈复位法），可构成 N 进制计数器。

## 参考文献

- [1] 阎石,王红编;清华大学电子学教研组编.数字电子技术基础(第6版).北京:高等教育出版社,2016
- [2] [美]哈里斯(Harris,D.M.)等著.陈俊颖译.数字设计和计算机体系结构(原书第2版).北京:机械工业出版社,2016
- [3] 白中英,谢松云主编;朱正东,方维,吴俊编著.数字逻辑(第六版).北京:科学出版社,2013
- [4] 缪相林,白中英主编;郭军,高荔,杨春武编著.数字逻辑习题解析与实验教程(第六版).北京:科学出版社,2013
- [5] 饶增仁等编著.数字电路实验教程.北京:清华大学出版社,2013
- [6] 马汉达,赵念强编著.数字逻辑电路设计学习指导与实验教程.北京:清华大学出版社,2012
- [7] (美)布莱恩特(Bryant,R.E.),奥哈拉伦(O'Hallaron,D.R.)著.龚奕利,雷迎春译.深入理解计算机系统(原书第二版).北京:机械工业出版社,2010.11(2014.1重印)
- [8] 王爽著.汇编语言(第二版).北京:清华大学出版社,2008.4
- [9] 唐朔飞编著.计算机组成原理(第二版).北京:高等教育出版社,2008.1
- [10] 周大海主编.计算机组成原理实验与课程设计教程.北京:北京航空航天大学出版社,2015.8
- [11] 沈美明,温冬蝉编著.IBM-PC汇编语言程序设计(第二版).北京:清华大学出版社,2001.8