

# I 前言

---

## I.I 关于本书

传统数字电路教材会伴随大量与物理器件有关的篇幅，但现在电路工艺和实现方法发展迅速，某些物理电路原理不再是必修，因为我们需要根据实际情境选择适合的实现方法，而不同实现方法间的细节区别又非常大。但另一方面，它们背后的数字电路原理是一致的。

因此本书只会介绍电路背后的抽象结构和设计原理，但不会讨论与实现工艺有关方面的东西。例如，本书会介绍算术电路的结构，并且讨论它的数学原理，但不会讨论这些电路如何通过物理工艺实现。

删除物理电路相关的部分后，数电教材变得非常简短，于是本书就可以用更多电路结构来填充这些空白，让大家在相同甚至更短的时间内，学会更多的数字电路设计。

也就是说，本书的特点在于篇幅简短，读者可以快速学会各种电路的设计原理与结构，本书的页数，就是本书篇幅简短的直接证明。

学会数字电路的抽象原理后，再去学习某个具体工艺相关的特性，就轻松多了。这也是作者我给大家的建议。因为不同工艺背后的数字电路原理都是一致的，所以我们只需要根据工艺特性，稍微调整我们的抽象设计，就能得到一个具体的方案。

另一方面，现在电路自动化工具发展迅速，很多时候，我们只需要负责抽象设计，而具体的物理电路结构，则可交给软件自动生成，或者交给专精某种工艺的工程师，让他们来考虑具体电路实现。

另一方面，抽象设计本身是很灵活的，我们只需要稍微调整电路设计的参数和结构，就可以产生大量不同的设计。对于具体工艺，我们可以基于工艺特性调整，或选择我们的抽象设计，使得它最适配实现工艺。

所以，抽象设计还具有更高的可移植性：抽象设计只需要针对某个特定工艺稍作调整，就可以生成对应的专用设计，但某个特定工艺下的专用设计，很难通过调整，而移植到其它工艺上。

下图展示了电路设计的不同抽象层次间的关系：

电路背后的数学原理



抽象的电路结构

具体工艺下的电路结构

当然，我们也可以从应用角度看待：

受限于读者水平，这本书会有相当多错误与不足之处，大请家务必指出！！

## 2 目录

---

2 第一部分导读 (Introduction to The Part I) .....	6
2.1 第一部分导读.....	6
2.2 自主学习的建议.....	6
2.3 动机与问题.....	7
3 数字电路的基础元件 (Basic Circuit in Digital Circuit) .....	8
3.1 前言.....	8
3.2 真值表.....	8
3.2.1 真值表的化简与推理.....	8
3.3 逻辑运算与逻辑电路.....	8
3.3.1 逻辑运算.....	8
3.3.2 逻辑表达式.....	9
3.4 基础锁存器与触发器.....	9
3.5 逻辑表达式及其化简.....	9
3.6 物理电路中的数字电路.....	9
4 数字电路的基础知识 (Basic In Digital Circuit) .....	10
4.1 前言.....	10
4.2 进制符号 (Symbol of Number) .....	10
4.3 数制 (Number System) 、数码 (Digit) 、基数 (Radix) 、位权 (Weight) .....	11
4.3.1 位权与进制转换 (Radix Transfer) .....	11
4.3.2 进制的正式定义.....	11
4.4 二进制数 (Binary) 的运算.....	11
4.5 二进制小数.....	12
4.6 有限位数 (Finite Number System) 与比特 (Bit) .....	12
4.7 位运算 (Bit Operation) .....	12
4.8 有符号数 (Signed Number) .....	13
4.9 原码 (Sign Magnitude) 、反码 (1's complement) 、补码 (2's complement) .....	13
5 基础运算器 (Basic Operator) .....	14
5.1 前言.....	14
5.1.1 一些术语.....	14
5.2 加法器.....	15
5.2.1 半加器 (HA、Half Adder) 、全加器 (FA, Full Adder) 与逐位进位加法器 (RCA, Ripple-Carry Adder) .....	15
5.2.2 超前进位加法器 (CLA, Carry Lookahead Adder) .....	15
5.2.3 进位旁路加法器 (CBA, Carry Bypass Adder) .....	16
5.3 乘法器.....	17
5.3.1 基于加法器的阵列乘法器 (Array Multiplier) .....	17
5.3.2 基于进位保留加法器 (CSA, Carry Save Adder) 的阵列乘法器.....	18
5.3.3 加法树, 在运算器中运用结合律.....	18
5.4 减法器 (Subtractor) 与加减混合器 (Adder Subtractor) .....	18
5.5 除法器.....	19
5.5.1 恢复余数阵列除法器 (Restoring Array Divider) .....	19
5.5.2 不恢复余数阵列除法器 (Non-restoring Array Divider) .....	19
5.5.3 除法慢于乘法.....	20

5.6	十进制加法器 (Decimal Adder) .....	20
5.6.1	BCD 码简介.....	20
5.6.2	8421BCD 码加法器.....	20
5.7	进制转换 (Radix Transfer) .....	20
5.7.1	特殊情况直接转换 (Direct Conversion in Special Case) .....	20
5.7.2	位权相加法 (Weighted Sum) .....	21
5.7.3	短除法 (Short Division) .....	21
5.7.4	Double-Dabble 算法.....	21
5.8	移位器.....	22
5.8.1	逻辑移位、算数移位与循环位移.....	22
5.8.2	桶型移位器.....	22
6	存储器 (Memory) .....	23
6.1	前言.....	23
6.2	寄存器 (Register) 、寄存器组 (Register Set) 与寻址器 (Addresser) .....	23
6.2.1	寄存器 (Register) 与寄存器组 (Register Set) .....	23
6.2.2	译码器 (Decoder) .....	23
6.3	随机存取存储器与只读存储器.....	24
6.3.1	随机存取存储器 (RAM, Random Access Memory) 与只读存储器 (ROM, Read Only Memory) 的简单介绍.....	24
6.3.2	大规模存储器的组织.....	24
6.4	FIFO 与 FILO.....	25
6.4.1	FIFO .....	25
6.4.2	FILO .....	25
6.5	查找表与内容可寻址单元.....	25
6.5.1	查找表 (LUT, Look Up Table) .....	25
6.5.2	内容可寻址单元 (CAM, Content Addressable memory) .....	25
6.6	带特殊功能的寄存器.....	26
6.6.1	计数器 (Counter) .....	26
6.6.2	移位寄存器 (Shift Register) .....	26
6.6.3	线性反馈移位寄存器 (LFSR, Linear Feedback Shift Register) .....	26
7	时序电路设计.....	27
7.1	前言.....	27
7.2	时序电路基础.....	27
7.2.1	时钟 (CLK, Clock) 、同步 (SYNC, Synchronization) 与异步 (ASYNC, Asynchronous) .....	27
7.2.2	节拍发生器 (Sequence Timer) .....	28
7.3	串行运算器.....	28
7.3.1	串行运算器的结构.....	28
7.3.2	串行运算器的优势与劣势.....	28
7.4	有限状态机.....	29
7.4.1	状态转移图 (STD, State Transition Diagram) 与有限状态机 (FSM, Finite State Machine) .....	29
7.4.2	FSM 的电路实现与应用.....	29
7.5	时序电路的描述.....	30
7.5.1	时序电路的状态转移图、状态转移表、时序图.....	30
7.5.2	用方程描述时序电路.....	30
7.5.3	时序电路的分析与优化.....	30

8	数据交换 (Data Transfer) .....	31
8.1	前言.....	31
8.2	接口标准 (Interface Standard) 与接口协议 (Interface Protocol) .....	31
8.3	交换节点.....	32
8.3.1	交换节点的使用场景.....	32
8.3.2	网络 (Network) 与图论 (Graph Theory) , 一些术语.....	32
8.3.3	交换节点的功能与结构.....	33
8.4	交换协议.....	33
8.4.1	仲裁协议.....	33
9	第二部分导读.....	34
9.1	第二部分导读.....	34
9.2	计算机体系结构中的重要思想.....	34
10	认识计算机与 CPU 与 RISC-V.....	35
10.1	前言.....	35
10.2	计算机的工作流程.....	35
10.2.1	简化的计算机模型与工作流程.....	35
10.3	RV32I 指令集.....	36
10.3.1	RV32I 的用户执行环境.....	36
10.3.2	RV32I 的指令格式.....	36
10.3.3	RV32I 指令集.....	37
10.3.4	RV32I 的伪指令 (Pseudo Instruction) .....	38
10.4	认识 RV32I 汇编.....	39
10.4.1	汇编语言 (Assembly Language) 概述.....	39
10.4.2	从高级语言到机器语言.....	39
10.5	内存中的数据.....	39
10.5.1	大小端序.....	39
10.5.2	数据对齐.....	40
10.5.3	内存空间的划分.....	40
10.5.4	内存空间的管理.....	40
11	一个简单的 RV32I 处理器.....	41
11.1	前言.....	41
11.2	一个单周期 RV32I 处理器的内部抽象结构.....	41
11.2.1	抽象的内部架构视图 .....	41
11.2.2	抽象的执行过程.....	41
11.3	CPU 的更多细节.....	42
11.3.1	CPU 与计算机上的数据交换.....	42
11.3.2	计算机总线的时序控制.....	42
11.3.3	计算机中的 I/O 接口与外设.....	43

### 3 第一部分导读 (INTRODUCTION TO THE PART I)

---

#### 3.1 第一部分导读

第一部分讨论的是数字电路设计的一些基础理论与概念以及一些基础的数字电路结构。

另外感谢大家的勘误与建议！

#### 3.2 自主学习的建议

为了帮助一些年幼而学习能力有限的同学，我在这里先教大家如何自主学习。为了方便，我把这些建议都总结成了表格！

##### 解决一个问题的步骤

1. 确定问题是什么，准确地描述出你的问题，越准确越好。
2. 确定这个问题有哪些特殊的要求和需求，存在哪些限制，越清楚越好。
3. 确定解决这个问题可能需要哪些知识和工具，准备好它们。
4. 先设计出一个可行的解决方案，能解决问题，大致正确即可。
5. 检查你的方案是否满足特殊要求，是否存在 bug，并且确定如何改进这些问题。
6. 修复 bug 并改进方案。
7. 如果你对这个方案满意了，就进入下一步。否则返回第 5 步。
8. 你把一个问题解决了。

解决一个问题中可能产生新的问题，那么新问题同样按照上述步骤进行一遍。

##### 自主解决问题的各种途径

必应：可以在搜索页面选择国内版（中文结果）或国际版（英文结果）。比百度好用。  
谷歌：英文搜索结果很好，中文搜索结果也不错。可惜国内上不了。很容易搜出被盗版 pdf 电子书。  
百度：中文搜索结果还不错，英文搜索结果很少。  
维基百科：很优秀，英文词条比较全。国内上不了。  
百度百科：在墙内，维基百科的劣质代替品。  
知乎：可以搜到很多不错的文章。  
B 站：可以搜到不少视频资源。  
CSDN：大量转载文章和盗版资源，虽然是盗版资源但也可能要钱。  
Stack Overflow：一个 IT 技术问答网站。  
Stack Exchange：Stack Overflow 的父板块，涵盖内容不局限于 IT。  
谷歌学术：用于搜索论文文献，国内有许多镜像网。  
SCI-hub：提供 DOI 就能免费下载许多论文。类似的论文盗版网站还有很多。  
Library Genesis：能搜到很多英文资料书的 pdf 格式文件，能找到大部分你想找的英文书资料。

##### 英语烂如何看懂英文资料

1. 提高英文水平，水平越高阅读越流畅，即使不能完全理解所有词汇语法。
2. 使用翻译器/网页翻译/划词翻译，越 NB 的翻译器越好。
3. 对照翻译结果与原文，排除翻译错误。
4. 翻译错误的句子或单词，单独复制出来单独翻译一遍，可能就对了。
5. 还是翻不出某个单词是什么意思，就百度或者谷歌查这个单词。
6. 及时记笔记记下你理解出的东西。以防复习时又要重新翻译一遍。

你提的问题是不是问题（提问的智慧：[lug.ustc.edu.cn/wiki/doc/smart-questions/](http://lug.ustc.edu.cn/wiki/doc/smart-questions/)）

你提的问题在逻辑上合理吗？

你提问的表述真的准确表达了你想问的问题吗？

你提的问题有价值吗？回答这个问题有什么意义？回答这个问题会对你有哪些帮助？

这个问题是否更适合在将来知识储备更丰富的时候讨论？

这个问题是否更应该由自己来找到答案？

##### 自学的方法

1. 不要害怕晦涩，不要把理解停留在表面。不要不懂却故作装懂。最好能清晰认识自己会什么、不会什么、清楚什么、不清楚什么。
2. 把厚的书读薄，提炼出背后的思想与哲理。死记硬背只会过目不忘，走形式无法领悟精髓。比起知识点，更重要的是背后的思想哲理。
3. 把薄的书读厚，充分运用提炼出来的精髓，结合实际情况与更多思想，碰撞出更多火花。要学会应用和推广思想。
4. 搞清楚你学的知识是用来干什么的，解决什么问题的，如何被运用的。既要提高知识水平，也要提高实践水平。
5. 少读没用的书，一本书的薄厚在于你能从其中学到多少新的知识与思想，而不取决于它的页数或难度。
6. 明确你的学习目的，明确你学习的主线。明确你要学的东西和要解决的问题。
7. 让自己的学习保持在线上，明白自己需要用什么知识，需要学什么知识。用不到的知识可能真就一直都用不到。
8. 保持学习的耐心和兴趣，你知道支持你学习的动力是什么吗？

### 3.3 动机与问题

初学者容易遇到的一个问题就是不知道自己所学的知识是为什么服务的。所以在“动机与问题”这一部分里，我会列出与本书有关的一些问题，以让读者明白有哪些问题应该被注意：

问题	一种回答
<b>电路是为什么服务的？</b>	为它的应用场景服务的。需要什么电路，就要准备什么电路。
<b>电路的优化目标？</b>	取决于实际情况，利用有限的资源（比如有限的电路面积）在情境中最大化效益。不同情况下的优化目标是不同的。
<b>为什么要改善电路的处理延迟？</b>	有的情景对延迟很敏感，以现实中的餐厅为例，顾客如果等太久就会不满意。为此需要控制好延迟。
<b>如何改善电路的处理延迟？</b>	很多电路的延迟都取决于这个电路中延迟最大的那条路线，针对此进行优化一般就能明显地改善延迟。
<b>为什么要改善电路的处理频率？</b>	处理频率是指单位时间内完成的操作次数，频率与延迟不冲突，电路可能频率高但同时延迟也很高。
<b>如何改善电路的处理频率？</b>	比如将整块的大电路划分为头尾接在一起的多个小电路，通常小的电路的频率比整块的大电路更高。
<b>为什么要减小电路的面积？</b>	有的应用对电路面积很敏感，比如硅基工艺的CPU，CPU面积过大可能会带来包括成本等各种方面的许多问题。
<b>如何减小电路的面积？</b>	比如采用更紧凑的电路布局，或者面积更小的电路设计。
<b>为什么要改善电路的功耗？</b>	一方面是减少电费开销。另一方面，功耗也通常与性能有关，例如高功耗的CPU通常发热也会很严重，就容易过热导致必须降频。
<b>如何改善电路的功耗？</b>	比如采用低功耗的电路设计。
<b>什么是电路算法？</b>	简单来说就是电路功能的数学描述。好的算法能让电路性能产生质的飞跃，事半而功倍。
<b>电路算法的重要性？</b>	电路算法与电路的延迟、频率、面积、功耗等都密切相关。通常不会有完美的通用算法，而是针对某种情景特定设计的算法。
<b>运算器是干什么的？</b>	电路中的运算器用来处理运算的，比如四则运算。
<b>存储器是干什么的？</b>	电路中的存储器是用来存储数据的。
<b>能用和实用的区别？</b>	做出一个满足功能的设计非常简单，但是做出满足功能的优秀设计通常很难，也很重要。希望读者有点追求，不要止步于“能用”。

与本章有关的一些问题的描述与它们的一种回答。加粗是重点。当然，这些问题都是开放的，所以这些问题的答案也远不止一种。而且，本章的知识也不能完全解决它们。同样地，实际情况中也绝不仅仅只有这点问题。实际中，你应该发现并描述出自己发现的问题，然后试图给出一个符合要求的答案（很多时候不得不折中）

读者应该随时复习这张表格，以确保自己在学习设计时考虑了应该考虑的问题（即加粗部分的那些问题）。通常而言，一个实用的思维过程是：先问为什么要做，再问要做什么，再问怎么做，再问怎么优化。

## 4 数字电路的基础元件 (BASIC CIRCUIT IN DIGITAL CIRCUIT)

### 4.1 前言

这章我们来了解一下数字电路的一些基础电路知识，包括逻辑电路的基础逻辑门、触发器和锁存器。

如果你是名 redstoner，也可以选择先看一遍 [www.bilibili.com/read/cv6754204](http://www.bilibili.com/read/cv6754204)，然后跳过本章，直接学习后续内容！（从下一章：数字电路的基础知识继续）

### 4.2 真值表

输入 A	输入 B	输出 C	注释
0	0	0	A=0、B=0 时，C=0
0	1	1	A=0、B=1 时，C=1
1	0	0	A=1、B=0 时，C=0
1	1	1	A=1、B=1 时，C=1

一个真值表的例子

真值表就是一张用于枚举输入与输出间所有取指情况的表格，如上表就列举了输入 A、B 在不同取值情况下输出 C 的值（这里假定 A、B 均在 0、1 间取值）。

真值表能很直观地展示输入与输出间的关系，所以等下我们就会用真值表来描述逻辑电路元件的输入输出关系。

#### 4.2.1 真值表的化简与推理

输入 D	输入 E	输出 F
0	0	0
0	1	0
1	0	1
1	1	1

另一个真值表的例子

从上面那张表我们可以发现，只要确定 D、E、F 间两个量的值，就能根据表立即得知剩下那个量的值。即使仅已知其中一个量的值，比如已知 A=0，也可以排除 A≠0 时的情况。也就是说可以根据已知条件排除一些肯定不会发生的情况。

另外，真值表在一些情况下是可以化简的，如根据上表我们可以发现：当且仅当 D=0 时，F=0；当且仅当 D=1 时，F=1。也就是说 F 的值完全取决于 D 的取值，而与 E 无关。于是将上表的 E 剔除，得到下表：

输入 D	输出 F
0	0
1	1

化简后的真值表

关于真值表化简有不少技巧，但这里不讨论。首先是机器算法能自动帮我们化简真值表，其次这并不是现在的重点（学会用真值表描述输入与输出间的关系就行了）。

当然，机器化简也有一定局限性：基于同一张真值表有时可以得到多种不同化简结果，这时候就需要结合实际情况判断哪种化简结果更值得采纳。以及根据实际情况配置自动优化算法的参数策略设置，以优先得到更感兴趣的化简结果。

如果读者感兴趣，也可以通过搜索引擎了解相关的化简技巧与自动化简算法等。

### 4.3 逻辑运算与逻辑电路

#### 4.3.1 逻辑运算

中文名	英文名	符号	表达式	所有运算情况 (蓝色的是运算结果)	说明
非运算	not	'	A'	$0' = 1, 1' = 0$	A 为 0 时运算结果为 1，否则为 0。（即“非”逻辑，对象为真时输出位假，反之为真）
与运算	and	.	A·B	$0 \cdot 0 = 0, 0 \cdot 1 = 1 \cdot 0 = 0, 1 \cdot 1 = 1$	A、B 都为 1 时，运算结果为 1，否则为 0。（即“与”逻辑，所有对象都成立则为真）
或运算	or	+	A+B	$0+0 = 0, 0+1 = 1+0 = 1+1 = 1$	A、B 间只要一者为 1，运算结果就为 1，否则为 0。（即“或”逻辑，一者为真时为真）
异或运算	xor	$\oplus$	A⊕B	$0 \oplus 0 = 1 \oplus 1 = 0, 0 \oplus 1 = 1 \oplus 0 = 1$	A、B 不同时，运算结果为 1，否则为 0。（即“异”逻辑，对象不同时为真）
同或运算	xnor	$\odot$	A⊕B	$0 \odot 0 = 1 \odot 1 = 1, 0 \odot 1 = 1 \odot 0 = 0$	A、B 相同时，运算结果为 1，否则为 0。（即“同”逻辑，对象相同时为真）

本文采用的基础逻辑运算符号

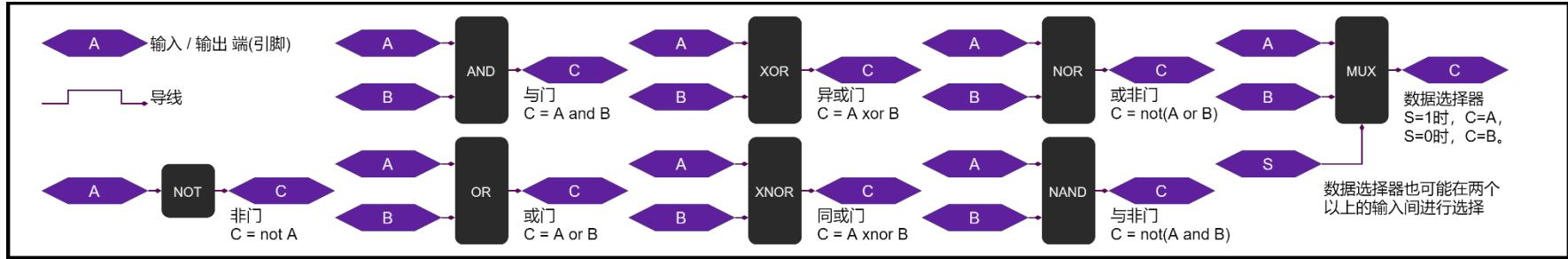
上表给出了各种不同的逻辑运算的符号和它们的定义，以及所有运算的情况。说明一栏中的括号说明了这些运算的中文名的由来，这些运算的中文名直接对应了这些运算的功能，可以基于此通过中文名快速记住这些运算的功能。

### 4.3.2 逻辑表达式

由前面所说的逻辑运算符与变量构成的表达式，即为所谓的逻辑表达式，例如：

上式表示，

本文将使用的输入端、输出端、导线及一些基础元件的电路符号如下：



本文采用的逻辑门符号。导线

图中这些逻辑门和数据选择器可能接入多个输入，变成  $n$  输入逻辑门，这里列出一些  $n$  输入逻辑门在本书中的定义：

名称	$n$ 输入逻辑门对应的逻辑表达式（用 $IN_n$ 表示第 $n$ 个逻辑门输入，OUT 表示逻辑门输出）
$n$ 输入与门	$OUT = IN_1 \cdot IN_2 \cdot IN_3 \cdot IN_4 \cdot \dots \cdot IN_n$
$n$ 输入或门	$OUT = IN_1 + IN_2 + IN_3 + IN_4 + \dots + IN_n$
$n$ 输入异或门	$OUT = IN_1 \oplus IN_2 \oplus IN_3 \oplus IN_4 \oplus \dots \oplus IN_n$
$n$ 输入同或门	$OUT = IN_1 \odot IN_2 \odot IN_3 \odot IN_4 \odot \dots \odot IN_n$
$n$ 输入与非门	$OUT = (IN_1 \cdot IN_2 \cdot IN_3 \cdot IN_4 \cdot \dots \cdot IN_n)'$
$n$ 输入或非门	$OUT = (IN_1 + IN_2 + IN_3 + IN_4 + \dots + IN_n)'$

一些  $n$  输入逻辑门的功能定义

## 4.4 基础锁存器与触发器

锁存器：<https://baike.baidu.com/item/锁存器>

触发器：<https://baike.baidu.com/item/触发器/193146#viewPageContent>

## 4.5 逻辑表达式及其化简

逻辑表达式：<https://baike.baidu.com/item/逻辑表达式>

逻辑表达式化简：<https://wenku.baidu.com/view/f9060918a76e58fafbb00301.html>

注意！这个简化并不是绝对的。大部分情况下，你能将一个表达式变换出各种不同的形式，但是它们之间很难看出谁更适合实际情况。这时就要结合实际，根据实际情况判断哪个逻辑表达式更好。

此外，除了基于公式的化简，还可以使用卡诺图或者机器化简算法来进行化简，如果有需要可以自行了解。

## 4.6 物理电路中的数字电路

晶体管与逻辑门：<https://wenku.baidu.com/view/8c33283a31126edb6f1a1046.html>

## 5 数字电路的基础知识 (BASICS IN DIGITAL CIRCUIT)

### 5.1 前言

这章我们会讨论一些数电的基础数学知识，这些知识一般在大学数电教材中会被放在第一章，但为了降低学习曲线，我把这部分内容放到了这里，希望读者能透过逻辑电路的知识更直观地理解本章中的概念及其意义。

本章内容可能稍微有点多，且大部分都是理论推导，可能略显枯燥。但这都是在为后续内容做准备。学完本章的内容后，后面有关各种运算器和数字电路设计的内容会轻松许多。事实上，后续内容的难度取决于你对本章内容的理解程度。

如果你看完以后还是觉得没理解，认为需要更通俗的解释，可以在网上找到许多相关的通俗的科普视频。另外，为了降低学习难度，我删去了一些不那么重要的篇幅，并且没有给出严格的数学推导过程，如果有需要也请在网上自行查找资料。

### 5.2 进制符号 (SYMBOL OF DIGITS)

我们生活中最为熟悉的数，从小算到大的数，小学算术题里面的数，可以叫做十进制数。比如  $19+1=20$ ,  $87+12=99$ ，这就是十进制的加法计算，大家应该非常熟悉。

那么既然有了十进制，还会不会有二进制，八进制，六进制，十六进制这些其它进制？当然是有的，我们先来讨论二进制，与我们所熟知的十进制一样，二进制也是用来表示一个数字的。与十进制不同的是，它仅由 1 和 0 构成。

10 进制	0	1	2	3	4	5	6	7	8
2 进制	0	1	10	11	100	101	110	111	1000

10 进制	9	10	11	12	13	14	15	16	17
2 进制	1001	1010	1011	1100	1101	1110	1111	10000	10001

10 进制	18	19	20	21	22	23	24	25	26
2 进制	10010	10011	10100	10101	10110	10111	11000	11001	11010

十进制数与其对应的二进制数

上表就列举了一些十进制数所对应的二进制数。如表可见，二进制仅由 1 和 0 组成。

但是你仔细一看就会发现，有一些二进制数和十进制数之间很容易混淆，比如 10 进制的“10”和二进制的“10”完全一样，但它们的意思却不同。为了解决这个问题，我们会给每个数加一个下标，用这个下标来说明这个数是几进制的。给上面的表格添加下标后如下：

10 进制	$0_{10}$	$1_{10}$	$2_{10}$	$3_{10}$	$4_{10}$	$5_{10}$	$6_{10}$	$7_{10}$	$8_{10}$
2 进制	$0_2$	$1_2$	$10_2$	$11_2$	$100_2$	$101_2$	$110_2$	$111_2$	$1000_2$

10 进制	$9_{10}$	$10_{10}$	$11_{10}$	$12_{10}$	$13_{10}$	$14_{10}$	$15_{10}$	$16_{10}$	$17_{10}$
2 进制	$1001_2$	$1010_2$	$1011_2$	$1100_2$	$1101_2$	$1110_2$	$1111_2$	$10000_2$	$10001_2$

十进制数与其对应的二进制数（加了下标）

如上，现在我给每个十进制数都加了一个“10”的下标，给每个二进制都加了一个“2”的下标。而对于其他进制，我们也采用类似的做法：我们给某个数加上下标“k”以表明这个数是 k 进制的。

说完了二进制，我们再来看看其它进制是什么样的：

10 进制	0	1	2	3	4	5	6	7	8
2 进制	0	1	10	11	100	101	110	111	1000

8 进制	0	1	2	3	4	5	6	7	10
16 进制	0	1	2	3	4	5	6	7	8

10 进制	9	10	11	12	13	14	15	16	17
2 进制	1001	1010	1011	1100	1101	1110	1111	10000	10001

8 进制	11	12	13	14	15	16	17	20	21
16 进制	9	a	b	c	d	e	f	10	11

十进制数与其对应的不同进制数

这个表格就列出了一些十进制数所对应的二进制、八进制、十六进制数，可以发现 k 进制数的每位只可能有 k 种符号。比如二进制的每位只可能是 0、1，而 8 进制可能是 0、1、2、3、4、5、6、7，十进制则是 0、1、2、3、4、5、6、7、8、9。十六进制有整整 16 种，分别是 0、1、2、3、4、5、6、7、8、9、a、b、c、d、e、f。显然对于十六进制来说阿拉伯数字已经不够用了，所以这里用字母作为十六进制剩下的五个数的符号。

你会发现这个表格里我并没有增加下标，比如十六进制的 11 应该写成  $11_{16}$  这样以防混淆。为什么这里我就没加呢？因为我懒。而且有时候给每个数都写上进制下标的话，公式会变丑而影响阅读。所以有时候我并不会明确写出某个数是几进制的，但你应该根据上下文的说明判断某个数是几进制的。

现在我们才知道十进制外居然还有这么多不同的进制。但请记住，不同进制仅仅是同一个数的不同表示方式，不同进制之间是相互等价的，进制之间也可以相互转换。

### 5.3 数制 (NUMBER SYSTEM)、数码 (DIGIT)、基数 (RADIX)、位权 (WEIGHT)

现在我们知道  $k$  进制每位有  $k$  种符号了。通过之前的表格我们已经对进制的规律管窥一斑，但应该如何准确描述这些东西呢？这就是接下来要学习的东西。为了准确描述“ $k$  进制”的基本特征，我们需要用到以下术语：

术语	解释
数制	各种进制的总称，如二进制、十进制等等，它们的总称就叫数制。
数码	别名数字，表示数的符号。比如十进制有 10 个数码，分别是 0、1、2、3、4、5、6、7、8、9。
基数	数制拥有的数码的数量，比如十进制有 10 个数码，那它的基数为 10。同理 16 进制的基数为 16。
位权	第 $n$ 位的权重，比如十进制第 2 位的位权是 10，第 3 位的位权是 100。以此类推。其实就是第 $n$ 位的大小。
数制、数码、基数与位权的解释	

上表解释了这四个术语的含义。接下来重点关注一下位权，这里用一张表展示不同进制第  $n$  位的位权：

不同进制的第 $n$ 位的位权 ( $k$ 进制的第 $n$ 位的位权为 $k^{(n-1)}$ ，表中数据均为十进制数)									
	第 1 位	第 2 位	第 3 位	第 4 位	第 5 位	第 6 位	第 7 位	第 8 位	第 $n$ 位
10 进制	1	10	100	1000	10000	100000	1000000	10000000	$10^{(n-1)}$
2 进制	1	2	4	8	16	32	64	128	$2^{(n-1)}$
8 进制	1	8	64	512	4096	32768	262144	2097152	$8^{(n-1)}$
16 进制	1	16	256	4096	65536	1048576	16777216	268435456	$16^{(n-1)}$

表中贴心地直接告诉了你  $k$  进制第  $n$  位位权的计算公式是  $k^{(n-1)}$ ，也就是说现在你可以用这个公式计算出  $k$  进制第  $n$  位代表的大小（即位权）是多少了。位权是沟通不同进制的一栋桥梁，利用位权可以转换进制。

#### 5.3.1 位权与进制转换 (Radix Transfer)

已知这里有一个二进制数  $1001_2$ ，如何才能求出它对应的十进制数呢？根据位权公式我们知道，二进制数前四位的位权分别是  $1_{10}$ 、 $2_{10}$ 、 $4_{10}$ 、 $8_{10}$ 。于是我们让二进制的每位乘上所对应的位权，得到：

$$1_{10} \times 8_{10} + 0_{10} \times 4_{10} + 0_{10} \times 2_{10} + 1_{10} \times 1_{10} = 9_{10}$$

这样就知道了  $1001_2$  对应的十进制数是  $9_{10}$ 。

再举个例子，现在求出  $25_{10}$  所对应的二进制数：

$$2_{10} \times 10_{10} + 5_{10} \times 1_{10} = 10_2 \times 1010_2 + 101_2 \times 1_2 = 11001_2$$

因为要求得的是二进制数，所以还得先把十进制数转换成二进制数，然后在二进制下进行计算求出结果。虽然暂时还没学二进制的运算，但这不是现在的重点。

在这个计算步骤中可以注意到，我们只需要列出  $0、1、2、3、4、5、6、7、8、9$  和  $10、100、1000、10000\dots$  这些十进制数对应的二进制数就可以把左边的十进制数式转换成右边的二进制数式，因为式中就只出现了这些数。

$0、1、2、3、4、5、6、7、8、9$  这些数对应的二进制数可以直接事先算出来然后存在一张表里，计算时如果需要直接查表即可，这种思想叫做查表法。查表法事先将计算结果存在表中，而在计算时直接查表得到结果，从而避免了繁琐的计算过程。

在计算结果较小的情况下，对于  $10、100、1000、10000\dots$  也可以采用查表法。但即使不用查表法也可以专门推理出这些数与二进制的对应规律关系。显然这比直接找出所有十进制数与二进制数的对应规律关系要简单得多。

进制转换的算法非常多，但都是基于位权原理的，在后文讨论进制转换器时会介绍更多进制转换算法。

#### 5.3.2 $k$ 进制的正式定义

$k$  进制：基数为  $k$ ，第  $n$  位位权为  $k^{(n-1)}$  的数制。

### 5.4 二进制数 (BINARY) 的运算

现在我们就来学习如何计算二进制数。其实不同进制的运算方法都是一样的，还记得你小时候用的加减乘除竖式吗？二进制也可以用竖式来解决加减乘除，如：

$$\begin{array}{r} & 1 & 1 \\ + & 1 & 0 \\ \hline 1 & 0 & 1 \end{array} \quad \begin{array}{r} & 1 & 1 \\ - & 1 & 0 \\ \hline 0 & 1 \end{array} \quad \begin{array}{r} & 1 & 1 \\ \times & 1 & 0 \\ \hline 1 & 1 & 0 \end{array}$$

除法竖式也一样，我就不列了。类似地，其它进制也可以这样列竖式来计算加减乘除。

关于进位，在十进制中有  $9+1=10$ ，这叫逢 10 进 1。而在二进制中有  $1+1=10$ ，这叫逢 2 进 1。以此类推，在  $k$  进制中就有逢  $k$  进 1。这就是  $k$  进制的进位规律。

至于加减乘除以外的算法，也可以类似地从十进制算法推广得到。或者你也可以先将其它进制数转换为十进制数，然后用我们熟悉的十进制算法计算出结果，最后再将结果转回原进制表示。（用抽象代数的话说，不同数制是同构的。）

数字电路中几乎所有的运算器都是在二进制下进行运行的，这是因为用二进制的运算器更容易实现（其实不止运算器，其它电路模块也是）。所以我们主要关注的也是二进制运算器的算法。

## 5.5 二进制小数

不同进制的第 n 位小数的位权 (k 进制的第 n 位小数的位权为 $k^{-n}$ , 表中数据均为十进制数)									
	第 1 位小数	第 2 位小数	第 3 位小数	第 4 位小数	第 5 位小数	第 6 位小数	第 7 位小数	第 8 位小数	第 n 位小数
10 进制	0.1	0.01	0.001	0.0001	0.00001	0.000001	0.0000001	0.00000001	$10^{-n}$
2 进制	1/2	1/4	1/8	1/16	1/32	1/64	1/128	1/256	$2^{-n}$

上表给出了 k 进制第 n 位小数位权的公式  $k^{-n}$ , 它的形式跟 k 进制第 n 位数位权公式  $k^{(n-1)}$  很像。实际上, 如果把第 1 位小数改名叫第 0 位数, 第 n 位小数叫第 1-n 位数的话, 带入  $k^{(n-1)}$  也可以算出小数的位权。这样来看小数和整数的位权公式其实是统一的。

不同进制的小数的计算也可以通过列竖式来完成, 就跟十进制小数的竖式一样。小数的进制转换思路跟整数的进制思路一样, 都是每位乘与对应位权然后求和。

## 5.6 有限位数 (FINITE NUMBER SYSTEM) 与比特 (BIT)

所谓的有限位数就是位数有限的数。比特是一个单位, 符号为 bit 或者简写为 b, 复数形式为 bits。比特可以用于描述一个有限位二进制数有多少位, 比如我们可以用 8bits 来形容一个 8 位的有限位二进制数。此外还有一些其它单位, 它们与 bit 之间的关系是:

$$1 \text{ byte} = 8 \text{ bits}$$

$$1 \text{ KiB} = 1024 \text{ bytes}$$

$$1 \text{ MiB} = 1024 \text{ KiB}$$

$$1 \text{ GiB} = 1024 \text{ MiB}$$

$$1 \text{ TiB} = 1024 \text{ GiB}$$

是不是很熟悉? 它们就是你的电脑硬盘的存储大小单位。但是你的电脑上显示的是 MB 而不是 MiB, 是 GB 而不是 GiB, 这是为什么呢? 除了可能只是有的地方单纯习惯省略中间的 i 外, 也有可能是因为它们采用了这套标准 (SI 标准) :

$$1 \text{ KB} = 1000 \text{ Bytes}$$

$$1 \text{ MB} = 1000 \text{ KB}$$

$$1 \text{ GB} = 1000 \text{ MB}$$

$$1 \text{ TB} = 1000 \text{ GB}$$

某些骗子会混淆这两种标准来忽悠外行人。此外, 它们其实还有更大的单位比如 PB, 感兴趣可以自己去了解。

回到正题, 有限位数只能在它的位数范围内表示数, 如果计算超出了它的表示范围, 那么超出表示范围的数值就会被抛弃。比如一个 8bits 的有限位数算式是 1111111+1=0000 0000, 式中结果的表示范围只有 8bits, 于是正确结果 1 0000 0000 中的第九位被丢失了。

有限位数因为表示范围有限而丢失数值或导致计算结果出错的情况叫做溢出。

除了刚才这种情况, 小数精度不足也同样可以导致溢出。例如让一个精确到第二位小数的有限位数表示 1/99999 的话, 它只能四舍五入到零, 因为它本身最多只能精确表示到第二位小数。另外, 减法也可以导致溢出, 比如一个有限位数的表示范围在 +127 到 -127 之间的话, 那么它肯定不能表示小于 -127 的数, 因此在计算时产生小于 -127 的结果时就会导致结果出错, 也就是溢出。

## 5.7 位运算 (BIT OPERATION)

名称	符号	解释
与	A & B	A 和 B 的每位分别进行与逻辑运算。如 $1101 \& 0100 = 0100$ 。
或	A   B	A 和 B 的每位分别进行或逻辑运算。如 $1000   0001 = 1001$ 。
非	A'	A 的每位分别取反。如 $0100' = 1011$ 。
异或	A ⊕ B	A 和 B 的每位分别进行异或逻辑运算。如 $1001 \oplus 1011 = 0010$ 。
同或	A ⊙ B	A 和 B 的每位分别进行同或逻辑运算。如 $1001 \odot 1011 = 1101$ 。
(逻辑)左移	A << s	让 A 左移 s 位, 移出的空位用 0 补充。如 $0000\ 1101 << 4_{10} = 1101\ 0000$ 。
(逻辑)右移	A >> s	让 A 右移 s 位, 移出的空位用 0 补充。如 $1010\ 0000 >> 4_{10} = 0000\ 1010$ 。

8bits 有符号数的结构。其中 A 和 B 均为 n bit(s) 的有限位二进制数, s 必须大等于 0。除了逻辑移位还有算数移位, 但这里暂不讨论算数移位。

上表总结了各种位运算的定义, 你会发现它们其实就是以前学的逻辑运算的拓展, 就连符号都一样。逻辑运算只负责处理 1bit 的运算, 而位运算能处理 n bit(s) 的运算。左移和右移是新的东西, 它们叫移位运算, 用于移动二进制位。有时如果移太多位会导致溢出, 如  $0001 << 5_{10} = 0000$ , 数值被移到了表示范围外所以溢出了。为方便有时会规定在溢出时自动拓展位数, 如  $01 << 2_{10} = 100$ , 以避免溢出。

(逻辑)左移和(逻辑)右移还有两个有用的性质, 在不溢出或者自动拓展位数的情况下有:

$$A << s = A \times 2^s$$

$$A >> s = \frac{A}{2^s}$$

在运算器中这两条性质常被用于代替一些特殊的乘法和除法, 因为移位运算电路很简单。想一想, 这两条性质是如何成立的呢?

## 5.8 有符号数 (SIGNED NUMBER)

在计算机中可以用有符号数表示有符号的数值。相对地，无符号数用于表示没有符号的数。有符号数是一个  $n$  bit(s) 的有限位数，通常用它的最高位（第  $n$  位）用来表示符号，一般规定最高位为 0 时为正，为 1 时为负。以一个 8bits 的有符号数为例：

符号位 (第 8 位)	第 7 位	第 6 位	第 5 位	第 4 位	第 3 位	第 2 位	第 1 位
一个 8bits 有符号数的结构							

它的最高位变成了符号位，所以它只有 7 个位来表示实际的数值。一个无符号 8bits 数的表示范围是 255 到 0，而 8bits 的有符号数的表示范围是 +127 到 -127。而且，有符号数有 +0 和 -0 之分。以 8bits 的有符号数为例，0000 0000 表示 +0，1000 0000 表示 -0。

有符号数的符号位不应该接受或产生进位，否则可能出错。以 8bits 有符号数的一个加法算式为例，如果符号位接受进位，那么会有：0100 0000 + 0100 0010 = 1000 0010，转换成十进制表示就是  $64+66=2$ ，显然是错的。

有符号数还有一个问题，那就是它不能直接计算  $A+(-B)$ 。以两个 8bits 有符号数的加法  $0011\ 0011 + 1011\ 0000$  为例，正确的结果应该是 0000 0011，但是如果直接用加法竖式去算  $0011\ 0011 + 1011\ 0000$  的话显然得不到这个正确结果。

有符号数如果要计算  $A+(-B)$ ，只能先把它变成  $A-B$  再计算出结果。还是以  $0011\ 0011 + 1011\ 0000$  为例，正确的计算过程应该是：

$$0011\ 0011 + 1011\ 0000 = 0011\ 0011 - 0011\ 0000 = 0000\ 0011$$

那么有没有一种表示方法，能够直接计算  $A+(-B)$  呢？答案是有的，它就是接下来要介绍的补码。

## 5.9 原码 (SIGN MAGNITUDE) 、反码 (I'S COMPLEMENT) 、补码 (2'S COMPLEMENT)

名称	符号	求法
原码	$[N]_{原}$	原码就是有符号数。
反码	$[N]_{反}$	$[N]_{原} > 0$ 或 $[N]_{原} = +0$ 时 : $[N]_{反} = [N]_{原}$ 。 $[N]_{原} < 0$ 或 $[N]_{原} = -0$ 时 : 将 $[N]_{原}$ 符号位以外的其它位取反就能得到 $[N]_{反}$ 。
补码	$[N]_{补}$	$[N]_{原} > 0$ 或 $[N]_{原} = +0$ 时 : $[N]_{补} = [N]_{原}$ 。 $[N]_{原} < 0$ 时 : $[N]_{补} = [N]_{反} + 1$ 。（对于小数，请将 +1 改为加上最低位的位权，如 $0.111 + 0.001$ 就是给小数的最低位加个 1。） $[N]_{原} = -0$ 时 : 将 $[N]_{原}$ 的符号位设置为 0 就能得到 $[N]_{补}$ 。

原码、反码与补码的求法。后文我们会系统地介绍各种不同的数表示系统，届时我们会发现，上表中的反码和补码只是广义的补码表示系统的特例

十进制数	+7	+6	+5	+4	+3	+2	+1	+0	-0	-1	-2	-3	-4	-5	-6	-7	-8
原码	0111	0110	0101	0100	0011	0010	0001	0000	1000	1001	1010	1011	1100	1101	1110	1111	无
反码	0111	0110	0101	0100	0011	0010	0001	0000	1111	1110	1101	1100	1011	1010	1001	1000	无
补码	0111	0110	0101	0100	0011	0010	0001	0000	无	1111	1110	1101	1100	1011	1010	1001	1000

一些十进制数 (+7 到 -8) 所对应的原码、反码、补码（原码、反码、补码都被限定为了 4bits 的有限位数）

上面给出了原码、反码、补码的符号和求法。（严格来说这里讨论的补码叫 2 的补码，此外还有模 M 补码等，反码也叫 I 的补码）

与不同数制一样，这些不同码制也很容易被混淆，比如上表中的 1111 既可以是补码中的 1111 也可以是原码或反码中的 1111。和区分进制的做法类似，我们用方括号将数值圈起来，并且在右下角标注这个数的码字以区分码制。例如  $[0101]_{补}$ ，看到这个数的右下角我们就知道它是补码，而不是反码或者原码。

你可能会被第一张表中的  $[N]_{原}$ 、 $[N]_{反}$ 、 $[N]_{补}$  绕晕。如果会的话，请记住 N 是抽象的，N 本身不涉及具体的表示。 $[N]_{原}$ 、 $[N]_{反}$ 、 $[N]_{补}$  的意思分别是 N 这个数的原、反、补码表示。原、反、补码本质上只是有符号数的不同表示方式，就跟不同数制一样。

运算时，原码和反码的符号位都不会接受进位。但补码的符号位会接受进位。也就是说补码符号位会接受上一位的进位，且一般会忽略补码产生的溢出。这些规定很自然，例如按规定（忽略溢出）有： $[1111]_{补} + [0001]_{补} = [0000]_{补}$ ，用十进制表示就是  $-1 + 1 = 0$ ，完全正确。

如果你想问不同码制间的运算是什么情况（如  $[N]_{原} + [N]_{反}$ ），那我会告诉你这是未定义的。运算的前提就是参与运算的数的码制相同，就好比二进制不能跟十进制直接相加（如  $101_2 + 12_{10}$ ），必须先把数制统一再来计算。虽然它们可以相互转换，但也需要先统一后计算。

如果这里有两个 n bit(s) 的原码  $[A]_{原}$  和  $[B]_{原}$ ，那么根据定义可以知道它们分别对应的补码具有以下性质（读者可尝试自行推导）：

$$[A]_{补} + [B]_{补} = [A + B]_{补}$$

$$[A]_{补} - [B]_{补} = [A]_{补} + [-B]_{补} = [A - B]_{补}$$

第二条公式说明补码可以直接计算  $A+(-B)$ 。利用这点，可以先取负 B 得到 -B 的值，然后用加法替代减法 ( $A+(-B)$  代替  $A-B$ )。在设计运算器时如果利用这条性质，可以让单独一个加法器完成加法和减法两种运算。

补码没有 +0 和 -0 之分，因此它的表示范围比原码和反码要稍微大那么一点（如上面的表）。以 8bits 的原码、反码、补码为例，原码和反码的表示范围是 +127 到 -127，而补码的表示范围则是 +127 到 -128。n bit(s) 的情况以此类推。

正是补码具有如此多的良好性质，在计算机中一般都会统一用补码来表示带正负的数，而不是直接用有符号数。

可以把补码理解成一个时钟。假如现在它的时间在 13:00，要把时间调到 11:00 该怎么做？有两个办法，第一种方法就是让时针逆时针拨动 2 小时，第二种方法是将时针往前顺时针拨动 22 小时。补码计算  $A+(-B)$  的办法本质上跟第二种方法一样，以增代减。

有限位数跟时钟一样，都是循环的，以 4bits 无符号数为例，因为溢出， $1111 + 0001 = 0000$ 。就像 24:00 的下一点是 1:00 一样，从终点又回到了起点。如果不给这个无符号数 +1，那它会在这 4bits 里像时钟一样不停循环。（数学中类似的概念叫商群）

## 6 基础运算器 (BASIC OPERATOR)

### 6.1 前言

从这一章开始，我们将初步尝试利用前面所学的知识设计一些基础的运算器电路。如果前面学的很扎实，那本章的内容会很轻松。本章起我们将使用数字电路符号以及逻辑表达式来描述设计，如果忘了请去复习一下。

#### 6.1.1 一些术语

在实际应用中，我们会用到许多术语来描述电路的特征，或者作为改进的指标。这里列举其中的一些，现在看看就行：

术语名称	描述	说明
<b>稳健性</b>	模块越不容易出错、出错的后果越小，则稳健性越高。	有条公式是：模块可用性 = $\frac{\text{平均无故障时间}}{\text{平均无故障时间} + \text{平均修复时间}}$ 稳健性别名鲁棒性。
<b>改善比</b>	对原设计改动后，新设计较原设计改善了多少。	改善可以是不同方面的，通常单个调整不能改善所有方面，甚至对某些方面有负改善比。
<b>布线密度</b>	局部或全局的导线密度。	实际电路设计中如果某处的布线密度过高会导致很多问题。
<b>导线长度</b>	两器件间导线（互联导线）的长度。	导线本身具有一定的性能，具有信号传播延迟并且占据一定的空间。
<b>元件数</b>	一个模块使用的元件数。	元件和导线都会占用空间。
<b>元件性能</b>	一个元件的性能。	如工作频率等。
<b>耦合</b>	模块间的相互依赖/影响程度，或者说相关联度。	有时候这种关系是单方向的，例如 A 控制 B，B 却不控制 A。
<b>内聚</b>	模块内各部分彼此结合的紧密程度，或者说相关联度。	一个模块的内聚越高，它的内部越像是一个整体。
<b>延迟</b>	从开始到结束的时间差。	除了元件本身的延迟，导线也存在延迟（主要是互联导线）。
<b>频率</b>	单位时间内执行操作的次数。	频率不成整数比的模块之间在时序上较不容易配合。
<b>功率</b>	物理量，单位时间所作功。	相关物理量还有能耗、功耗、负载等，许多因素都会影响它们，它们也会影响你的设计。
<b>噪声</b>	预期外的，干扰电路正常工作的环境因素。	如果电路可能处于某些干扰严重的恶劣环境，请确保电路的稳健性能够对抗这些噪声。
<b>理想电路</b>	理想中的电路，通常会忽略噪声、线延迟等因素。	理想的假设可以简化我们的工作，但不要忘了，现实中的是实际电路，条件没有那么理想！
<b>实际电路</b>	实际中的电路。	设计理想电路时，也要尽可能考虑到其对应的实际电路涉及的许多因素和情况。
<b>扇入</b>	模块的输入直接接受了多少来自其它模块的输入。	扇入大就说明这个模块输入多。
<b>扇出</b>	模块的输出直接输出到了多少其他模块的输入。	扇出大就说明这个模块输出多。
<b>扇入系数</b>	门电路的扇出系数指门电路允许的输入端数目的范围。	实际电路设计中单个模块过大的扇入系数容易导致很多问题。
<b>扇出系数</b>	门电路的扇出系数指门电路允许的输出端数目的范围。	实际电路设计中单个模块过大的扇出系数容易导致很多问题。
<b>信号毛刺</b>	一些短时间内、正常功能外的信号输出震荡。	好的设计可以减少信号毛刺。必须将信号毛刺降低到可以容忍的地步。
<b>信号抖动</b>	较理想情况，实际情况下信号在某位置的小幅偏离。	可能由于噪声干扰等导致。设计时必须保证电路能容忍信号抖动等干扰因素。
<b>信号偏斜</b>	一条线路上的信号比预期提前/延迟到达。	可能由于导线延迟等导致。设计时序时必须考虑信号偏斜等实际中可能发生的误差因素。
<b>信号漂移</b>	两个同步信号间的相位差越来越大。	可能由于周期不同步导致。
<b>到达时间</b>	信号到达电路指定位置所需时间。	到达时间通常涉及一对数据，即信号改变后可能的最早和最晚到达时间（一个范围）。
<b>关键路径</b>	模块中最值得关注的一条线路。	例如模块中延迟或线路最长的线路，针对关键路径进行改进通常可以获得较大改善比。
<b>需求时间</b>	信号根据预期按时到达电路指定位置所需时间。	要确保待处理的输入数据已经稳定到达了输入端，才能开始处理输入端的值，否则会出错。
<b>建立时间</b>	输入信号到达输入后，被模块使用前需保持不变的时间。	就像自拍，自拍前必须把脸贴在摄像机前面，以让摄像机定位调整焦距。
<b>保持时间</b>	输入信号在被使用时，需要保持不变的时间。	又像自拍，拍照时不能乱动，不然拍出来的照片就花了。
<b>响应时间</b>	触发某一电路后，自触发开始到电路产生响应的时间。	响应可以定义为预期中电路要做的事情，触发可以定义为该电路输入端的某个状态等。
<b>复位时间</b>	电路恢复到某一状态所需时间。	比如撤离输入后，电路恢复到初始态所需的时间。
<b>上升时间</b>	信号从低电平变到高电平所需时间。	实际的电平改变当然是有延迟的。如果在电平改变时对信号采样，会采出不稳定的结果。
<b>下降时间</b>	信号从高电平变到低电平所需时间。	如果你的电路不止高低两个电平，这个概念可以拓展到更多不同的电平间。
<b>稳定时间</b>	信号到达某电平后，电平稳定所需时间。	电平改变后通常不会立即稳定，而是伴随一定的震动。经过稳定时间后才会稳定。

一些术语的描述与说明。做了一些个人解读，但最终还是符合这些术语的目的。觉得没描述清楚可以在网上搜索相关资料，会有更详细的描述

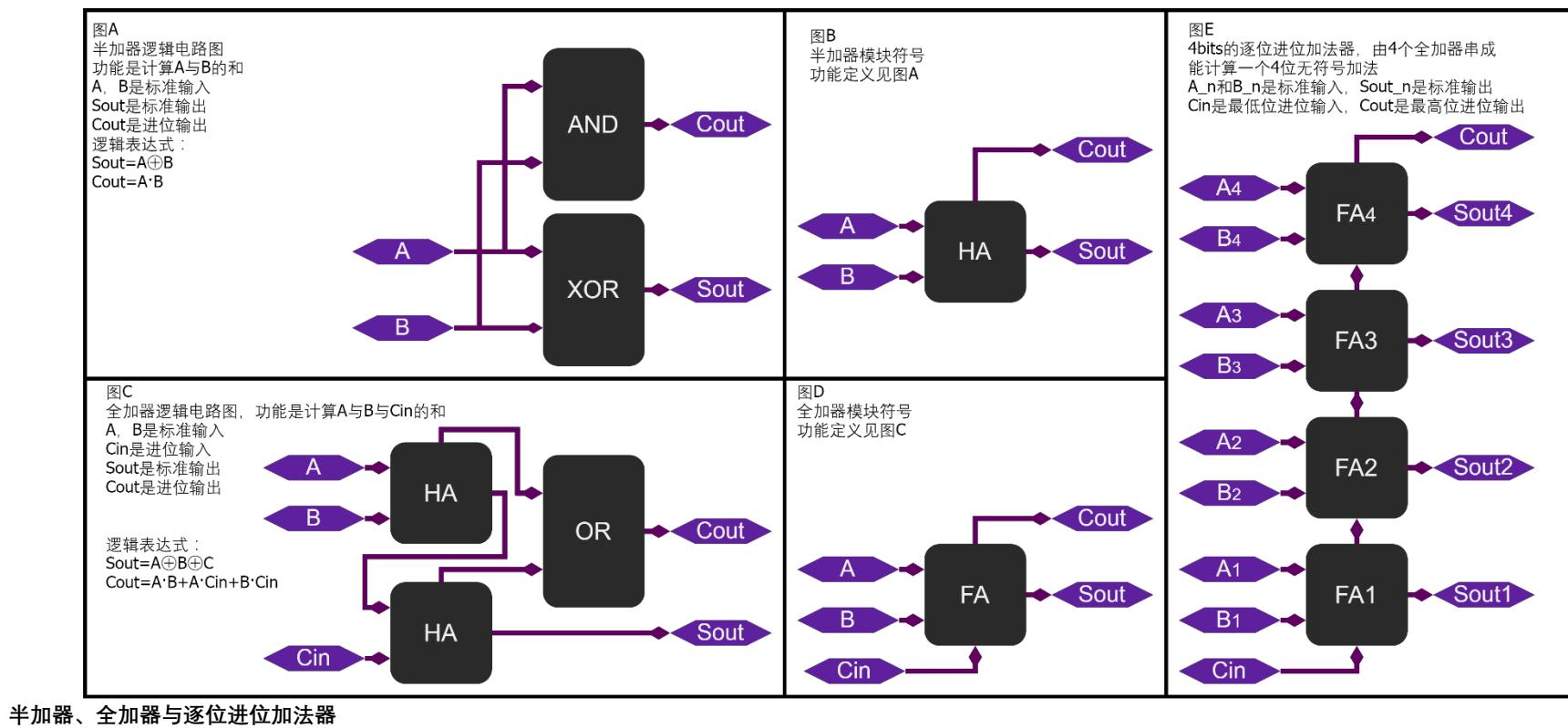
出于篇幅，这些描述术语的描述地很仓促，如果要用，读者可以通过搜索引擎详细了解它们。在后文我也会重新详细讨论它们。

这些术语主要与电路分析、验证与优化相关（如静态时序优化），初学者可能无法完全领悟这些概念的意义（其实，这些词的应用讨论与能展成好几本书），但在实践中最终会体会到它们的重要性（用到你吐！），这些重要概念值得你定时回顾！

最后，我个人总结了一些硬件算法资料，可用于复习、参考或寻找灵感：[zhuanlan.zhihu.com/p/162912128](http://zhuanlan.zhihu.com/p/162912128)

## 6.2 加法器

### 6.2.1 半加器 (HA、Half Adder) 、全加器 (FA, Full Adder) 与逐位进位加法器 (RCA, Ripple-Carry Adder)



#### 6.2.1.1 全加器 (FA, Full Adder)

FA 的电路如上图 C, 它的功能就是处理单个位的加法。联想一下二进制加法竖式的计算, FA 的功能实际上就是单个位上二进制加法的计算过程, 且包括进位的传递。

这里再多说一句, 这些模块的内部实现并不是唯一的, 只要电路的功能符合定义即可。实际上, 针对不同的实际情况, 一个模块的不同内部实现可以表现出不同的优劣势, 应该在不同实现中权衡出最合适的选择。至于如何权衡, 那又是门学问。

#### 6.2.1.2 逐位进位加法器 (RCA, Ripple-Carry Adder)

将 n 个 FA 的进位输入和输出按照上图 E 的方式分别将每位加法器的 Cin 和 Cout 串起来, 就得到了一个 n bit(s) 的二进制无符号加法器。这就是逐位进位加法器 RCA, 这个名字是根据它逐位的进位方式得来的, 它的功能就是计算两个 n bit(s) 无符号二进制数的和。

在图 C 中, A<sub>n</sub> 和 B<sub>n</sub> 分别是操作数的输入, 而 Sout<sub>n</sub> 则是运算结果的输出。图 C 中电路的功能就是计算 Sout = A + B。

此外, 图 E 中的 Cin 端能提供一个额外的 +1 输入, 当 Cin=1 时, 就有 Sout=A+B+1 (这里的+还是二进制加法的意思), 求补码的时候可以利用 Cin 来实现 +1。Cout 则是最高位的进位输出, 如果它的输出为 1, 那就说明这次计算溢出了, 可以用它来判断是否溢出。

#### 6.2.1.3 RCA 的弊端

RCA 很慢, 因为它的进位是一位一位从最低位向高位传递的, 想象一下让一个 RCA 计算 1111 1111 + 0000 0001 会怎么样? 进位从最低位产生, 逐位传递最终到达最高位并且产生溢出, 期间必须经过 7 个 FA。如果是 64 位加法器, 最坏的情况下 Cin 的进位将逐位经过 64 个 FA 逐位传递, 实在是太慢了!

RCA 速度低下的原因在于必须逐位传递进位, 那有没有办法不逐位传递进位? 当然有, 而且特别多。据我所知的就有不下十种有关的加法器设计方案。这些加法器无一例外都改进了进位链以减少进位延迟, 且不同的设计方案有不同的优缺点, 需要因地制宜。

#### 6.2.1.4 RCA 的简单分析

RCA 有一条特别长的线路, 就是最低位进位输入 Cin 到最高位进位输出 Cout 这条, 我们可以把它看作是这个电路的关键路径。如图 E, 最坏情况是: Cin 输入的信号将逐位地经过每位 FA, 最终到达 Cout。如果每跨过 1 个全加器都会经过 1 个与门和 1 个或门, 最坏情况下 Cin 到 Cout 的信号将跨过 n 个与门和 n 个或门, n 是 RCA 中 FA 的数量。

RCA 的优势在于它的元件数量特别少, 一个 n bit(s) 的逐位进位加法器只需要将 n 个 FA 的进位链串联起来即可。

RCA 的内部信号很不同步, 假如 A、B 输入端同时输入信号, 那么每个 FA 都同时可能产生输出和进位输出, 但进位输出又会输出到下一个加法器的进位输入, 于是下一个加法器的输出和进位输出又被改变...直到有一个 FA 不产生进位, 这种震荡才结束。

#### 6.2.2 超前进位加法器 (CLA, Carry Lookahead Adder)

CLA 的思路很暴力, 就是把整个进位链彻底展开成并行电路。想一下, RCA 中哪些情况下第 n 个全加器的进位输入为 1? CLA 展开进位链的办法就是用逻辑门分别判断所有进位生成的情况。

由于 CLA 的电路规模比较大, 我这里仅用公式描述它。如果你想看它的电路图的话, 可以在网上自行查找。这里推荐一个不错的资料 (它把 CLA 叫做 LCA, 其实是一个东西) : [zhuanlan.zhihu.com/p/101332501](http://zhuanlan.zhihu.com/p/101332501)

现在我们写出一个  $N$  位 CLA 对应的逻辑表达式。假设  $A_n$  和  $B_n$  是第  $n$  位的输入,  $OUT_n$  是第  $n$  位的输出,  $c_0$  是最低位的进位输入,  $c_{N+1}$  是最高位的进位输出, 则有:

$$n = 0, 1, 2, 3 \dots N-2, N-1, N$$

$$G_n = A_n B_n, P_n = A_n \oplus B_n$$

$$c_1 = G_0 + c_0 P_0$$

$$c_2 = G_1 + c_1 P_1 = G_1 + G_0 P_1 + c_0 P_1 P_0$$

$$c_3 = G_2 + c_2 P_2 = G_2 + G_1 P_2 + G_0 P_2 P_1 + c_0 P_2 P_1 P_0$$

$$c_4 = G_3 + c_3 P_3 = G_3 + G_2 P_3 + G_1 P_3 P_2 + G_0 P_3 P_2 P_1 + c_0 P_3 P_2 P_1 P_0$$

...

$$c_N = G_{N-1} + c_{N-1} P_{N-1} = \dots$$

$$c_{N+1} = G_N + c_N P_N = \dots$$

$$OUT_n = P_n \oplus c_n$$

为了让公式更简洁, 我让下标  $n$  从 0 开始而不是从 1 开始, 就是说最低位输入输出分别是  $A_0$ 、 $B_0$ 、 $OUT_0$  而不是  $A_1$ 、 $B_1$ 、 $OUT_1$ 。

上式中的  $c_n$  就是 CLA 的进位链部分,  $c_n$  最右边就是一大堆的与运算和或运算, 如果对应到电路中, 就是一大堆的与门和或门。所以 CLA 的元件数非常大。随着位数增加,  $c_n$  最右边的式子会越来越长, 对应电路的规模也会越来越大。

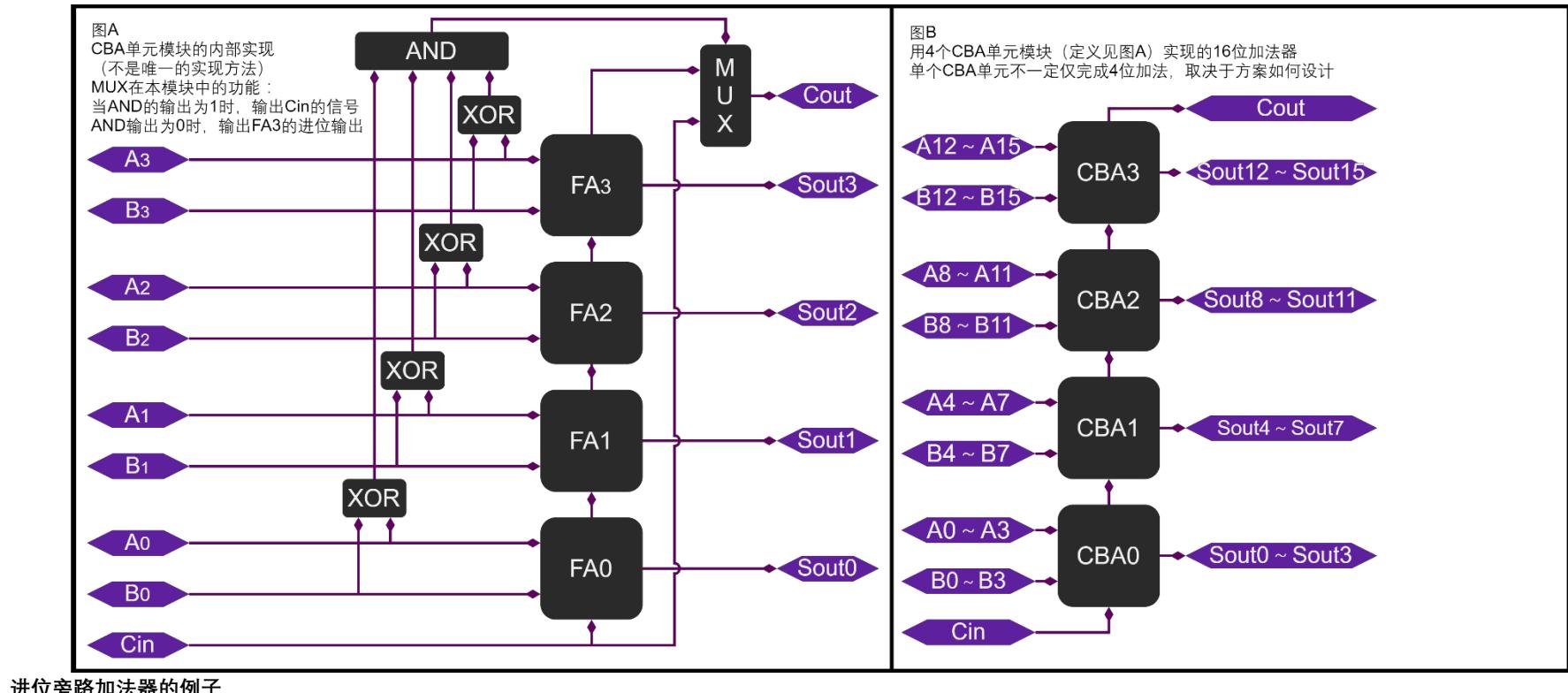
运算时可以直接根据  $G_n = A_n B_n$ ,  $P_n = A_n \oplus B_n$  得到  $G_n$  和  $P_n$ 。然后根据  $c_n$  最右边的那个越来越长的式子直接得到  $c_n$  的值, 再根据  $OUT_n = P_n \oplus c_n$  直接得到运算输出。这样就不需要逐位传递进位了, 也就不需要担心进位传递的延迟了。

### 6.2.2.1 CLA 的分析

CLA 能直接同时生成所有位的进位输入 (式中的  $c_n$ ), 而不需要花时间传递进位。但它的缺点也很明显, 就是电路的规模特别大! 请想象一下 64 bits CLA 的样子。况且, 导线延迟也是一个不可忽视的问题。

RCA 和 CLA 可以算是两个相反方向的极端。前者规模小但延迟大, 后者规模大但延迟小。但在实际情况中, 一般既不能让延迟太大, 也不能让规模太大。所以在实际设计中通常会选择更为折中的方案, 将延迟和规模都控制在合理范围内。

### 6.2.3 进位旁路加法器 (CBA, Carry Bypass Adder)



进位旁路加法器的例子

CBA 的结构如上图。相较于 RCA 和 CLA, 它是一种更为折中的设计, 更适合用于位数高一些的加法器的实现。RCA 需要逐位传递进位, 图 A 中的 CBA 单元内部也是如此。但在 CBA 单元间 (如图 B), CBA 通过旁路让跨位较大的进位跳过了 CBA 模块内 FA 的进位链, 从而大大减少了较坏情况下的进位延迟。

想象一下 64 bits 的加法器, 如果是 RCA, 那么最低位进位输入到最高位的进位将经过 64 个加法器, 但如果是 CBA, 则只需要经过大约  $\frac{64}{n}$  个 CBA 单元 ( $n$  是单个 CBA 单元的位宽)。由于进位信号可以直接跨过 CBA 单元内部。CBA 得以在不增加过多元件数的情况下, 大大减少了进位延迟。

组成 CBA 的每个 CBA 单元的位数不一定非得一致, 比如可以让第一个 CBA 单元是 5bits 的, 第二个却是 3bits 的。

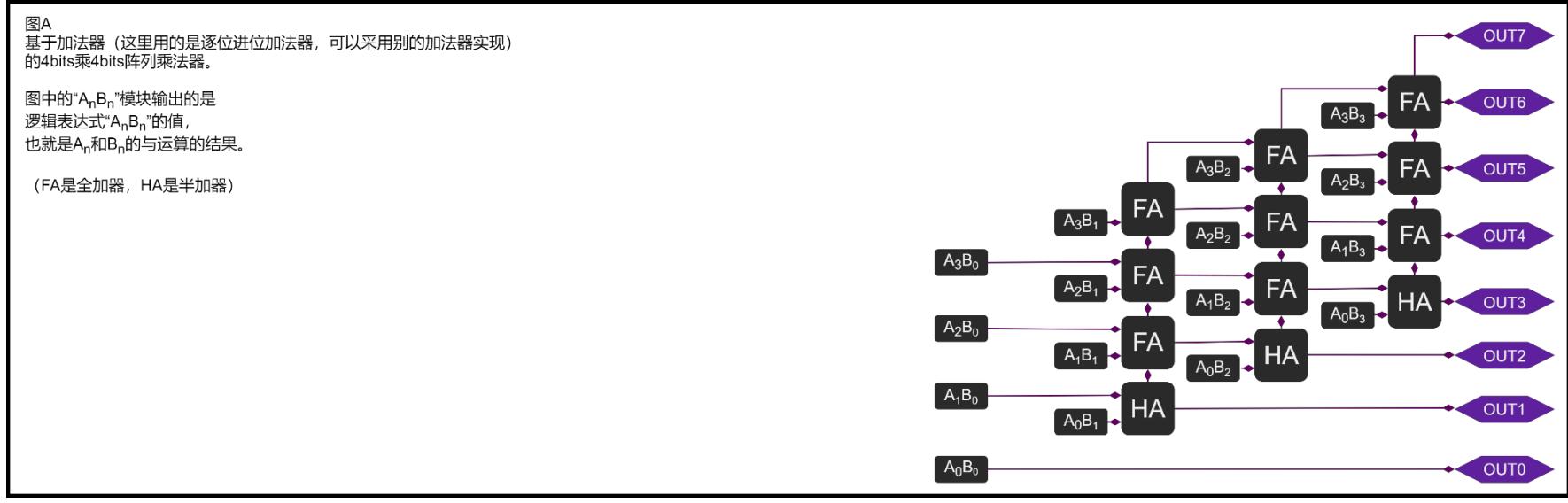
也可以在 CBA 单元内用 CLA 实现 CBA 单元的功能, 低位数 CLA 的规模还不是非常大。

还可以在多个 CBA 单元之间添加旁路, 让这些 CBA 单元组合变成一个更大的 CBA 单元。这在大位数的情况下可以进一步改善性能。

有关加法器的讨论在此暂告一段落, 更多的加法器方案会在将来中讨论。

## 6.3 乘法器

### 6.3.1 基于加法器的阵列乘法器 (Array Multiplier)



阵列乘法器的例子

请回忆一下二进制乘法竖式的计算过程，这个阵列乘法器的原理跟竖式是一样的。令  $A_n$  和  $B_n$  是二进制数  $A$  和  $B$  的第  $n$  位数， $n$  从 0 开始数，则有：

$$w_n = 2^n$$

$$A \times B = w_0 \cdot A \cdot B_0 + w_1 \cdot A \cdot B_1 + w_2 \cdot A \cdot B_2 + w_3 \cdot A \cdot B_3 + w_4 \cdot A \cdot B_4 + \cdots + w_n \cdot A \cdot B_n$$

你应该还记得  $w_n = 2^n$  其实就是二进制位权的公式。（这里为了公式的简洁性， $n$  也是从 0 开始数的）

于是再根据移位运算在不导致溢出情况下的性质  $A \ll s = A \times 2^s$ ，就能得到：

$$w_n \cdot A = 2^n \cdot A = A \ll n$$

$$A \times B = (A \ll 0) \cdot B_0 + (A \ll 1) \cdot B_1 + (A \ll 2) \cdot B_2 + (A \ll 3) \cdot B_3 + \cdots + (A \ll n) \cdot B_n$$

这样就用更为简单的移位运算代替了部分乘法操作，而  $B_n$  非 1 即 0，因此：

$$(A \ll n) \cdot B_n = \begin{cases} 0, & B_n = 0 \\ A \ll n, & B_n = 1 \end{cases}$$

可以用与门来完成这步运算：当  $B_n=0$  时候，输出 0。 $B_n=1$  时，输出  $A \ll n$  的值。有关移位，实际上把线路歪一下（第  $n$  位输出接到第  $n+1$  位输入）就能实现这个移位运算，就如上图那样。最后再将每个  $(A \ll n) \cdot B_n$  用加法器加起来，就得能到运算结果了。整个过程只用到了移位加法还有与位运算，这就是上图中阵列乘法器的原理。如果你不喜欢公式，记住电路图就行了，这个电路图很规则。

最后一步中用到了  $n-1$  个  $n$  bit(s) 的加法器来完成所有项的求和。求出  $n$  个数的和的模块可以叫做  $n$  操作数加法器。（这个  $n$  就是有多少个被加数的意思）图中的阵列乘法器的做法是将  $n-1$  个加法器的输入输出串联以求出所有项的合，但  $n$  操作数加法器除了这种做法以外，还有其他一些性能更好的设计（如基于结合律的），马上就会讨论。

#### 6.3.1.1 基于加法器的阵列乘法器的分析，在运算器中运用交换律

首先将它的公式写出来，我们令  $B$  是一个  $n$  bits 的二进制数， $A$  是一个  $k$  bits 的二进制数：

$$A \times B = (A \ll 0) \cdot B_0 + (A \ll 1) \cdot B_1 + (A \ll 2) \cdot B_2 + (A \ll 3) \cdot B_3 + \cdots + (A \ll n) \cdot B_n$$

我们发现，等式右边的项数是由  $B$  的位数控制的，也就是说  $B$  有多少位，右边就有多少个项。如果对应到电路中，那么  $B$  有  $n$  位就意味着电路中存在  $n-1$  个加法器。而  $A$  的位数  $k$  则决定了每个加法器是多少位的。

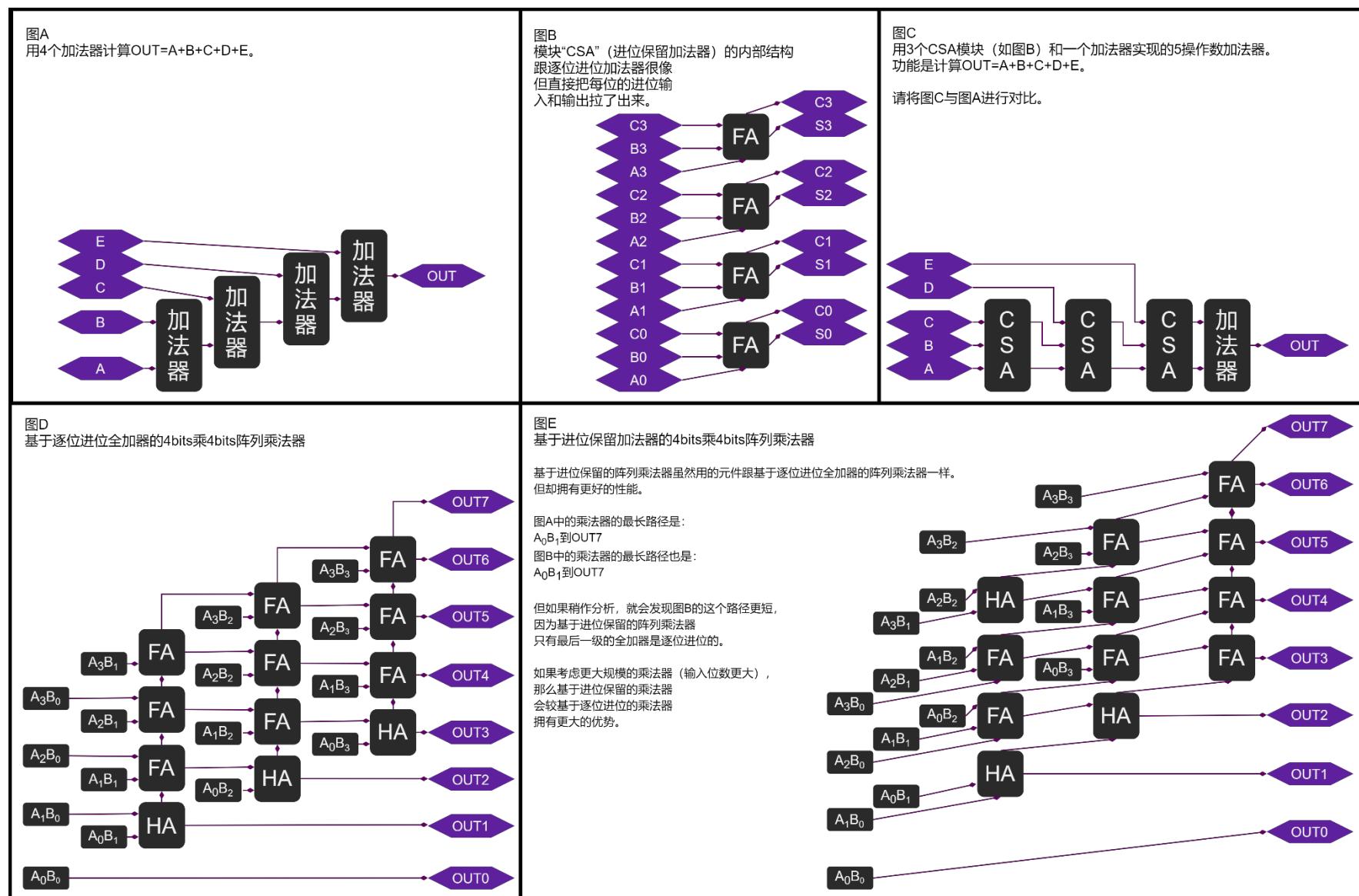
如果  $A$  和  $B$  的位数不等，那么我们可以在设计时交换  $A$  和  $B$ ，则有：

$$B \times A = (B \ll 0) \cdot A_0 + (B \ll 1) \cdot A_1 + (B \ll 2) \cdot A_2 + (B \ll 3) \cdot A_3 + \cdots + (B \ll n) \cdot A_n$$

于是情况就变成了  $A$  的位数决定电路中有多少个加法器， $B$  的位数决定每个加法器有多少位。所以在  $A$  和  $B$  的位数不等时，我们可以选择是否交换乘数与被乘数，以选择降低加法器数量或者降低加法器的位宽（输入端的位数）。交换操作数这种做法在输入位数不对称的运算器中很有用，可能仅仅只是运用了一下运算的交换律，就大大改善了模块的各种指标。

在图 A 中的阵列加法器中，第一个加法器的最低位输入到最后一个加法器的最高位输出这个路径最长，可以被认为是关键路径。当然，如果阵列乘法器用的不是 RCA，那么关键路径可能会有所不同。

### 6.3.2 基于进位保留加法器 (CSA, Carry Save Adder) 的阵列乘法器



进位保留加法器与基于进位保留加法器的阵列乘法器，还有作为对比的逐位进位加法器与基于逐位进位加法器的阵列乘法器

如上图 E，基于 CSA 设计的阵列加法器相较于图 D 中的阵列乘法器，在不增加使用元件数的前提下获得了更好的性能。

如上图 C，CSA 不仅可用于阵列乘法器中，也可用于计算 n 个数的求和。

基于 CSA 的乘法器或 n 操作数加法器的原理涉及到了冗余数表示系统，在将来会做深入的讨论。

### 6.3.3 加法树，在运算器中运用结合律

实数的加法和乘法有结合律：

$$A + B + C = (A + B) + C$$

$$A(BC) = (AB)C$$

这意味着：

$$A_0 + A_1 + A_2 + A_3 + A_4 + A_5 \dots = (A_0 + A_1) + (A_2 + A_3) + (A_4 + A_5) \dots$$

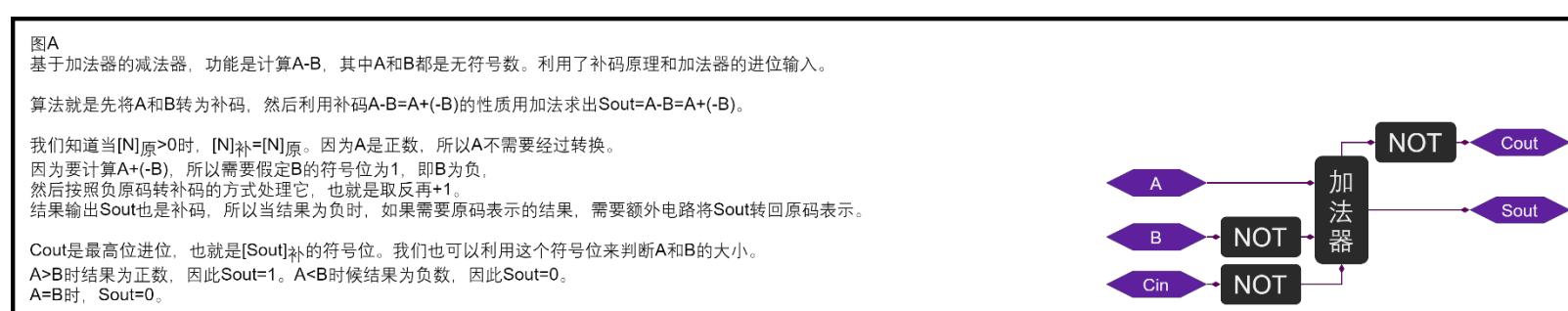
就是说如果要求出 K 个数的和，可以用  $K/2$  个 2 操作数的加法器分别独立计算  $A_i + A_{i+1}$ ，经过一轮计算后，K 个变成了  $K/2$  个数，然后再重复一次这个操作， $K/2$  个数变成了  $K/4$  个数... 经过不断的求和最终得到了结果。可以看出这种并行求和需要大约计算  $\log_2 K$  轮，如果运算器是阵列运算器的话，会用到大量的加法器，但性能上的改善是显著的。

基于这种结合律原理的电路结构叫做加法树。想象一个树的样子！树枝末端是数据输入，在树枝分叉处树枝通过运算结合律将两根或多根树枝合并为一。最终所有树枝都合并到了最下方的树干上，这个树干就对应着加法器的运算输出。

显然，只要运算满足结合律，就能在其中运用这种树型结构提升性能。加法树在乘法器和加法器等各种运算器中运用广泛。

有关乘法器的讨论也暂告一段落，更多的乘法器方案会在将来讨论。

## 6.4 减法器 (SUBTRACTOR) 与加减混合器 (ADDER SUBTRACTOR)



基于补码的减法器与加减混合器

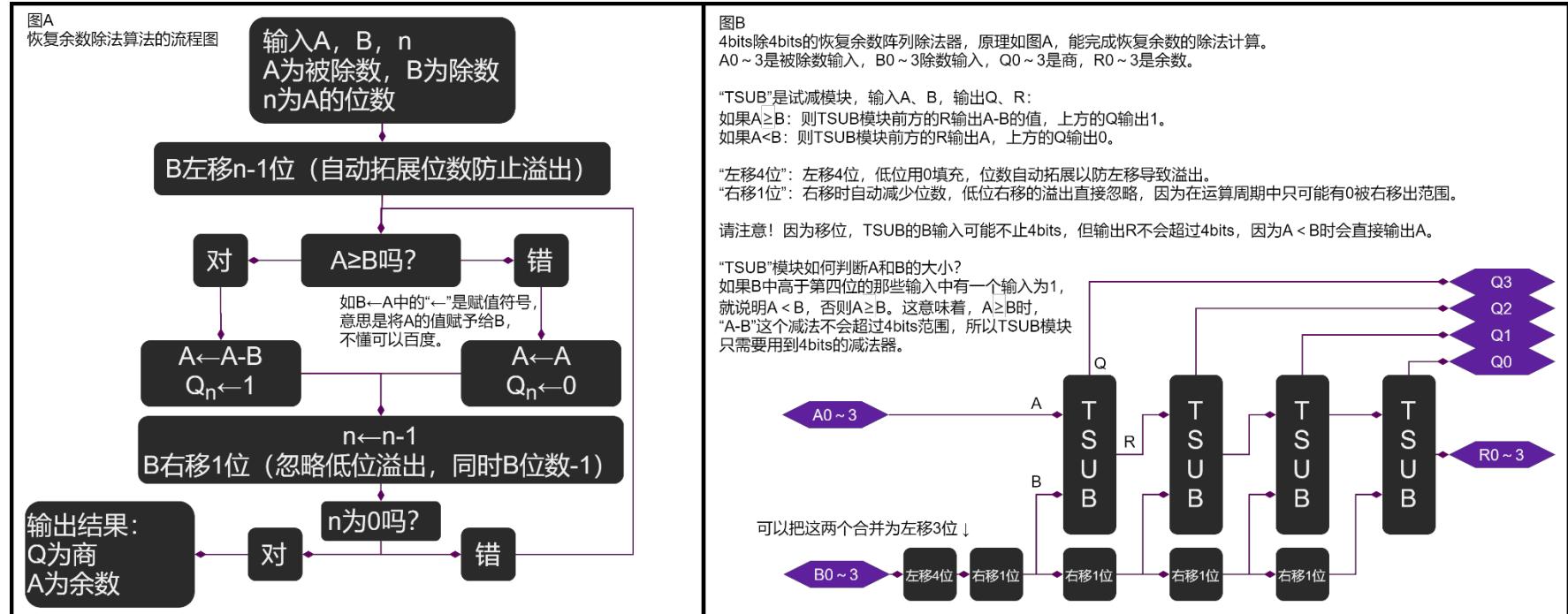
如图 A，在数字电路中，利用补码原理就可以用加法器来计算减法。如果将图 A 的每个 NOT 换成异或门，然后用一个控制线接入这个异或门的另一个输入，就可以实现加减混合器，它能控制电路完成  $A+B$  和  $A-B$  两种运算。

如果在设计之初就考虑图 A 中减法器的非门，那么可以利用逻辑化简将这些非门和加法器合并到一起，以减少一定的元件数。基于此可以设计出专用的减法器，如果需要可以参考：[zhuanlan.zhihu.com/p/112654170](http://zhuanlan.zhihu.com/p/112654170)

在数字电路中几乎所有的减法器和加减混合器（也可能没有）都是基于补码原理设计的，所以只要搞懂了补码原理和加法器的做法，你也就搞懂了减法器和加减混合器的做法。

## 6.5 除法器

### 6.5.1 恢复余数阵列除法器 (Restoring Array Divider)

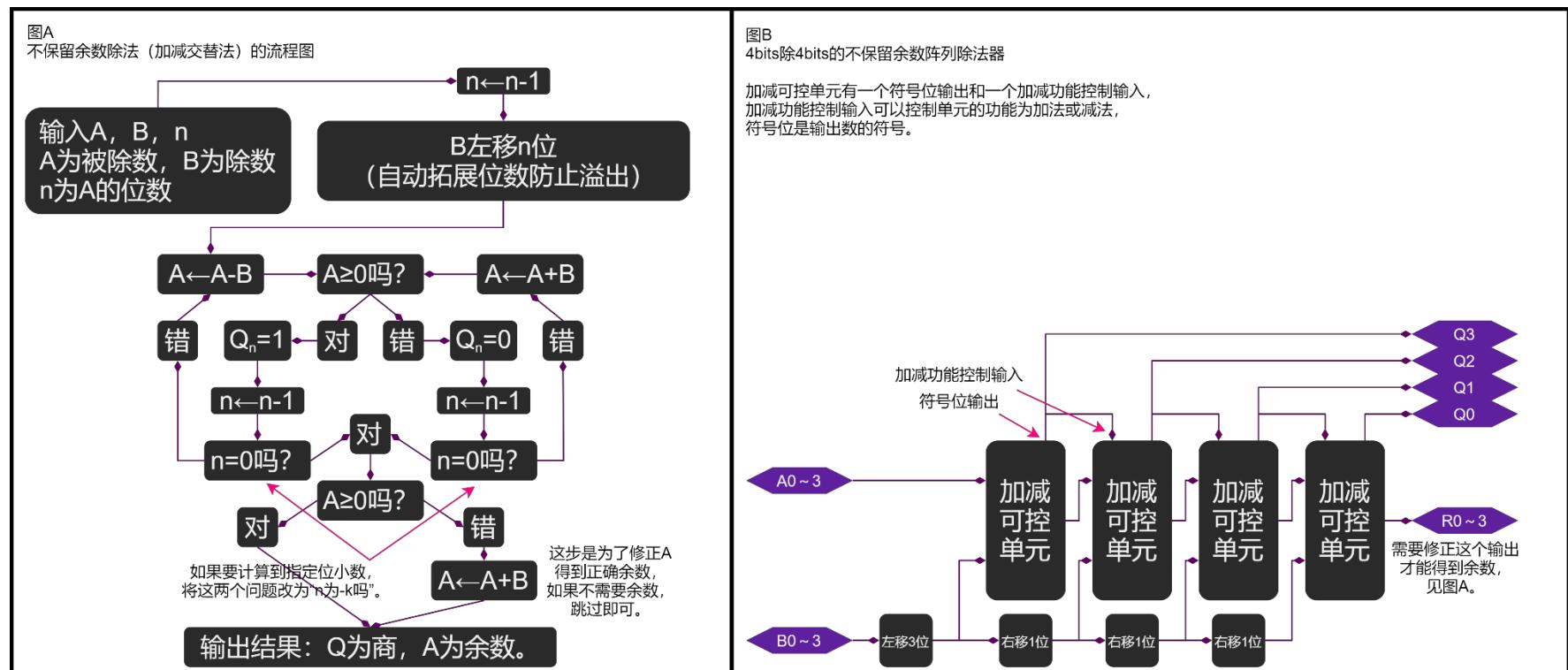


恢复余数除法算法过程与例子。图 B 可能比图 A 更加清晰易懂

请回忆一下二进制除法竖式的计算过程，这个恢复余数阵列除法器的原理跟竖式是一样的，就如图 A 中的那样。恢复余数阵列除法器能在计算商的同时保留余数。

如果需要计算商的小数部分的话，利用上除法器输出的余数结果 R 继续和 B 进行右移试减就能得到商的第 n 位小数，过程还是和二进制除法竖式一样，因此不再赘述。

### 6.5.2 不恢复余数阵列除法器 (Non-restoring Array Divider)



不恢复余数除法算法过程与例子。图 B 可能比图 A 更加清晰易懂

不恢复余数除法又名加减交替法，如上图。请注意一个事实（有关其中的移位，还是假定自动拓展位数防止溢出）：

$$A - B + \frac{B}{2} = A - B + (B \gg 1) = A - (B \gg 1) = A - \frac{B}{2}$$

加减交替法就是通过这个原理对恢复余数除法的计算过程进行一定变形得到的。有关推导很简单，感兴趣的读者可以自己尝试推导。

### 6.5.3 除法慢于乘法

相较于乘法，除法并没有那么多效率优秀的算法，所以大部分情况下除法都是慢于乘法的。实际上，有一条原则就叫尽量不涉及或少涉及除法，因为除法算法的效率并不高。避免使用除法的一个好办法就是将  $A/B$  写成  $A \times (1/B)$  以将除法化为乘法。

## 6.6 十进制加法器 (DECIMAL ADDER)

### 6.6.1 BCD 码简介

十进制数码	0	1	2	3	4	5	6	7	8	9
8421BCD 码	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001

十进制数	10	22	35	36	40	55	69	73	88	99
8421BCD 码	0001 0000	0010 0010	0011 0101	0011 0110	0100 0000	0101 0101	0110 1001	0111 0011	1000 1000	1001 1001

一些十进制数所对应的 BCD 码

BCD 码是一种用 1 和 0 表示十进制数的办法，每位十进制数用 4 个二进制位表示，上表给出了每个十进制数码所对应的 BCD 码，并且举了一些十进制数所对应的 BCD 码的例子。如上表，对于 8421BCD 码，1010、1011、1100、1101、1110、1111 这几个编码没有被使用，因为这几个未使用的编码是未定义的，它们不应该出现。

上表展示的是 8421 码，它是众多 BCD 码其中的一种，其它的 BCD 码有如 5421 码和 2421 码，但这里只讨论 8421BCD 码。

在电路和计算机中，可以用 BCD 码来直接表示十进制数，并且直接在 BCD 码下进行十进制运算。当然，对于其它进制，也同样可以用若干二进制位表示它的所有数码，道理都是一样的。

虽然十进制更符合大部分人的习惯，但十进制的有关硬件算法却无法在各方面比拟二进制的硬件算法。在数字电路和计算机中，采用二进制表示才是主流（有许多理由让工程师采用二进制作为表示，而不是其它进制）。即便在要求下输入和输出都必须是十进制的，通常的解决思路也是先将输入转为二进制数，然后再将结果转为十进制输出，以在二进制表示下完成计算。

### 6.6.2 8421BCD 码加法器



8421BCD 码加法器的例子

如上图，这个 8421BCD 码加法器的原理实际上和十进制竖式是一样的，区别在于它采用了 8421BCD 码来表示十进制数。

运用补码原理也可以将该模块改成减法器或加减混合运算器，补码原理对任何数制都可以适用，这里不再赘述。

除了十进制，其它进制也可以设计出类似的加法器，并且利用补码原理实现减法器或加减混合运算器。

实际中 BCD 码的使用很少，而且相较于二进制，BCD 码的乘法器和除法器都不太优秀，因此不做有关 BCD 码运算器的更多讨论。

## 6.7 进制转换 (RADIX TRANSFER)

### 6.7.1 特殊情况直接转换 (Direct Conversion in Special Case)

对于一些特殊情况，数制间可以直接完成相互转换。

比如二进制与十六进制，由于 4 bits 的二进制数恰好有 16 个状态，而将这 4 位二进制数看作整体时，它的进位输出又恰好是逢 16 进 1，所以每 4 位二进制数恰好可以对应 1 位十六进制数，因此它们可以直接转换。

以  $1010\ 1001_2$  为例： $1011$  所对的十六进制数是  $A$ ， $1001$  所对应的十六进制数是  $9$ ，因此  $1010\ 1001_2$  所对应的十六进制数是  $A9_{16}$ 。如果将这个例子的步骤反过来，就变成了从  $A9_{16}$  得到它所对应的二进制数  $1010\ 1001_2$ 。

对于一些其它情况，也是同理，只要  $n$  位  $k$  进制数恰好可以对应  $m$  位  $j$  进制数，那么  $k$  进制和  $j$  进制间就可以照着上面这样的步骤直接相互转换。

也正是因为如此，很多人喜欢在汇编中直接使用十六进制表示数值，因为十六进制可以像上面这样简单地对应到二进制数上，而且 1 位十六进制数就可以代表 4 位二进制数，这会让代码看起来更加紧凑。

### 6.7.2 位权相加法 (Weighted Sum)

之前我们就讨论过基于位权相加的进制转换方法。首先将一个数写成数码乘与位权的求和式，然后在目标进制表示下计算这个式子，算出的结果就是该数在目标进制下的表示。例如二进制转十进制就是在十进制下写出这个二进制数所对应的求和式，然后可以通过 BCD 码加法器在十进制下计算出这个式子的结果，最终就得到了转换结果。（如果需要，你也可以先对这个式子做个变形再计算）

这种办法实际上效率很不错，因为它的过程就是计算一个求和式，也就是相当于一个  $n$  操作数的加法。而关于  $n$  操作数的加法有许多性能优秀的算法，比如之前在阵列乘法器中用到的进位保留加法器，或者运用加法树进行求和。

### 6.7.3 短除法 (Short Division)

假如有一个十进制整数  $A$ ，要求出它在二进制下的表示。基于短除法的办法就是不停给  $A$  除以 2（带余数除法），直到商变成 0，然后将每次得到的余数（很显然这个余数非 1 即 0）从低到高排列（第一次除以 2 得到的余数在最低位），就得到了  $A$  在二进制下的表示了。

这里举个例子！如果要求十进制数 123 对应的二进制数，那么根据短除法有：

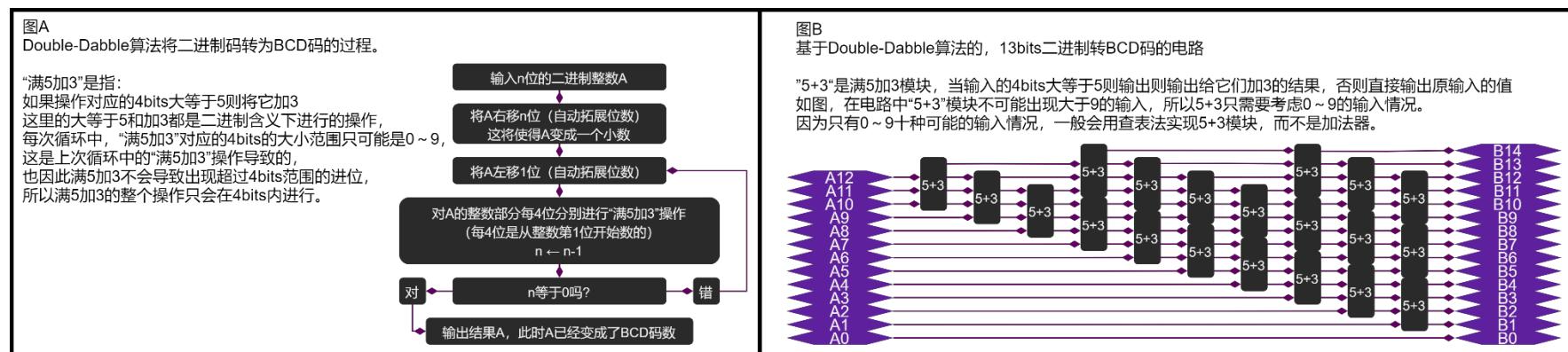
$$\begin{aligned}123/2 &= 61 \text{ 余 } 1 \\61/2 &= 30 \text{ 余 } 1 \\30/2 &= 15 \text{ 余 } 0 \\15/2 &= 7 \text{ 余 } 1 \\7/2 &= 3 \text{ 余 } 1 \\3/2 &= 1 \text{ 余 } 1 \\1/2 &= 0 \text{ 余 } 1\end{aligned}$$

将其中的余数逆序从低到高从左到右排列，就得到了十进制数 123 在二进制数下的表示，即 1111011。

这个方法不仅可用于十进制转二进制，也可以推广到其它进制间的转换，其它情况下的推广交给读者。

感兴趣的读者可以试着证明这个算法，证明过程可以参考：[zhuanlan.zhihu.com/p/268927255](http://zhuanlan.zhihu.com/p/268927255)

### 6.7.4 Double-Dabble 算法



Double-Dabble 算法过程与例子

Double-Dabble 算法如上图，它的功能就是将二进制码转为 BCD 码。

Double-Dabble 算法是可逆的，将图 B 中的电路反过来，它就变成了 BCD 码转二进制码电路。反过来的具体过程是将其中的“满 5 加 3”单元变成“满 8 减 3”（如果大等于 8 则减 3），然后所有的输入端变成输出端，输出端变成输入端。

此外，经过推广，Double-Dabble 也适用于二进制小数与十进制小数的转换。这种情况用到的是“满 8 减 3”单元，并且需要将输入的小数部分全部挪到左边（整数部分），然后在循环中不断右移，每移一次就对每 4 位分别进行一次“满 8 减 3”操作，直到结束。如果将这个过程反过来，那就变成了十进制小数转二进制小数。

Double-Dabble 算法也可以推广到其它进制间的转换，其他情况下的推广同样交给读者。

感兴趣的读者可以试着证明这个算法，证明过程可以参考：[zhuanlan.zhihu.com/p/268927255](http://zhuanlan.zhihu.com/p/268927255)

## 6.8 移位器

### 6.8.1 逻辑移位、算数移位与循环位移

名称	定义
逻辑左移	将二进制数据左移 n 位，移位时空出的低位用 0 填充。
逻辑右移	将二进制数据右移 n 位，移位时空出的高位用 0 填充。
算数左移	和逻辑左移没有区别。
算数右移	将有限位二进制数据右移 n 位，空出的高位根据移位前数的最高位填充，移位前最高位为 1 则用 1 填充，否则用 0 填充。
循环左移	将二进制数据左移 n 位，移位时空出的低位用移位时移出范围的高位填充，形成一个循环。
循环右移	将二进制数据右移 n 位，移位时空出的高位用移位时移出范围的低位填充，形成一个循环。

六种移位运算的求法

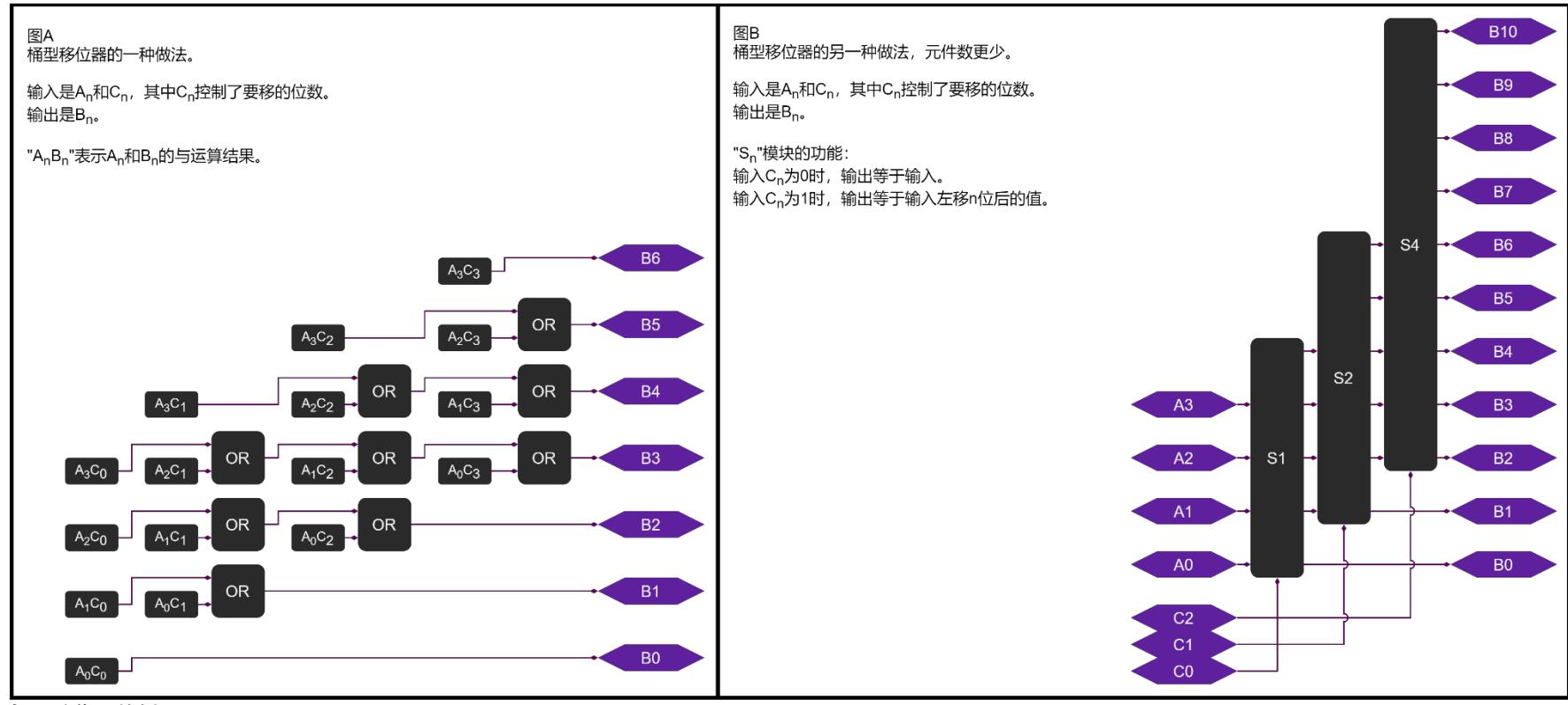
上表总结了逻辑左移、逻辑右移、算数左移、算数右移四种移位运算。

逻辑移位之前讲过了，就是直接让二进制数左移或右移（如果是有限位数，还是可能会导致溢出）。通常来说，如果要对一个无符号数进行位移，一般都会采用逻辑位移。

但如果数据是补码数，直接位移会导致符号位的错乱或者丢失，所以这时候就需要用到算数移位。算数移位能保证补码数在移位数据的正负不会错乱。（当然，算数左移可能会让一个较大的正补码数移位后超出表示范围，从而导致溢出并使数据变成负数）

循环移位较逻辑移位和循环移位用的少一些，通常用于某些特殊算法中。

### 6.8.2 桶型移位器



桶型移位器的例子

如上图，桶型移位器是一种组合逻辑电路，它的功能就是移  $n$  位。

图 A 中的桶型移位器直接用控制线  $C_n$  控制输出了左移  $n$  位后的  $A$ 。

图 B 的设计要更为明智一些，将移位分为  $n$  次操作，第  $n$  次移位由控制线  $C_n$  决定是移  $2^{n-1}$  还是将输入原封不动地直接输出。这样，如果要将输入  $A$  左移  $P$  位，直接将  $P$  所对应的二进制数输入到  $C$  输入端即可。

稍作计算就知道，对于一个能左移  $0 \sim N$  位桶型移位器，图 A 中的设计需要  $N$  列或门，而 B 中的设计只需要大约  $\log_2 N$  个  $S_n$  模块。

图中的桶型移位器仅实现了逻辑左移，如果要实现逻辑右移，将图中电路所有的移位方向反转即可。也可以将左移和右移的电路拼在一起，让一个桶型移位器同时实现逻辑左移和逻辑右移的功能。

至于算数移位，算数左移和逻辑左移是一样的，而算数右移只需要添加一点额外的的电路，将移位后空出的位根据符号位的值设置就行了。

如果要实现循环移位的话，则应该将超出表示范围的输出部分接到下方移位空出的部分。

循环移位也可以用两次逻辑移位实现，假定  $A$  是一个  $m$  bits 的无符号数，如果要让它循环左移  $n$  位，则循环移位结果应该是：

$$(A \ll n) | (A \gg (m - n))$$

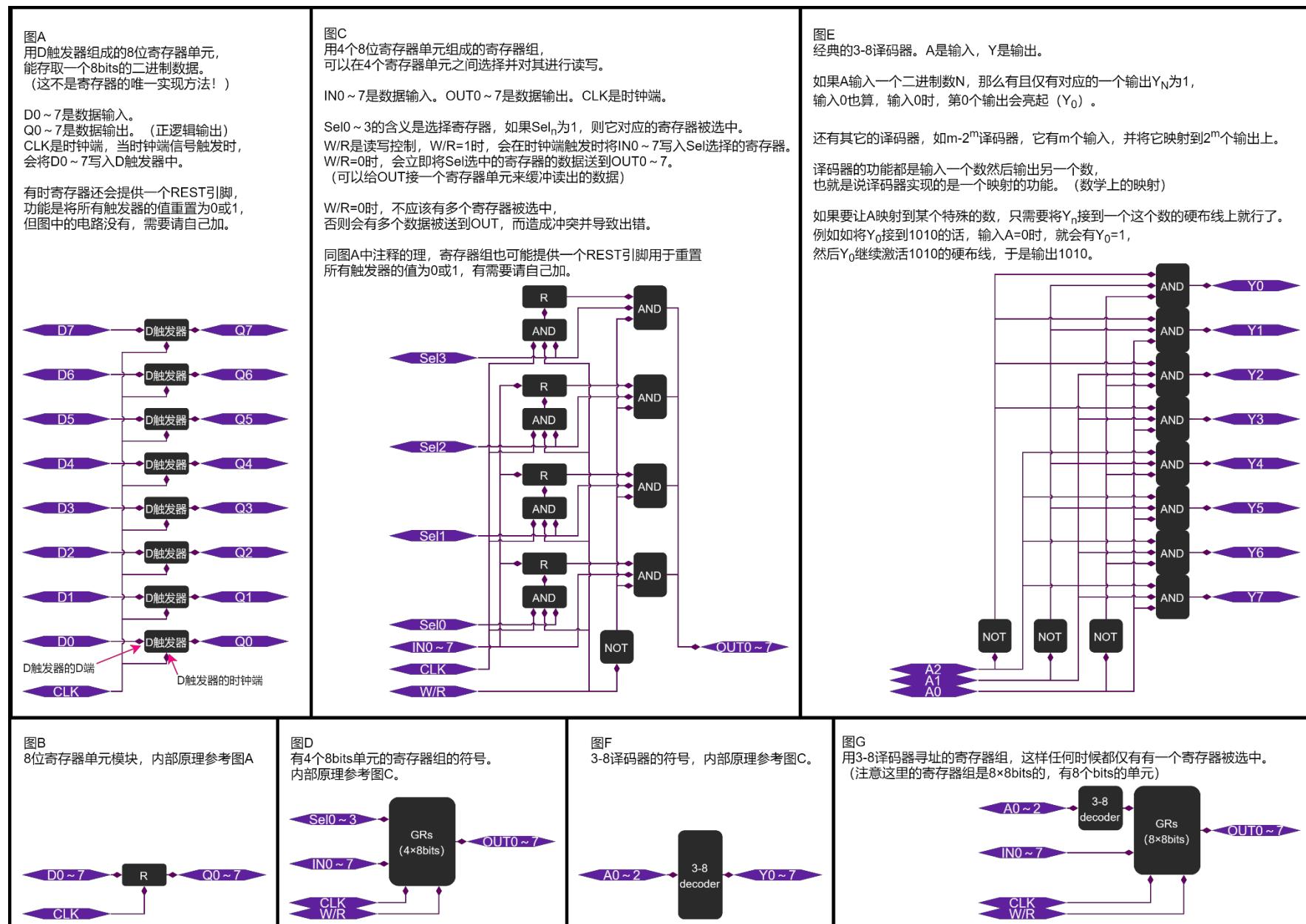
通过右移和或位运算生成并将高位部分拼接到左移时空出的低位部分了，循环右移也同理。

# 7 存储器 (MEMORY)

## 7.1 前言

这部分我们简单介绍一些存储器器件概念，存储器用于存储数据。

## 7.2 寄存器 (REGISTER)、寄存器组 (REGISTER SET) 与寻址器 (ADDRESSER)



寄存器、寄存器组与译码器的例子

### 7.2.1 寄存器 (Register) 与寄存器组 (Register Set)

寄存器是用来寄存数据的器件，如上图 A 就是一个基于 D 触发器的 8bits 寄存器，它的功能就是存储一个 8bits 的数据。

将多个寄存器组织到一起，就得到了一个寄存器组（有的地方叫寄存器堆或寄存器文件），如图 C。寄存器组能同时存储多个数据。在寄存器组中每个寄存器都有一个编号，叫做这个寄存器的“地址”，如果要操作寄存器组中的某个寄存器，首先要给出这个寄存器的地址以选中这个寄存器，然后再进行下一步操作。

有时我们会让寄存器组中的某个寄存器的数值固定为某个固定的数，比如让寄存器组中地址为 0 的寄存器的值固定为 0，这么做的好处是在需要这个数值的时候，直接访问这个寄存器就行了。且由于它的值固定为 0，即使对它写入数据它也依旧为 0，所以也可以把它作为一个数据垃圾桶使用，如果在运行时产生了没用却又不知道扔哪里的数据结果，直接让它写入这个寄存器使数据丢失就行。

在处理器中为了让寄存器组获得更高的读写效率，通常会为一个寄存器组配置多个读写通道（多通道存储器），使得它能够在一个时钟周期（触发器仅触发一次）完成多个读写请求，会在后面详细讨论。

### 7.2.2 译码器 (Decoder)

如图 E，译码器就是这样的一个功能器件，将一个数映射到另一个数。如果译码器被用于索引存储器的单元（就如图 G 那样），那么通常会把这个译码器叫做寻址器，并且把寻址器的输入数据叫做地址。顾名思义，这个地址就是寄存器单元的“地址”。

使用了寻址器后，我们就能用一个 n bits 的二进制数来索引一个存储器中的存储单元，8bits 能索引 256 个单元、16bits 能索引 65536 个单元、32bits 能索引 4294967296 个单元，这个增长是指数爆炸的。

### 7.2.2.1 级联译码器 (Concatenated Decoder)



级联译码器的例子

可以用多个小译码器级联成一个大译码器。如上图就用 2 个 2-4 译码器实现了一个 3-8 译码器，这比直接做一个大译码器更合理。

级联是一种设计思想，其目的是将一个大的电路整体解耦成由多级或多层次的小电路。

## 7.3 随机存取存储器与只读存储器

### 7.3.1 随机存取存储器 (RAM, Random Access Memory) 与只读存储器 (ROM, Read Only Memory) 的简单介绍

RAM 的功能就相当于一个寄存器组，或者从另一个角度说，寄存器组就是一种 RAM。

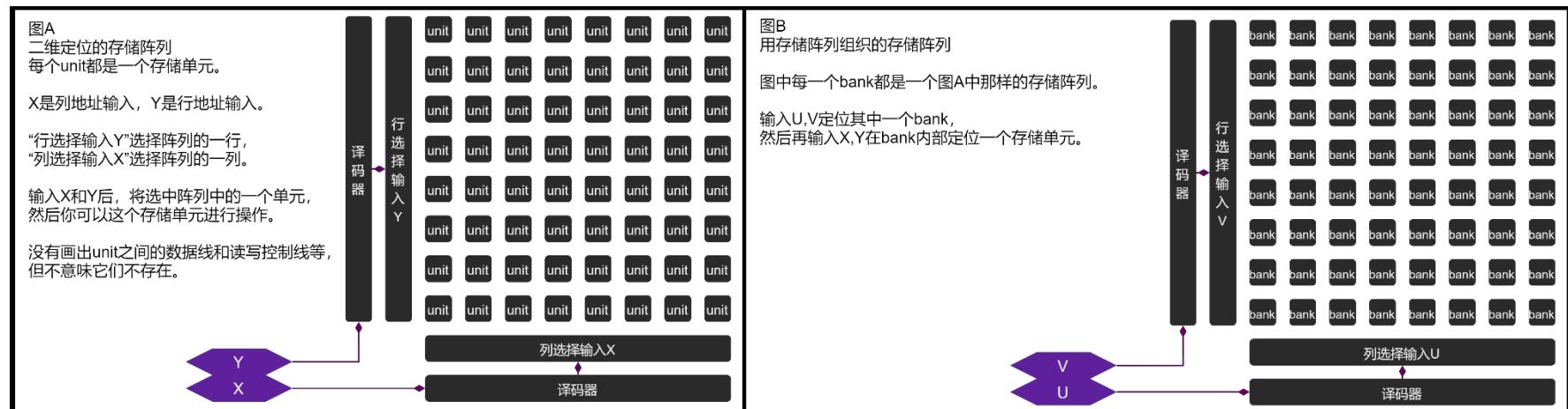
RAM 具有  $n$  个存储单元，每个存储单元都有一个唯一的编号，叫做这个存储单元的地址。给定一个地址选中其中某个单元后，可以对它的内容进行读取或者写入。

例如你的电脑内存就是一种 RAM，当然它规模相对于常见的寄存器组要大得多。一个  $4 \times 8\text{bits}$  的寄存器组（4 个 8bits 单元）的容量是 4Bytes，而你的电脑内存的容量单位动辄 GiB 起步，你还记得 Byte 和 GiB 这两个单位之间相差多少数量级吗？

ROM，中文名叫只读存储器。跟 RAM 差不多，也有  $n$  个存储单元且各有一个唯一的地址编号，区别是只读存储器只能读取单元中的数据，而不能将数据写入单元中。ROM 中的数据都是事先写入而不能更改的。（其实也有一些 ROM 具有一定的数据擦写能力）

例如生活中的 DVD 光碟就是一种 ROM。

### 7.3.2 大规模存储器的组织



二维寻址的内存阵列与内存层次示意图

当 RAM 的规模较大时，为了让 RAM 拥有更好的性能，有必要对它的存储结构进行一定的组织。在上图 A 中，RAM 中的存储单元被排列成二维的阵列，并且通过用两个寻址器来对其中的单元进行 XY 定位。

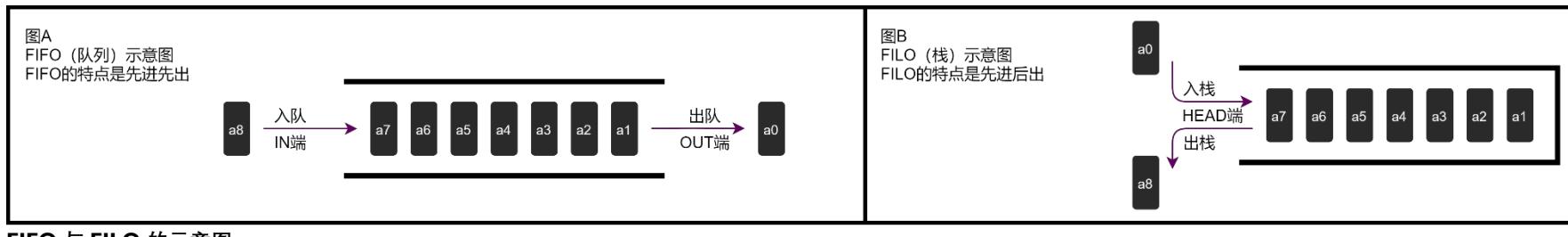
将图 A 的二维阵列封装后，再像图 B 中的那样排列它们，这个存储器内部就组织出了更多的层次。要在图 B 中的存储器里访问某个存储单元，首先需要在图 B 中的阵列上输入二维地址(U,V)来选中其中的一个 BANK，然后在一个 BANK 内再输入二维地址(X,Y)来定位其中的一个存储单元，然后才对这个存储单元进行读写操作。

然后你也可以把图 B 中的阵列当作一个单元，并且继续这样套娃下去。现实中我们电脑的内存条就是以这种分层次的方式组织的。

有相当多的理由让我们以这种多层次的方式组织大规模存储器，但具体为什么暂不做讨论。当然我建议读者不妨以一个存储器的设计者/生产者的视角来看待这个问题，不过读者可能对有关的设计和生产不太了解。

除了二维阵列以外，你也可以选择其它的组织方式，例如三维阵列或者一维阵列甚至三角阵列（开玩笑的），或者也可以混合使用多种组织方式。如何组织存储体见仁见智，主要取决于实际情况，最终目的还是改善存储器各方面的指标。

## 7.4 FIFO 与 FILO



FIFO 与 FILO 的示意图

### 7.4.1 FIFO

FIFO 是一种特殊的存储器，你也可以叫它队列（Queue）。见图 A，它的特点就是先进先出。

队列有两个端口用于存取数据，不妨分别叫它们 IN 端和 OUT 端。你能作的就是不断将数据从 IN 端塞入队列中，然后再从 OUT 端取出它，最先进入队列的元素会最先被取出。

如果要做一个直观的比方，那就把它想象成游乐场入口的队列，很明显先进入队列的人肯定总是要比后进入队列的人先进入游乐场。（当然，插队是禁止的！）

FIFO 至少有两种最基本操作，分别是入队和出队。出队就是通过 OUT 端从队列读取一个数，入队就是将数据从 IN 端送入 FIFO 中。如果对应到上面的比喻，入队就是人从队列的末尾入队，出队就是人在银行窗口完成事务离开队列。

FIFO 的用途有许多，主要是用来缓冲数据。

### 7.4.2 FILO

FILO 又名栈（Stack）。见图 B，FILO 与 FIFO 类似，但区别在于 FILO 是先进后出的。

FILO 类似有口无肛的腔肠动物，它只有一个端口，你也只能从这个端口写入/读取数据，不妨把这个端口叫做 HEAD 端。

FILO 也至少有两种基本操作，分别是入栈（PUSH）和出栈（POP）。入栈就是把元素放入栈的顶部，出栈就是把 FILO 最前的元素取出来（也就是上次入栈的元素）。

FILO 主要用于暂时存放一些数据，有时也会在一些算法中得到应用。有一种类型的计算机叫堆栈机，它们就是以栈为基础的，但这种计算机仅在上世纪活跃过一段时间，现在的堆栈机只是一种较为冷门的非主流体系。

## 7.5 查找表与内容可寻址单元

### 7.5.1 查找表 (LUT, Look Up Table)

LUT 的本质就是一个 RAM。把数据事先存入这个 RAM 中后，输入地址 n 就能得到这个地址所对应单元内的数据。LUT 的用途非常广泛，它可以用来存储一些经常用到的常数，或者用 LUT 来实现某个映射。（映射输入是地址 n，输出是对应单元的内容）

如果将一个逻辑电路的真值表存于 LUT 中，那么这个 LUT 将可以模拟这个逻辑电路。因此 LUT 还是 FPGA（现场可编程门阵列）的重要基础组成单元。

### 7.5.2 内容可寻址单元 (CAM, Content Addressable memory)

CAM 相当于升级版的 LUT。它的每个存储项都由两个存储单元组成，第一个存储单元存储的内容叫 TAG，第二个存储单元的内容叫 DATA。（如果你学过编程，可以把 CAM 当作编程中的关联数组）

CAM 可以进行匹配搜索。首先对 CAM 输入一个 TAG，然后这个 TAG 会与 CAM 中每个存储项的 TAG 进行匹配，如果输入的 TAG 跟某个存储项的 TAG 匹配了，那么则输出这个被匹配的存储项的 DATA。如果有任何一个存储项与其匹配，CAM 则会输出一个信号表示匹配失败。

此外，CAM 也应该支持普通的访问方式（每个存储单元有一个唯一的物理地址），以用于修改每个存储项的 TAG 和 DATA 的值。

当你需要从一大堆数据项中找到某个匹配的数据项时，就可以使用 CAM 来提高效率。

## 7.6 带特殊功能的寄存器

有时我们会给寄存器添加一些特殊功能以让寄存器本身获得一定的数据处理功能，这在一些时候会很方便。

### 7.6.1 计数器 (Counter)

计数器就是带有自增功能的寄存器（给寄存器内部的数据自增），自增功能就是给寄存器里的数值“+1”，如控制自增功能的引脚收到一个下降沿信号后给寄存器内的值+1。有时也会让它提供自减也就是“-1”的功能。当然需要的话让它提供“+n”或“-n”功能也可以。

另外说一下，对于二进制计数器，如果要让它 $\pm 2^n$ ，可以让计数器从第 n 位开始+1 或-1，这样 $\pm 2^n$ 的开销还是跟普通的±1一样。

同理对于 k 进制计数器，让它 $\pm k^n$ 也可以有同样的做法。

#### 7.6.1.1 计数器的实现

实现计数器最简单的办法就是直接用一个加法器来实现+n 的功能，但是实际上有更好的办法。

大部分情况下计数器都只需要+1 或者-1 这两种功能。你可以先建立一个完整的加法器，然后仅利用最低位进位输入 Cin 来实现+1 功能，这时你会发现加法器有一半的输入端是用不到的，删去多余的电路部分后，就留下了一个仅具备+1 功能的电路了。对于-1 而言，也是在减法器上进行相同的步骤。

此外，有关+1 还有另外一种算法：从最低位开始不断取反，直到遇到某个位为 0。（遇到的这个位也会被取反）

-1 也有类似的算法：从最低位开始不断取反，直到遇到某个位为 1。（遇到的这个位也会被取反）

如果要实现 $\pm 2^n$ ，请从第 n 位开始执行这个算法。（最低位是第 0 位）

基于触发器的“异步计数器”和“同步计数器”电路就是基于该算法的，相关电路可以自行搜索查找。

### 7.6.2 移位寄存器 (Shift Register)

移位寄存器就是自带移位功能的寄存器（可以让寄存器内部的数据移位），它自带左移或者右移或者两种移位功能。具体来算移位还分算数移位和逻辑移位等等各种不同的移位运算，总之需要多少功能你就加多少功能。

许多情况下电路只会用到带有左移 1 位或者右移 1 位功能的移位寄存器，这时只需要把线歪一下（第 n 位输出接到第 n+1 位输入实现左移，或接到第 n-1 位输入实现右移动）就行了。对于仅带有左移或者右移 $2^n$ 位功能的移位寄存器来说，也是同理。

#### 7.6.2.1 二维移位寄存器 (Two Dimension Shift Register)

二维移位寄存器是在一个二维的 bit 阵列上进行移位操作的寄存器。想象一个由 1 和 0 摆成的方阵，让它所有的元素同时朝某个方向移动（上、下、左、右）的样子，这就是二维的移位寄存器。多维移位阵列也同理。

二维或者多维的移位寄存器在实际中用的比较少，主要运用在某些特殊的算法或者张量的运算器中。

### 7.6.3 线性反馈移位寄存器 (LFSR, Linear Feedback Shift Register)

LFSR 是一种常用于生成伪随机数的器件，它是基于移位寄存器的。（一维的，当然你要做多维的也不是不可以）

移位寄存器每次会左移 1 位，左移 1 位后最低位会空出来，在一般的移位运算中我们会用 0 来填充这个空位。但在线性反馈移位寄存器中，我们使用 F(s) 来决定这个位该填 1 还是填 0。在 F(s) 中，s 是先前寄存器中的值，而 F 是一个映射到 1bit 的线性函数。

一般的做法就是用多个异或门构成组合逻辑电路将 s 的每位作为输入进行处理，最后仅输出一个 1bit 的值，然后将这个输出作为结果填充到移位时产生的空位，由此来实现函数 F。

赋予给寄存器的初始值叫做“种子”，显然这个种子确定了接下来 LFSR 会生成的伪随机数。

#### 7.6.3.1 非线性反馈移位寄存器 (NLFSR, Non-linear Feedback Shift Register)

顾名思义，NLFSR 就是将线性反馈移位寄存器中的线性函数 F 改为非线性函数 G。对于 NLFSR，一般的做法是采用多个与门和异或门构成组合逻辑电路来实现函数 G。

除生成伪随机数外，LFSR 和 NLFSR 还在加密、纠错等方面有用。关于它们就不做更多介绍了，如果你要用到它们，可以自行了解。

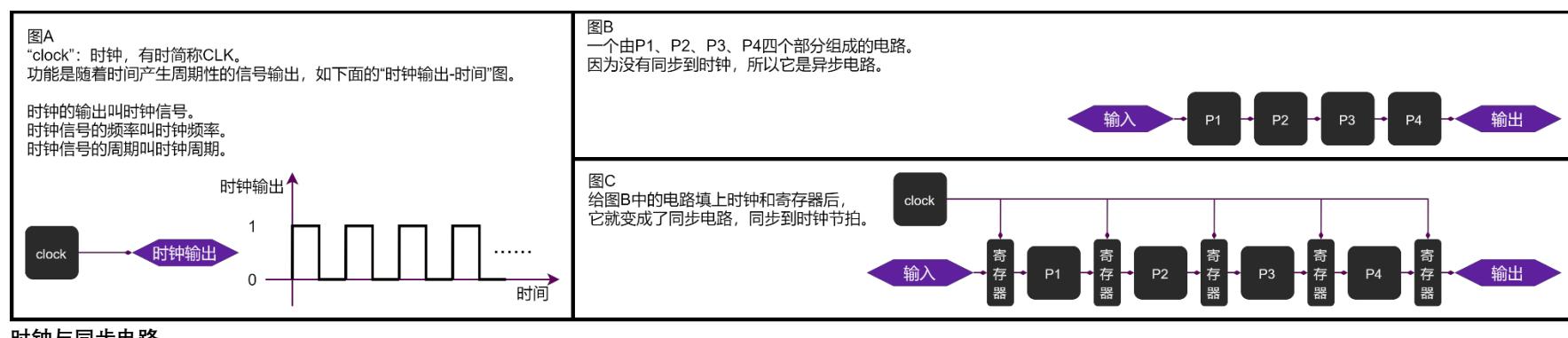
# 8 时序电路设计

## 8.1 前言

第三章介绍了基础运算器，第四章介绍了存储器，如果再给它们加上适当的时序，就可以设计出更为复杂的电路系统。

## 8.2 时序电路基础

### 8.2.1 时钟 (CLK, Clock) 、同步 (SYNC, Synchronization) 与异步 (ASYNC, Asynchronous)



如图 A，时钟是一种产生周期信号的重要元件，它能提供一个标准的节拍来指挥其它电路工作。如果电路是按照时钟信号的节拍同步工作的，那么就说这个电路是同步电路，否则就说这个电路是异步电路。

图 B 是一个异步电路，给他插上一些寄存器并且接入时钟后，它就变成了同步电路（见图 C）。

图 C 中的电路是这样工作的：电路中所有被时钟驱动的寄存器会在每个时钟周期将输入端的数据写入寄存器中（寄存器是 D 触发器构成的，而 D 触发器是边沿触发的），且由于寄存器的值被改变，寄存器间的电路又会接收到来自寄存器的新输入，于是中间电路产生的结果并在下个周期写入下个寄存器内。

图 C 中电路的工作过程就像是富士康流水线一样，原材料（输入数据）一个接一个地进入传送带（寄存器），经过不同工序（寄存器间的电路）最终运出流水线（输出数据）。这种电路结构在数电中可以叫做管线（英文名 pipeline，有的地方直接叫它流水线）。

异步电路与同步电路各有不同的特点和优缺点，这里对比其中的一些：

同步电路的特点和优缺点	异步电路的特点和优缺点
电路按照时钟节拍有序工作。	电路不按照时钟节拍工作。
需要等待时钟节拍。	不需要等待时钟节拍。
电路之间根据节拍工作，不需要握手信号。	电路间需要握手信号确定对方的输入输出是否准备就绪。
电路时序更容易分析。	电路时序不容易分析。
电路可以通过超频或降频（调整时钟频率）以调整电路频率。	不能直接调整电路频率。

### 同步电路与异步电路一些特点和优缺点的对比

实际集成电路设计中同步电路和异步电路是相互混用的，如图 C 中虽然寄存器被时钟驱动，但寄存器间的电路并没有接受时钟信号，所以其实这些中间的电路依旧是异步电路。只不过相较于整个都是异步电路的图 B，图 C 中的异步电路仅存在于局部。

实际设计中，一般在局部以组合电路为主，但整体电路会同步到一个时钟以协调电路各部分有序工作，从而获得同步的许多好处。如果某些部分比较慢，可以让它每 k 个时钟节拍运行一次，而不是每个时钟节拍都必须工作一次。（这里的 k 是正整数）

一个时钟震荡的频率是固定的，但利用分频器可以产生 k 倍周期长度的时钟信号。分频器的原理就是一个计数器，接受时钟信号的输入并对其记数，每收到 k 次节拍震荡 1 次。

你也可以让电路在时钟信号的上升沿和下降沿都工作一次（即采用双边沿触发方式），这样就能让它直接获得两倍于时钟频率的工作频率。（在 DDR 内存中得到了运用）

如果两个电路的时钟频率是一样的，那么它们很好连起来。如果不一样，比如 A 的频率是 B 的 3.33422 倍，那它们之间就不怎么同步，必须让其中一方升频或降频把频率统一到整倍数比才能将 A 和 B 很好地直接连接起来。（当然相位也要考虑的！）

实际设计中的同步电路结构跟图 C 中的结构是类似的，用时钟驱动许多的寄存器暂存中间结果，然后在寄存器间设置电路（图中的 P<sub>n</sub>）实现功能。一般来说寄存器间的电路都是性能差不多纯组合逻辑电路。

如果某个组合逻辑电路太大，不妨在其中间插入一些时钟驱动的寄存器以解耦它，否则它可能无法较好地匹配高速的时钟。当然，以一个恰当的方式将一整块电路适当地划分为多个部分也是一个非常值得思考的问题。

之所以寄存器间基本都是纯组合逻辑电路，是因为寄存器和时钟节拍它们本身已经为电路提供存储和时序的功能了。如果在寄存器间涉及更复杂的电路，就容易由于局部过于复杂而拖累整体。想象一下，如果富士康的生产流水线某个环节出错或速度慢了，整个流水线的运作都将受其影响。

## 8.2.2 节拍发生器 (Sequence Timer)

节拍发生器又名时序脉冲发生器，它是一种产生周期时序脉冲的器件，产生的脉冲可作为控制信号用于指挥电路各部分协调工作。它产生的时序脉冲输出其实就是一个非 1 即 0 的周期数列，如依次输出 1,0,1,1,0,1,0,1,1,0,...，这个周期数列的最小正周期是 1,0,1,1,0。这个最小正周期是可编程的，电路需要什么样的时序控制信号，就给它设置什么样的周期。

节拍发生器是时钟驱动的，每时钟节拍都产生一次脉冲。（1 或者 0，由它的周期设置决定）

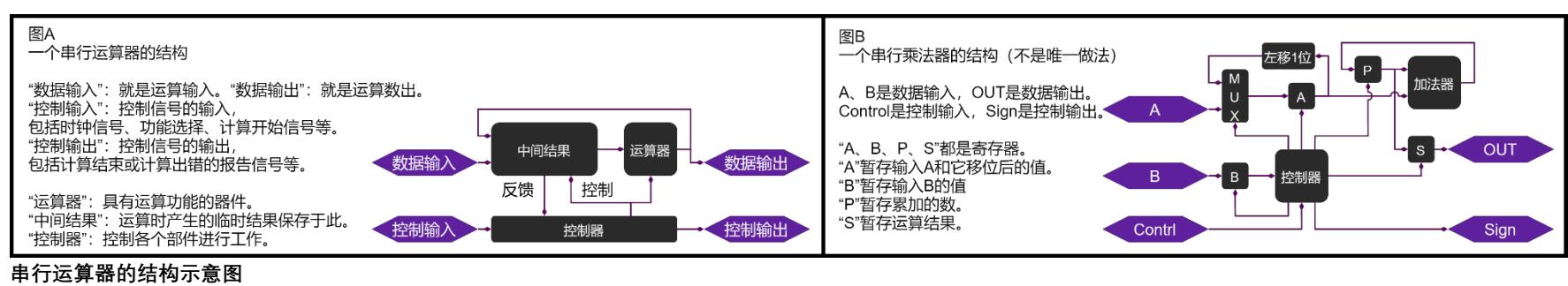
节拍发生器的电路实现就是一个循环移位（最高位左移会被移到最低位，形成一个圈。最低位右移同理）的移位寄存器，受时钟信号驱动在时钟信号边沿位移，然后我们具体把寄存器中的某一位（如最高位）的输出引出来，作为节拍发生器的输出。如果需要调整它的周期，只需要更改移位寄存器里的值就行了。（如果你需要让它重新从第一位开始循环输出的话，需要将寄存器里的循环周期重置为初始状态）

如果电路需要更复杂的时序的话，可以同时配合时钟、分频器、节拍发生器、计数器、ROM 等来实现更复杂的时序发生器。

## 8.3 串行运算器

第三章中的阵列运算器中大量重复运用了某些模块，就连基本的加法器（如逐位进位加法器）也大量重复使用了全加器模块。如果我们仅保留阵列运算器中大量重复模块的其中一个或几个，然后通过时需设计让数据集中在几个模块上重复进行计算，是不是就能大量减少电路规模？使用单个模块反复完成多次操作，这就是串行运算器的思路。

### 8.3.1 串行运算器的结构



串行运算器的结构示意图

串行运算器的结构如上图 A，串行运算器的工作流程是这样的：

1. 通过控制输入对控制器发出信号，表示数据输入准备就绪，应开始计算。（如果还有其它需要说明的，例如选择运算模式，一并说明。）
2. 控制器将数据输入保存在本地（如果数据输入能在整个运行周期中一直稳定保持，可以让运算器直接从数据输入端口获得输入而不保存。）
3. 控制器将数据输入准备给运算器，并配置好运算器的各种功能模式（如果需要配置）。
4. 控制器让运算器开始执行计算，并将计算结果保存好。
5. 已经得到了计算结果吗？如果还没算完就返回第 3 步继续重复。如果算完了就进入下一步。
6. 将计算结果输出到数据输出端口，并且通过控制输出向外界汇报表示运算结束（如果需要汇报）。

在字节寻址串行运算器的工作周期流程（仅供参考）

图 B 是一个具体的例子，它是一个基于加法器累加的串行乘法器。算法跟之前学过的基于加法器的阵列乘法器一样，但这次它是串行实现的，因此电路规模较阵列乘法器会小得多。

此外，之前讨论过的其它运算器（逐位进位加法器、基于 CSA 的阵列乘法器、恢复余数阵列除法器、不恢复余数阵列除法器、各种进制转换器）也都可以以串行运算器的思路实现。你应该试着去想一下，前面讨论的运算器的串行版本。

另外，像乘法或者多个数求和这样的运算，可以把它们拆开成多个子运算，比如  $A+B+C+D$  拆成  $(A+B)+(C+D)$ 。如果可以这么做，那么你就可以提供多个串行运算器同时进行这些子运算，最后再将它们汇总合并。这能在很大程度上弥补串行运算器性能不足的缺陷。

### 8.3.2 串行运算器的优势与劣势

串行运算器最直观的优势在于它的规模较一般的并行运算器（指非串行运算器）通常要小得多。想象一下 128bits 乘 128bits 的乘法运算器分别用纯并行的设计思路和纯串行的设计思路实现的样子，很显然串行的实现会小得多。

但串行运算器会比并行运算器要慢一些（慢多少取决于实际情况），因为它的每次计算周期都必须准备和保存运算器的输入和输出。

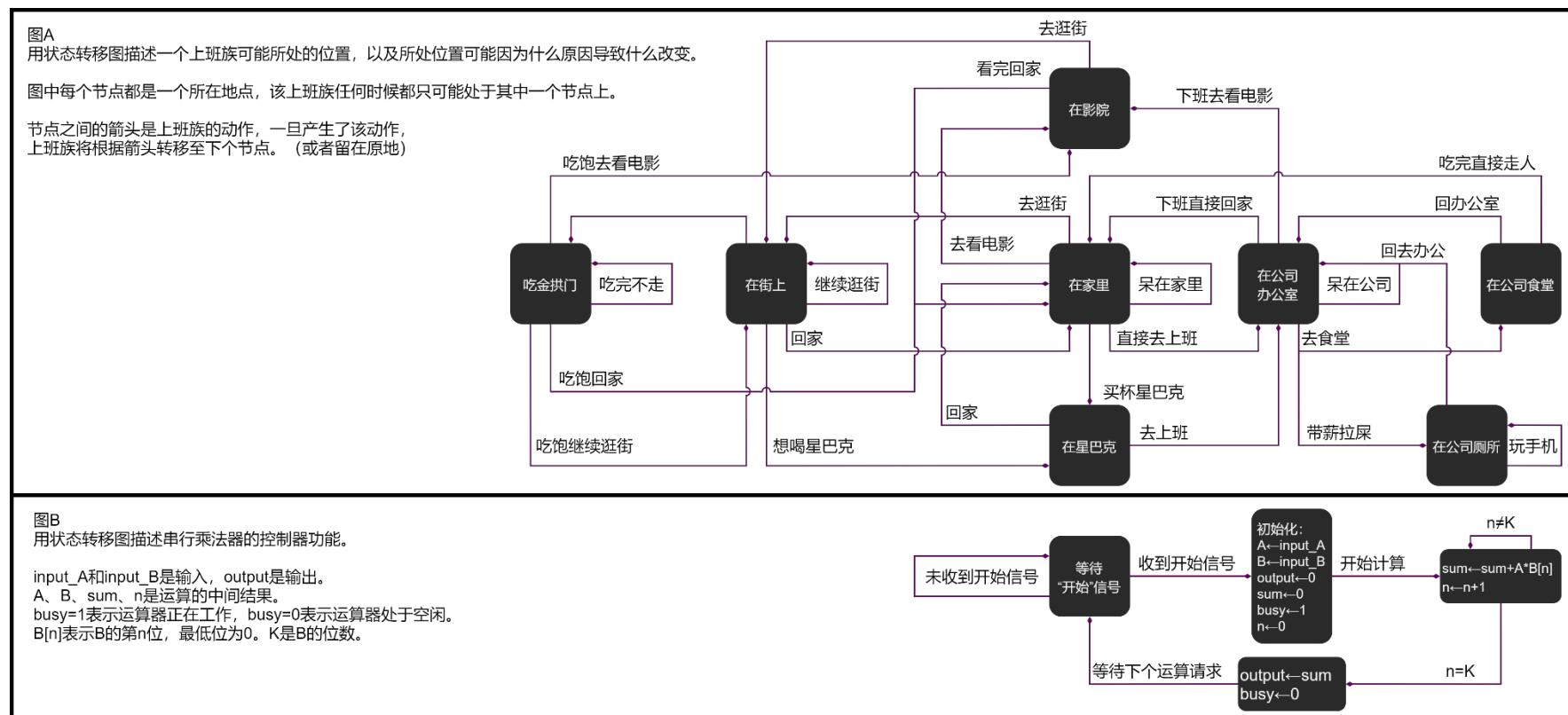
还有一个缺点就是，串行运算器需要涉及控制器来控制各个部件运行，串行运算器的控制时序有时会很难设计，因为你要在多方面的因素之间取舍。（比如频率太高容易出错，频率太低又无法满足性能需求）

总的来说，串行运算器的设计难度要比并行运算器高不少，凡是搭上时序的东西都是如此。

如果你还没领悟到控制工程难度和复杂度的重要性，请记住控制工程难度与复杂度在整个设计流程中是不可忽略的一部分。你必须保证所有工作的难度和复杂度都处在可接受范围内且不会出现太多难以克服的困难，否则最终很可能无功而返。

## 8.4 有限状态机

#### 8.4.1 状态转移图 (STD, State Transition Diagram) 与有限状态机 (FSM, Finite State Machine)



## 状态转移图的例子

图 A 和图 B 都是 STD 的例子。FSM 就是在 STD 上根据规则不断转移当前所在节点的机器。现在具体给出 FSM 的具备要素：

有限状态机的具备要素	解释
现态	当前所处的状态，处在状态转移图中的哪个节点。
条件	又称为事件，就是状态转移图中箭头上的字，条件满足时会触发动作。（条件可能是来自外部的输入或者现态等等）
动作	条件满足后执行的动作，如迁移至下一个节点，或者呆在原地。
次态	条件满足后需要迁往的下一个状态（状态转移图中的节点），次态是较当前状态而言的。

## 有限状态机的四个具备要素

如果使用集合论语言，FSM 可以描述为一个五元组  $M = (Q, \Sigma, \delta, q_0, F)$ ，其中：

$Q = q_0, q_1, q_2 \dots q_n$  是有限状态集。在任一确定时刻 FSM 只能处于一个确定的状态  $q_i$ 。

$\Sigma = \sigma_1, \sigma_2, \sigma_3 \dots \sigma_n$  是有限输入字符集合。在任一确定的时刻，有限状态机只能接收一个确定的输入  $\sigma_i$ 。

$\delta: Q \times \Sigma \rightarrow Q$  是状态转移函数，在某一状态下，给

$q_0 \in Q$  是初始状态， FSM 由此状态开始接收输入。

这个五元组就是 **FSM** 的严格定义。如果读者没上过高中可能会看不懂，但是看不懂也没关系，因为这个五元组定义的理解不做要求。

影响实践。理解有限状态机的四

此外，FSM 主要分为两大类：

第一类：输出只和状态有关而与输入无关，称为 Moore 状态机。

FSM 能很好地描述许多系统的功能。在实际设计中，如果需要实现某个特定功能的控制器一般都会先用画出它的状态转移图，然后再考虑具体实现。实际上在许多 IC 设计软件中都提供了自动将 FSM 转为硬件代码/电路图的功能，只要你在软件中画出了 FSM，它就能自

#### 第五章 管理的逻辑

FSM 的电路实现很简单，只需要一个查找表（LUT）和一个寄存器即可。寄存器用于存储 FSM 的当前状态，LUT 中存储了 FSM 所有可能的状态转移情况，并且根据寄存器的值和输入输出 FSM 的次态，然后再将次态写入寄存器内，一次状态转移就完成了。（当然，FSM 的实现基本上都是时钟驱动的同步电路）

如果其中一人是政府的，另一人可以使用一个政府的、而不仅仅是其优势的、而不仅仅是其优势的

FSM 的最主要应用还作为电路控制单元/控制器的一个抽象描述。使用 FSM 可以清晰地描述一个控制器的行为功能，并提供了一个标准化的电路实现方法（如基于 LUT 的方法），从而大大改善了控制器的设计难度与可调试性。

## 8.5 时序电路的描述

由于时序电路引入了时间这一维度（通过电路状态沟通过去与未来），时序电路分析与描述要比组合逻辑电路复杂得多。有关时序电路的分析可以涉及许多概念与工具（你应该还记得这部分开头的那个超长的术语表），它们在实际项目被用于电路设计的分析、验证与优化。这里暂时只讨论其中少部分的基本概念，主要是让读者至少有办法像逻辑表达式那样抽象地描述一个时序电路。

### 8.5.1 时序电路的状态转移图、状态转移表、时序图

时序电路的状态转移图、状态转移表、时序图都是用来描述时序电路的工具，它们都能完整描述一个时序电路，但是画出来的样子不同。作为例子，JK触发器及其对应的状态转移图、状态转移表、时序图如上。

### 8.5.2 用方程描述时序电路

时序电路也可以用方程来描述，涉及到的几个方程分别被叫做激励方程（别名驱动方程）、状态方程和输出方程。这里我们再次以JK触发器为例，这是JK触发器的激励方程：

### 8.5.3 时序电路的分析与优化

## 9 数据交换 (DATA TRANSFER)

### 9.1 前言



这部分我们来了解数据交换的基础概念，这里所谓的数据交换意思是指几个电路之间的输入输出交换。上图就是一个例子，图中的几个部件通过中间的“总线 BUS”电路相互交换数据。在本书中，我把这种为被接入部件提供相互转发数据功能的结构称为交换节点。

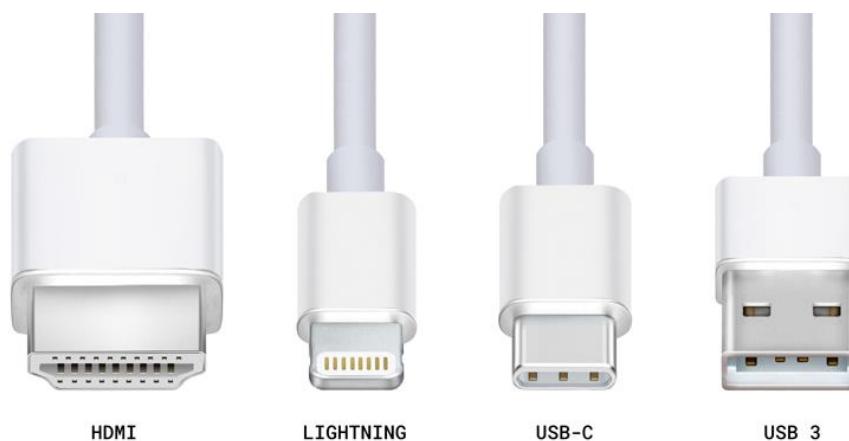
数据交换也是一个重要主题。例如，如果数据交换的效率低于各部件生产和消费数据的效率，数据交换就会成为一个性能瓶颈。

### 9.2 接口标准 (INTERFACE STANDARD) 与接口协议 (INTERFACE PROTOCOL)



设备面板上的各种接口

所谓接口（硬件接口）就是指两个设备/模块/组件间的连接方式，如上图就展示了一个电子设备上的各种接口。除了接口还有接头，接头就是用来插进接口的那个头，例如下图：



生活中常简的一些接头。USB-C 也常被称为 Type-C，平常所说的 USB 口更多是指图中的 USB 3 口

接口标准与接口协议则规范了一种接口的物理形状、性能参数、数据通讯方式等一系列需要确定的标准。

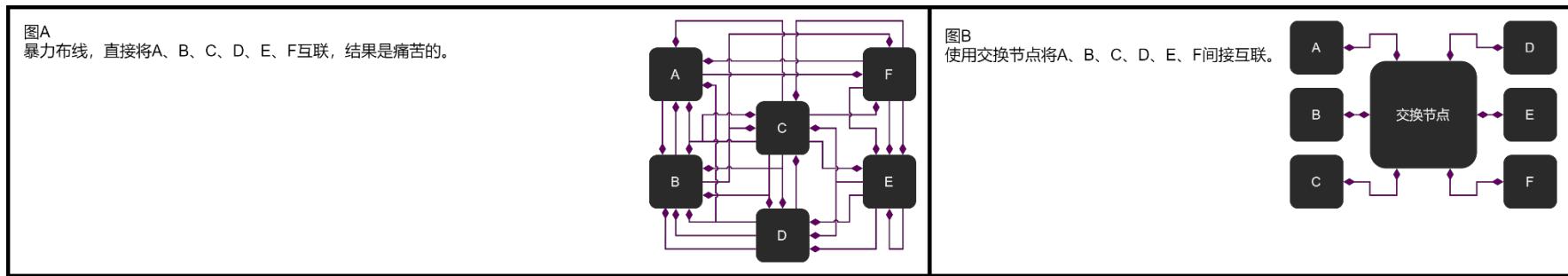
接口标准与接口协议这种东西存在的意义在于提供了一个标准化的设备互连方式，例如，只要我们的设备提供了一个 USB 口，那么其它拥有 USB 头的设备就可以接入并根据该 USB 接口的通讯协议与我们的设备互联。

另一方面，这也意味着，如果我们的产品要与第三方的产品沟通，就必须在我们与对方间协定一个双方都将支持的接口标准和协议。

一般来说，应该采用被业内广泛采用的标准和协议，除非市场上已有的标准和协议无法满足需求。生活中的例子就是，电脑外设产品通常都会采用上图那样的 USB 头作为设备接口，因为绝大部分电脑都会带有多个 USB 口。（另一方面，电脑制造商也会因此提供 USB 口）

读者可以通过搜索引擎简单了解一些常见的接口标准与协议，并且做好与它们打交道的准备。

### 9.3 交换节点



使用交换节点转发数据

如上图，交换节点就是一个这样的，用于处理多个组件间的数据交换的一个组件。相较于图 A 中直接连线的方式，图 B 中交换节点的加入带来了许多好处，比如：

1. 整个电路的布线布局更简洁了，可读性大大增加。
2. 更利于管理电路、排除故障、测试和 debug。
3. 提供了一个规范的拓展方式：一个交换节点会提供若干个接口，如果某个组件要加入这个数据交换节点，只需要让该组件本身能够支持该交换节点的接入标准即可，而不需要考虑其它的东西。（就像电脑主机通常都会提供的 USB 接口那样，只要你的电子设备提供了一个相同标准的 USB 头，就能插入它并通过该接口与电脑互联）
4. 更利于拓展、加入或去除设备：如果想要给交换节点接入新组件，只要将符合标准的组件接口与节点接口相连即可；如果你希望去掉某个已加入交换节点的设备，只需要在相应接口处将组件接口与节点接口脱离即可。

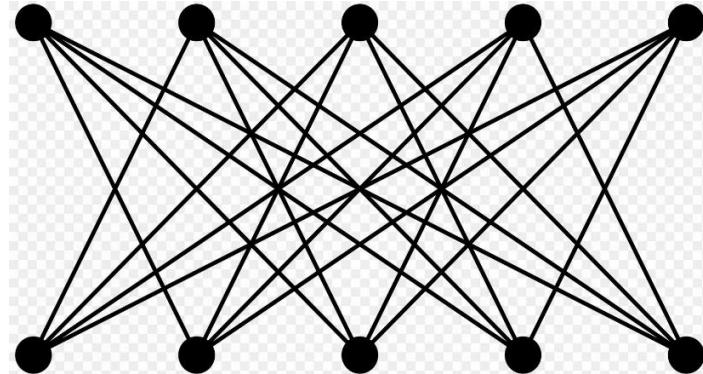
节点本身也是组件，所以你也可以让多个交换节点之间相互连接，以多个小的交换节点组成一个大的交换网络。

#### 9.3.1 交换节点的使用场景

当然，并不是什么时候都适合引入“交换节点”。如果部件较少，比如只有两三个，那可能还是直接布线更好。交换节点通常被使用在那些部件较多、且数据交换的形式更适合交换节点的地方。（最直接的判断方法是：比较基于交换节点和不基于交换节点的设计哪个更好）

总之，任何技术都应该考虑它的使用背景。另一方面，交换节点本身的设计也有可以有很多讨论的设计细节。在未来，我们会专门讨论一些量化数据交换过程的模型，并且利用它们来科学地指导我们选择和调整设计。但在这里，泛泛而谈即可。

#### 9.3.2 网络 (Network) 与图论 (Graph Theory)，一些术语



图的例子

如果把各个部件分别看作“节点 (Node)”，然后再把各部件之间的通路看作“边 (Edge)”（有向边或无向边），就可以把整个电路看作是一个网络，或者称它为图 (Graph)，上图就是一个例子。

用准确的数学概念描述现象是量化分析的基础，除了让我们拥有一个相当严格的描述语言外，更重要的是能够由此构建工程与数学理论的桥梁，数学的任何结论相关结论都可以成为我们的强力武器。（当然，可能涉及的数学基本不会超出应用数学的范畴）

但是我们暂时不深入讨论这些，在这里，我只是想解释一些常用基础术语：

术语	解释
图	也叫网络，由点和边构成的抽象结构。其中点被边连接，一条边连接两个点。
点	在图中被边连接的点，也叫节点。
边	在图中连接两条点的边，可以分有向边和无向边。有向边是有方向的，无向边是无方向的。
有向图	有向图中的边是有向边。
无向图	无向图中的边是无向边。
点属性	点的属性。应用中我们会让节点带一或多个属性，比如让他带有一个属性值 c，表示这个节点所能容纳的数据包的数量。
边属性	边的属性。比如让边带有属性 t，表示数据包通过该边所需的时间，即时延。类似的还有速率、带宽、吞吐量、利用率等。
流	在网络上流动的东西。比如计算机网络上流动的信息数据。
流量	点、边或网络进出数据的量。
堵塞	网络转发数据的能力低于数据注入网络的速度的情况，类似堵车。
拓扑	计算机中的拓扑一般指图中各个点的分布情况和边的连接情况。（数学家对“拓扑”一词有更严格、广泛的数学定义）

上面的解释是不严谨的，没有用数学术语严格定义。关于图，一种严格的定义是（不需要掌握）：图 G 是一个有序二元组( $V, E$ )，其中  $V$  的元素称为点。 $E$  的元素称为边。 $E$  的元素都是二元组( $x, y$ )，满足  $x, y \in V$ 。

### 9.3.3 交换节点的功能与结构

现在我们将注意力集中在交换节点的内部。交换节点本质是就是一个拥有  $n$  个接口，并在其接口间转发数据的电路。不同的交换节点设计拥有不同的性能和特性，我们也可以将多个交换节点的组合看作是一个更大的交换节点。上图就展示了一些交换节点的内部方案，它们各有不同的优劣势。

//router switch hub

## 9.4 交换协议

所谓协议，就是约定成俗的事实，加入协议者都必须遵循协议的规则。我们可以把协议分为主动和被动两种：

类型	解释
主动协议	加入协议的设备主动遵循的协议。例如：规定设备在交换数据前，必须发出一个申请讯息，并获得许可。
被动协议	加入协议的设备被动遵循的协议。例如：规定设备发送数据的频率不能超过 $P$ ，否则禁止该设备一小时内再次发送数据。

接下来，我们会讨论一些交换协议的例子，并讨论这些协议的制定动机。

### 9.4.1 仲裁协议

如上图，A、B、C、E、F 这六个设备通过中间的总线 BUS 交换数据。那么问题来了：假设总线 BUS 同一时间只能处理一个请求，但这时候有两个或以上的设备同时向总线 BUS 发出请求，该怎么办？

一种办法就是采用所谓的仲裁协议，所谓仲裁协议，就是用来仲裁当下某资源应该被谁使用的协议。仲裁协议也有不同的仲裁方式，这里简单地介绍其中一些：

I.

## 10 第二部分导读

---

### 10.1 第二部分导读

第二部分的主题是计算机组成原理。我们会先从大体上直观理解一遍计算机和 CPU 的工作过程以及原理，然后再具体讨论到它们的实现。这部分我们会讨论 CPU 的实现，目标是理解如何实现一个简单的 CPU。

在后面的部分中我们还会讨论有关处理器设计的更多优化技术和设计方法，并且以一个更高的层次看待计算机体系结构。

然后再次感谢大家的建议与勘误！

### 10.2 计算机体系结构中的重要思想

这部分请参考 [zhuanlan.zhihu.com/p/164550580](https://zhuanlan.zhihu.com/p/164550580) 中的“计算机体系结构的八个重要思想”。

# II 认识计算机与 CPU 与 RISC-V

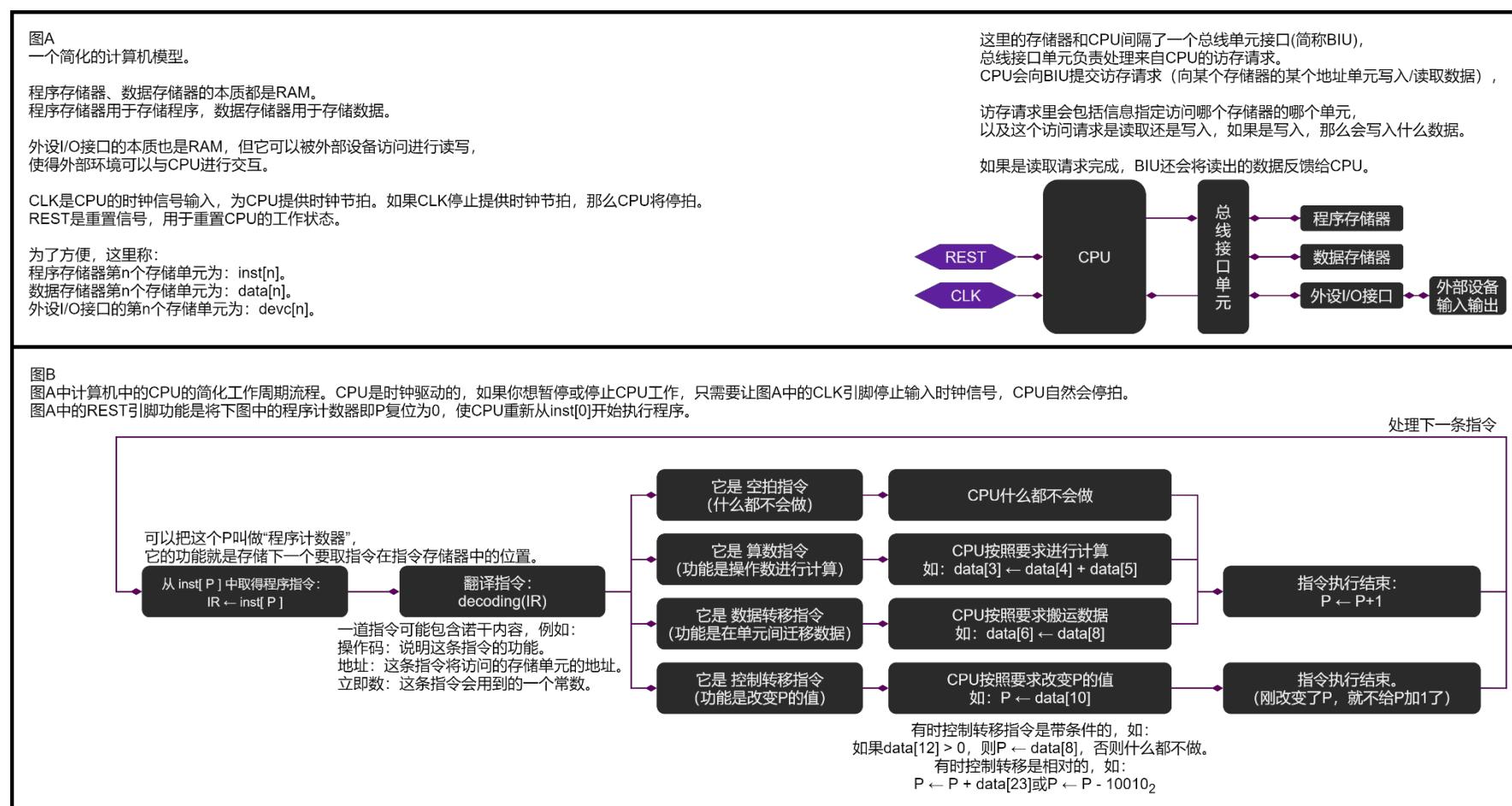
## II.1 前言

这部分我们从最简单的模型来开始认识计算机与 CPU 的工作过程及原理，并且介绍当下比较流行的 RISC-V 指令集。

这里推荐一个很不错的计算机工作原理科普系列视频：[www.bilibili.com/video/BV1EW411u7th](http://www.bilibili.com/video/BV1EW411u7th)

## II.2 计算机的工作流程

### II.2.1 简化的计算机模型与工作流程



简化的计算机结构与 CPU 工作周期

图 A 和图 B 展示了一个简化简化再简化的计算机和 CPU 的模型与工作流程。下表展示了这个计算机从开机到关机的过程：

1. 开机。时钟开始运作向 CPU 提供时钟信号，同时通过 REST 引脚将 CPU 重置至初始化，然后从 inst[0] 开始获取并执行指令。
2. 从 inst[0] 开始不停执行程序存储器中的程序指令，且根据指令完成各种操作。
3. 关机。时钟停止运作。

计算机从开机到关机的过程（被严重简化）

在 CPU 的执行过程中，CPU 执行的程序指令扮演着无比关键的角色。CPU 从程序存储器中获得并且能直接识别的指令叫机器指令，它是一个 n bits 的二进制码。一个计算机程序就是由多条机器指令构成的。

一个 CPU 能识别的所有指令在一起就组成了这个 CPU 的指令集 (ISA, Instruction Set Architecture)。指令集规定了 CPU 应该将作为代码的二进制码识别为什么指令，以及指令的功能。

不同 CPU 可能采用不同的指令集，如 intel 的 i9 系列处理器采用了 intel 自家的 x64 指令集，而高通的骁龙 820 处理器采用了 ARM 的 ARM 指令集。

在本书中，我将采用目前较为流行的 RISC-V 指令集来讨论 CPU 工作原理。RISC-V 本身包涵多套指令集，在实际设计中按照需要选取其中需要实现的几个指令集来设计处理器。此外，RISC-V 也提供了大量的客制化空间，可以按照规范为其添加一些自定义的指令。

这里列举几个 RISC-V 的基础指令集和拓展指令集：

指令集名	说明
RV32I	32 位的基础整数指令集，内含各种 32 位的基础指令和整数运算指令。可用于 32 位处理器。
RV64I	64 位的基础整数指令集，拓展自 RV32I。可用于 64 位处理器。
M	整数乘除拓展指令集，拓展了整数乘法和除法指令，不能独立于基础指令集。
C	压缩指令拓展指令集，添加了一些 16 位的指令，大大减少了程序体积，但不能直接独立于基础指令集。
B	位操作拓展指令集，拓展了一些位操作。
L	十进制浮点拓展指令集，拓展支持了 IEEE 754-2008 标准规定的十进制浮点算术运算。

RISC-V 的部分指令集，更多指令集可以在 [RISC-V 指令集手册 \(The RISC-V Instruction Set Manual Volume I: Unprivileged ISA\)](#) 或 [RISC-V 手册](#) 中了解。

可以看出 RISC-V 分离出了不同功能的指令集，这种设计大大增加了指令集的灵活性，以满足多样化的实际应用情景。例如 RISC-V 将整数乘除指令分割到了一个单独的拓展指令集 M 中，这样的话一些小型的芯片方案在选择 RISC-V 作为指令集时就可以选择不支持 M 指令集以减少各方面的开销。（当然，分割指令集的这种做法的缺点是容易导致碎片化，就是大家用的不够统一）

## 11.3 RV32I 指令集

这节我们来了解 RISC-V 中的 RV32I 指令集，这是 RISC-V 最基础也是最重要的指令集。透过 RV32I 的学习，我还会介绍一些有关指令集的基础概念与术语，而且今后我们讨论的处理器设计也会基于这个指令集。

这是一些有关 RISC-V 的资料（请读者下载一份“RISC-V 手册”作为参考）：

RISC-V 手册（中文）：[riscvbook.com/chinese/](http://riscvbook.com/chinese/)

RISC-V 最新标准文档（英文）：[riscv.org/technical/specifications/](http://riscv.org/technical/specifications/)

RISC-V 官网：[riscv.org/](http://riscv.org/)

### 11.3.1 RV32I 的用户执行环境

RV32I 假定采用了 RV32I 的 CPU 必须在 CPU 内部提供以下东西以支持 RV32I 指令集的功能：

1. 由 32 个 32bits 的寄存器单元组成的通用寄存器组，这 32 个寄存器单元分别叫做  $x_0$ 、 $x_1$ 、 $x_2$ 、 $x_3 \dots x_{30}$ 、 $x_{31}$ 。
2. 寄存器  $x_0$  的值永远为 0。（无论如何从  $x_0$  中读到的值都是 0。可以把没用的数据写入  $x_0$ ，把  $x_0$  当作垃圾桶来丢弃数据）
3. 一个程序计数器，叫做  $pc$ 。在 32 位处理器中它也应是 32bits 的。

这个通用寄存器组（简称 GPR 或 GR 或 GPRs 或 GRs）和程序计数器都是用户可见的，也就是说在用户程序中可以通过指令直接利用这些资源。比如在寄存器和存储器之间转移数据，或者用寄存器中的数据进行计算，或者改变  $pc$  的值以实现跳转等等...

现代的主流 CPU 都会在内部内置寄存器组用于暂存数据，因为这能大大减少处理器访问存储器的次数。（访问存储器在实际情况中是很慢的，相对于访问寄存器组而言）

### 11.3.2 RV32I 的指令格式

RV32I 的指令有 6 种指令编码格式，并且它们都是统一 32bits 长的，具体如下：

格式						格式名
func7	rs2	rs1	func3	rd	opcode	R-type
imm[11:0]		rs1	func3	rd	opcode	I-type
imm[11:5]	rs2	rs1	func3	imm[4:0]	opcode	S-type
imm[12]	imm[10:5]	rs2	func3	imm[4:1]	imm[11]	B-type
imm[31:12]				rd	opcode	U-type
imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode	J-type

第 31 ~ 25 位	第 24 ~ 20 位	第 19 ~ 15 位	第 15 ~ 12 位	第 11 ~ 7 位	第 6 ~ 0 位	←每段的长度 (从 0 开始数)
第 31 位	第 30 ~ 25 位			第 11 ~ 8 位	第 7 位	

RV32I 的 6 种指令格式

其中：

1.  $imm[A:B]$  这样的符号的意思是从第  $B$  位 ( $B$  最低为 0) 到第  $A$  位的一段二进制数，如  $imm[7:2]$  就表示从第 2 位到第 7 位的一段数。
2.  $imm[B]$  特指第  $B$  位数，如  $imm[11]$  表示第 11 位数。
3.  $imm$  是立即数，它是存在于指令内的一段指令需要用到的常数，比如用于参与运算或者向寄存器载入常数等等。
4.  $rs1$ 、 $rs2$  是源寄存器， $rd$  是目标寄存器，它们的本质是一段 5bits 的地址，用于在 CPU 通用寄存器组中指定某个寄存器。
5.  $opcode$  是操作码，用来说明这个指令是什么格式的什么指令。 $func$  是功能码，功能也是用来区分指令，但  $func$  不区分指令格式。
6. 如果将  $rs1$  或者  $rs2$  设置为  $x_0$ ，那么将从  $x_0$  中读出“0”。如果将  $rd$  设置位  $x_0$ ，那么向  $rd$  写入的数据会被丢弃。

RISC-V 是经过大量经验沉淀与思考后设计出来的，以下一段是“RISC-V 手册（中文版）”的原话：

分支指令（B 类型）的立即数字段在 S 类型的基础上旋转了 1 位。跳转指令（J 类型）的直接字段在 U 类型的基础上旋转了 12 位。因此，RISC-V 实际上只有四种基本格式，但我们可以保守地认为它有六种格式。

为了帮助程序员，所有位全部是 0 的指令是非法的 RV32I 指令。因此，试图跳转到被清零的内存区域的错误跳转将会立即触发异常，这可以帮助调试。类似地，所有位全部是 1 的指令也是非法指令，它将捕获其他常见的错误，诸如未编程的非易失性内存设备、断开连接的内存总线或者坏掉的内存芯片。

为了给 ISA 扩展留出足够的空间，最基础的 RV32I 指令集只使用了 32 位指令字中的编码空间的不到八分之一。架构师们也仔细挑选了 RV32I 操作码，使拥有共同数据通路的指令的操作码位有尽可能多的位的值是一样的，这简化了控制逻辑。最后，当我们看到，B 和 J 格式的分支和跳转地址必须向左移动 1 位以将地址乘以 2，从而给予分支和跳转指令更大的跳转范围。RISC-V 将立即数中的位从自然排布进行了一些移位轮换，将指令信号的扇出和立即数多路复用的成本降低了近两倍，这也简化了低端实现中的数据通路逻辑。

RISC-V 的每处细节都经过了设计者的仔细推敲。RISC-V 出现的时间较晚，2010 年时才开始立项。得益于此，RISC-V 能够从历史中吸取大量的设计经验与理论并付之实践。

有关 RISC-V 的更多设计思路和动机可以在 RISC-V 手册中了解到（如果你是指令集设计的爱好者，我会强烈推荐你细读一遍 RISC-V 手册，因为其除了介绍 RISC-V 外，还讨论了大量的指令集设计原则与经验）。

### 11.3.3 RV32I 指令集

RV32I 指令集在“RISC-V 手册”中的第 25 面有列出，附页 A 详细说明了 RISC-V 指令集中每条指令的具体定义（不局限于 RV32I）。为了方便读者，这里我把其中 RV32I 的部分单独列了出来：

RV32I 指令集索引 (个人)												
内容包括 RV32I Base Integer Instruction Set (Version 2.1) RISC-V 中文手册 : riscvbook.com/chinese/												
BY RESENS RV32I 寄存器												
31 个 32bits 的通用寄存器，分别是 x1、x2、x3...x30、x31。 1 个值永远为 0000 0000 0000 0000 0000 0000 0000 0000 的 x0 寄存器。(32bits) 1 个 32bits 的程序计数器 pc。 (本表中的说明仅针对 32 位的处理器实现)												
符号说明												
<p>“<math>\leftarrow</math>”是赋值符号，将右边的值赋予给左边的变量，如 <math>a \leftarrow 10</math> 就是将 10 赋予给 a。“&amp;”“<math>\oplus</math>”分别是与位运算、或位运算、异或位运算。</p> <p>“<math>&lt;_n</math>”是自动拓展位数的左移符号，它会自动拓展位数，如 <math>1010 &lt;_4 = 1010\ 0000</math>。</p> <p>“<math>x[rd]</math>”表示用 rd 进行寻址（在通用寄存器组中）所访问的寄存器单元。（可以是 <math>x0 \sim x31</math> 中的一个）如 <math>x[1]</math> 就代表通用寄存器 x1，<math>x[30]</math> 代表通用寄存器 x30。</p> <p>“pc”是程序计数器的意思，它在每个指令执行结束后都会 +4 以指向下一个指令。为什么是 +4 不是 +1？因为 RISC-V 采用的是地址寻址模式，pc 每次 +1 仅会跨过一个 8bits 的单元，所以需要 +4 来跨过 32bits，因为在 RV32I 中一个指令的长度就是 32bits。一些涉及到 pc 的值的运算中，可能会出现两个位数相同的无符号数与补码数的加法，它们应该直接相加。（把两个都当作无符号数直接进行无符号数加法）</p> <p>“sext(snmu)”是符号拓展，本表中的意思是对 M_bits 的补码数 snmu 进行符号拓展至 32 bits。（补码的符号拓展就是将新位数下高于符号位的位全设置为符号位的值，不理解可以百度“符号拓展”）</p> <p>“uext(unmu)”是无符号拓展，本表中的意思是对于 M_bits 的无符号数 unmu 进行符号拓展至 32 bits。（无符号数的无符号拓展就是将新位数下高于符号位的位全设置为 0，不理解可以百度“无符号拓展”）</p> <p>“if(cond) : seq”是条件语句，意思是如果语句中的 cond 是对的，则执行 seq，否则什么都不做。</p> <p>“(cond)? seq1 : seq2”是条件运算符号，意思是如果 cond 是对的，则执行 seq1，否则执行 seq2。</p> <p>“<math>\geq, \leq, &lt;, &gt;</math>”是补码数的大小比较符号，比较时会考虑补码数的正负。“<math>\geq_u, \leq_u</math>”是无符号数的大小比较符号，显然无符号数不会为负。</p> <p>“M[addr]”表示用 addr 进行寻址（在存储器中）所访问的存储器单元。</p> <p>“G[A:B]”是 G 从第 B 位到第 A 位这一段数。（最低位是第 0 位，也就是从 0 开始数）“G[B]”特指 G 的第 B 位。“G[A:B]C[D:E]”表示将 G[A:B]、G[C]、G[D:E]这几段数从右到左拼接起来，其它同理。</p> <p>“<math>&lt;&lt;_n &gt;&gt;_n</math>”分别是逻辑左移和逻辑右移。逻辑位移就是普通的位移，左移时低位补 0，右移时高位补 1。如果移位超出范围，超出范围的部分将丢失。</p> <p>“<math>&gt;&gt;</math>”是算数右移。算术左移和逻辑左移一样。算数右移要考虑到符号位，右移时高位根据符号位进行补充，最高位为 1 则补 1，为 0 则补 0。</p>												
指令名	中文名	指令参数与编码	功能说明	立即数指令	无条件控制转移 (跳转) 指令	有条件控制转移 (跳转) 指令						
LUI	高位立即数加载	rd, immediate U-type 格式 <table border="1"><tr><td>immediate[31:12]</td><td>rd</td><td>0110111</td></tr></table>	immediate[31:12]	rd	0110111	$x[rd] \leftarrow \text{sext}(\text{immediate} <<_{\text{ext}} 12);$ (其中 immediate 是 20bits 的补码数。)	把立即数 immediate 左移 12 位再进行符号拓展，结果写入 x[rd]。					
immediate[31:12]	rd	0110111										
AUIPC	PC 加立即数	rd, immediate U-type 格式 <table border="1"><tr><td>immediate[31:12]</td><td>rd</td><td>0010111</td></tr></table>	immediate[31:12]	rd	0010111	$x[rd] \leftarrow pc + \text{sext}(\text{immediate} <<_{\text{ext}} 12);$ (其中 pc 是 32bits 的无符号数，immediate 是 20bits 的补码数)	把立即数 immediate 左移 12 位再进行符号拓展，与 pc 的值相加后将结果写入 x[rd]。					
immediate[31:12]	rd	0010111										
JAL	跳转并链接	rd, offset J-type 格式 <table border="1"><tr><td>offset[20:10:1 11 19:12]</td><td>rd</td><td>1101111</td></tr></table>	offset[20:10:1 11 19:12]	rd	1101111	$x[rd] \leftarrow pc + 4;$ $pc \leftarrow pc + \text{sext}(\text{offset} <<_{\text{ext}} 1);$ (其中 pc 是 32bits 的无符号数，offset 是 20bits 的补码数)	当前的下个指令的地址存入 rd，然后给 pc 加上符号拓展后的 2*offset 进行跳转。					
offset[20:10:1 11 19:12]	rd	1101111										
JALR	跳转并寄存器链接	rd, rs1, offset I-type 格式 <table border="1"><tr><td>offset[11:0]</td><td>rs1</td><td>010</td><td>rd</td><td>1100111</td></tr></table>	offset[11:0]	rs1	010	rd	1100111	$x[rd] \leftarrow pc + 4;$ $pc \leftarrow (x[rs1] + \text{sext}(\text{offset} <<_{\text{ext}} 1)) \& 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110;$ (其中 pc 是 32bits 的无符号数，x[rs1] 是 32bits 的补码数，offset 是 12bits 的补码数)	当前的下个指令的地址存入 rd，然后计算 x[rs1] + sext(32(offset << ext 1))，将计算结果的最低有效位设置为 0 后写入 pc 进行跳转。		有条件的控制转移 (跳转) 指令	
offset[11:0]	rs1	010	rd	1100111								
BEQ	等于时分支	rs1, rs2, offset B-type 格式 <table border="1"><tr><td>offset[12:10:5]</td><td>rs2</td><td>rs1</td><td>000</td><td>offset[4:1 11]</td><td>1100011</td></tr></table>	offset[12:10:5]	rs2	rs1	000	offset[4:1 11]	1100011	$\text{if}(x[rs1] = x[rs2]): pc \leftarrow pc + \text{sext}(\text{offset} <<_{\text{ext}} 1);$ (其中 pc 是 32bits 的无符号数，x[rs1]、x[rs2] 是 32bits 的补码数，offset 是 12bits 的补码数)	如果 x[rsA] = x[rsB] 成立，则给 pc 加上符号拓展后的 2*offset 进行跳转，否则什么都不做。		
offset[12:10:5]	rs2	rs1	000	offset[4:1 11]	1100011							
BNE	不相等时分支	rs1, rs2, offset B-type 格式 <table border="1"><tr><td>offset[12:10:5]</td><td>rs2</td><td>rs1</td><td>001</td><td>offset[4:1 11]</td><td>1100011</td></tr></table>	offset[12:10:5]	rs2	rs1	001	offset[4:1 11]	1100011	$\text{if}(x[rs1] \neq x[rs2]): pc \leftarrow pc + \text{sext}(\text{offset} <<_{\text{ext}} 1);$ (其中 pc 是 32bits 的无符号数，x[rs1]、x[rs2] 是 32bits 的补码数，offset 是 12bits 的补码数)	如果 x[rsA] ≠ x[rsB] 成立，则给 pc 加上符号拓展后的 2*offset 进行跳转，否则什么都不做。		
offset[12:10:5]	rs2	rs1	001	offset[4:1 11]	1100011							
BLT	小于时分支	rs1, rs2, offset B-type 格式 <table border="1"><tr><td>offset[12:10:5]</td><td>rs2</td><td>rs1</td><td>100</td><td>offset[4:1 11]</td><td>1100011</td></tr></table>	offset[12:10:5]	rs2	rs1	100	offset[4:1 11]	1100011	$\text{if}(x[rs1] <_s x[rs2]): pc \leftarrow pc + \text{sext}(\text{offset} <<_{\text{ext}} 1);$ (其中 pc 是 32bits 的无符号数，x[rs1]、x[rs2] 是 32bits 的补码数，offset 是 12bits 的补码数)	如果 x[rsA] < s x[rsB] 成立，则给 pc 加上符号拓展后的 2*offset 进行跳转，否则什么都不做。		
offset[12:10:5]	rs2	rs1	100	offset[4:1 11]	1100011							
BGE	大于时分支	rs1, rs2, offset B-type 格式 <table border="1"><tr><td>offset[12:10:5]</td><td>rs2</td><td>rs1</td><td>101</td><td>offset[4:1 11]</td><td>1100011</td></tr></table>	offset[12:10:5]	rs2	rs1	101	offset[4:1 11]	1100011	$\text{if}(x[rs1] \geq_s x[rs2]): pc \leftarrow pc + \text{sext}(\text{offset} <<_{\text{ext}} 1);$ (其中 pc 是 32bits 的无符号数，x[rs1]、x[rs2] 是 32bits 的补码数，offset 是 12bits 的补码数)	如果 x[rsA] ≥ s x[rsB] 成立，则给 pc 加上符号拓展后的 2*offset 进行跳转，否则什么都不做。		
offset[12:10:5]	rs2	rs1	101	offset[4:1 11]	1100011							
BLTU	小于时分支(无符号)	rs1, rs2, offset B-type 格式 <table border="1"><tr><td>offset[12:10:5]</td><td>rs2</td><td>rs1</td><td>110</td><td>offset[4:1 11]</td><td>1100011</td></tr></table>	offset[12:10:5]	rs2	rs1	110	offset[4:1 11]	1100011	$\text{if}(x[rs1] <_u x[rs2]): pc \leftarrow pc + \text{sext}(\text{offset} <<_{\text{ext}} 1);$ (其中 pc 是 32bits 的无符号数，x[rs1]、x[rs2] 是 32bits 的无符号数，offset 是 12bits 的补码数)	如果 x[rsA] < u x[rsB] 成立，则给 pc 加上符号拓展后的 2*offset 进行跳转，否则什么都不做。		
offset[12:10:5]	rs2	rs1	110	offset[4:1 11]	1100011							
BGEU	大于时分支(无符号)	rs1, rs2, offset B-type 格式 <table border="1"><tr><td>offset[12:10:5]</td><td>rs2</td><td>rs1</td><td>111</td><td>offset[4:1 11]</td><td>1100011</td></tr></table>	offset[12:10:5]	rs2	rs1	111	offset[4:1 11]	1100011	$\text{if}(x[rs1] \geq_u x[rs2]): pc \leftarrow pc + \text{sext}(\text{offset} <<_{\text{ext}} 1);$ (其中 pc 是 32bits 的无符号数，x[rs1]、x[rs2] 是 32bits 的无符号数，offset 是 12bits 的补码数)	如果 x[rsA] ≥ u x[rsB] 成立，则给 pc 加上符号拓展后的 2*offset 进行跳转，否则什么都不做。		
offset[12:10:5]	rs2	rs1	111	offset[4:1 11]	1100011							
从存储器中读取数据到寄存器指令												
LB	字节加载	rd, rs1, offset I-type 格式 <table border="1"><tr><td>offset[11:0]</td><td>rs1</td><td>000</td><td>rd</td><td>0000011</td></tr></table>	offset[11:0]	rs1	000	rd	0000011	$x[rd] \leftarrow \text{sext}(M[x[rs1] + \text{sext}(\text{offset})][7:0]);$ (其中 x[rs1] 是 32bits 的补码数，offset 是 12bits 的补码数)	在存储器中从地址 x[rs1] + sext(offset) 读取 1 个字节，经符号扩展后写入 x[rd]。			
offset[11:0]	rs1	000	rd	0000011								
LH	半字加载	rd, rs1, offset I-type 格式 <table border="1"><tr><td>offset[11:0]</td><td>rs1</td><td>001</td><td>rd</td><td>0000011</td></tr></table>	offset[11:0]	rs1	001	rd	0000011	$x[rd] \leftarrow \text{sext}(M[x[rs1] + \text{sext}(\text{offset})][15:0]);$ (其中 x[rs1] 是 32bits 的补码数，offset 是 12bits 的补码数)	在存储器中从地址 x[rs1] + sext(offset) 读取 2 个字节，经符号扩展后写入 x[rd]。			
offset[11:0]	rs1	001	rd	0000011								
LW	字加载	rd, rs1, offset I-type 格式 <table border="1"><tr><td>offset[11:0]</td><td>rs1</td><td>010</td><td>rd</td><td>0000011</td></tr></table>	offset[11:0]	rs1	010	rd	0000011	$x[rd] \leftarrow \text{sext}(M[x[rs1] + \text{sext}(\text{offset})][31:0]);$ (其中 x[rs1] 是 32bits 的补码数，offset 是 12bits 的补码数)	在存储器中从地址 x[rs1] + sext(offset) 读取 4 个字节，经符号扩展后写入 x[rd]。			
offset[11:0]	rs1	010	rd	0000011								
LBU	字节加载(无符号)	rd, rs1, offset I-type 格式 <table border="1"><tr><td>offset[11:0]</td><td>rs1</td><td>100</td><td>rd</td><td>0000011</td></tr></table>	offset[11:0]	rs1	100	rd	0000011	$x[rd] \leftarrow \text{uext}(M[x[rs1] + \text{sext}(\text{offset})][15:0]);$ (其中 x[rs1] 是 32bits 的补码数，offset 是 12bits 的补码数)	在存储器中从地址 x[rs1] + sext(offset) 读取 1 个字节，经无符号位扩展后写入 x[rd]。			
offset[11:0]	rs1	100	rd	0000011								
LHU	半字加载(无符号)	rd, rs1, offset I-type 格式 <table border="1"><tr><td>offset[11:0]</td><td>rs1</td><td>101</td><td>rd</td><td>0000011</td></tr></table>	offset[11:0]	rs1	101	rd	0000011	$x[rd] \leftarrow \text{uext}(M[x[rs1] + \text{sext}(\text{offset})][7:0]);$ (其中 x[rs1] 是 32bits 的补码数，offset 是 12bits 的补码数)	在存储器中从地址 x[rs1] + sext(offset) 读取 2 个字节，经无符号位扩展后写入 x[rd]。			
offset[11:0]	rs1	101	rd	0000011								
向存储器写入来自寄存器的数据指令												
SB	存字节	rs1, rs2, offset S-type 格式 <table border="1"><tr><td>offset[11:5]</td><td>rs2</td><td>rs1</td><td>000</td><td>offset[4:0]</td><td>0100011</td></tr></table>	offset[11:5]	rs2	rs1	000	offset[4:0]	0100011	$M[x[rs1] + \text{sext}(\text{offset})][7:0] \leftarrow x[rs2][7:0];$ (其中 x[rs1] 是 32bits 的补码数，offset 是 12bits 的补码数)	将 x[rs2] 的低位字节存入内存地址 x[rs1]+sext(offset)。		
offset[11:5]	rs2	rs1	000	offset[4:0]	0100011							
SH	存半字	rs1, rs2, offset S-type 格式 <table border="1"><tr><td>offset[11:5]</td><td>rs2</td><td>rs1</td><td>001</td><td>offset[4:0]</td><td>0100011</td></tr></table>	offset[11:5]	rs2	rs1	001	offset[4:0]	0100011	$M[x[rs1] + \text{sext}(\text{offset})][15:0] \leftarrow x[rs2][15:0];$ (其中 x[rs1] 是 32bits 的补码数，offset 是 12bits 的补码数)	将 x[rs2] 的低 2 个字节存入内存地址 x[rs1]+sext(offset)。		
offset[11:5]	rs2	rs1	001	offset[4:0]	0100011							
SW	存字	rs1, rs2, offset S-type 格式 <table border="1"><tr><td>offset[11:5]</td><td>rs2</td><td>rs1</td><td>010</td><td>offset[4:0]</td><td>0100011</td></tr></table>	offset[11:5]	rs2	rs1	010	offset[4:0]	0100011	$M[x[rs1] + \text{sext}(\text{offset})][31:0] \leftarrow x[rs2][31:0];$ (其中 x[rs1] 是 32bits 的补码数，offset 是 12bits 的补码数)	将 x[rs2] 存入内存地址 x[rs1]+sext(offset)。		
offset[11:5]	rs2	rs1	010	offset[4:0]	0100011							
带立即数的运算指令												
ADDI	加(立即数)	rd, rs1, immediate I-type 格式 <table border="1"><tr><td>immediate[11:0]</td><td>rs1</td><td>000</td><td>rd</td><td>0010011</td></tr></table>	immediate[11:0]	rs1	000	rd	0010011	$x[rd] \leftarrow x[rs1] + \text{sext}(\text{immediate});$ (其中 x[rs1] 是 32bits 的补码数，immediate 是 12bits 的补码数)	把符号扩展的立即数 immediate 加到 x[rs1] 上，结果写入 x[rd]。忽略算术溢出。			
immediate[11:0]	rs1	000	rd	0010011								
SLTI	小于则设置位(立即数)	rd, rs1, immediate I-type 格式 <table border="1"><tr><td>immediate[11:0]</td><td>rs1</td><td>010</td><td>rd</td><td>0010011</td></tr></table>	immediate[11:0]	rs1	010	rd	0010011	$(x[rs1] <_s \text{sext}(\text{immediate}))? rd \leftarrow 1 : rd \leftarrow 0;$ (其中 x[rs1] 是 32bits 的补码数，immediate 是 12bits 的补码数)	比较 x[rs1] 和有符号扩展的 immediate，如果 x[rs1] 更小，向 x[rd] 写入 1，否则写入 0。			
immediate[11:0]	rs1	010	rd	0010011								
SLTIU	小于立即数则置位(无符号)	rd, rs1, immediate I-type 格式 <table border="1"><tr><td>immediate[11:0]</td><td>rs1</td><td>011</td><td>rd</td><td>0010011</td></tr></table>	immediate[11:0]	rs1	011	rd	0010011	$(x[rs1] <_u \text{sext}(\text{immediate}))? rd \leftarrow 1 : rd \leftarrow 0;$ (其中 x[rs1] 是 32bits 的补码数，immediate 是 12bits 的补码数)	比较 x[rs1] 和有符号扩展的 immediate。（比较时均视为无符号数进行无符号数比较）如果 x[rs1] 更小，向 x[rd] 写入 1，否则写入 0。			
immediate[11:0]	rs1	011	rd	0010011								
XORI	异或(立即数)	rd, rs1, immediate I-type 格式 <table border="1"><tr><td>immediate[11:0]</td><td>rs1</td><td>100</td><td>rd</td><td>0010011</td></tr></table>	immediate[11:0]	rs1	100	rd	0010011	$x[rd] \leftarrow x[rs1] \oplus \text{sext}(\text{immediate});$ x[rs1] 和有符号扩展的 immediate 按位异或，结果写入 x[rd]。				
immediate[11:0]	rs1	100	rd	0010011								

ORI	或(立即数)	<b>rd, rs1, immediate</b> I-type 格式 <table border="1"><tr><td>immediate[11:0]</td><td>rs1</td><td>110</td><td>rd</td><td>0010011</td></tr></table>	immediate[11:0]	rs1	110	rd	0010011	$x[rd] \leftarrow x[rs1] \mid sext(immediate);$  $x[rs1]$ 和有符号扩展的 immediate 按位或，结果写入 $x[rd]$ 。		
immediate[11:0]	rs1	110	rd	0010011						
ANDI	与(立即数)	<b>rs1, rd, immediate</b> I-type 格式 <table border="1"><tr><td>immediate[11:0]</td><td>rs1</td><td>111</td><td>rd</td><td>0010011</td></tr></table>	immediate[11:0]	rs1	111	rd	0010011	$x[rd] \leftarrow x[rs1] \& sext(immediate);$  $x[rs1]$ 和有符号扩展的 immediate 按位与，结果写入 $x[rd]$ 。		
immediate[11:0]	rs1	111	rd	0010011						
SLLI	逻辑左移(立即数)	<b>rs1, rd, shamt</b> I-type 格式 <table border="1"><tr><td>0000000</td><td>shamt</td><td>rs1</td><td>001</td><td>rd</td><td>0010011</td></tr></table>	0000000	shamt	rs1	001	rd	0010011	$x[rd] \leftarrow x[rs1] <<_l shamt;$ (其中 shamt 是 5bits 的无符号数) 把 $x[rs1]$ 左移 shamt 位，空出的位置填入 0，结果写入 $x[rd]$ 。(逻辑左移)	
0000000	shamt	rs1	001	rd	0010011					
SRLI	逻辑右移(立即数)	<b>rs1, rd, shamt</b> I-type 格式 <table border="1"><tr><td>0000000</td><td>shamt</td><td>rs1</td><td>101</td><td>rd</td><td>0010011</td></tr></table>	0000000	shamt	rs1	101	rd	0010011	$x[rd] \leftarrow x[rs1] >>_l shamt;$ (其中 shamt 是 5bits 的无符号数) 把 $x[rs1]$ 右移 shamt 位，空出的位置填入 0，结果写入 $x[rd]$ 。(逻辑右移)	
0000000	shamt	rs1	101	rd	0010011					
SRAI	算数右移(立即数)	<b>rs1, rd, shamt</b> I-type 格式 <table border="1"><tr><td>0000000</td><td>shamt</td><td>rs1</td><td>001</td><td>rd</td><td>0010011</td></tr></table>	0000000	shamt	rs1	001	rd	0010011	$x[rd] \leftarrow x[rs1] <<_s shamt;$ (其中 shamt 是 5bits 的无符号数) 把 $x[rs1]$ 右移 shamt 位，空出的位置用 $x[rs1]$ 的最高位填充，结果写入 $x[rd]$ 。(算数右移)	
0000000	shamt	rs1	001	rd	0010011					
运算指令										
ADD	加	<b>rd, rs1, rs2</b> R-type 格式 <table border="1"><tr><td>0000000</td><td>rs2</td><td>rs1</td><td>000</td><td>rd</td><td>0110011</td></tr></table>	0000000	rs2	rs1	000	rd	0110011	$x[rd] \leftarrow x[rs1] + x[rs2];$ (其中 $x[rs1]$ 、 $x[rs2]$ 是 32bits 的补码数) $x[rs1]$ 和 $x[rs2]$ 相加，结果写入 $x[rd]$ 。忽略算术溢出。	
0000000	rs2	rs1	000	rd	0110011					
SUB	减	<b>rd, rs1, rs2</b> R-type 格式 <table border="1"><tr><td>0100000</td><td>rs2</td><td>rs1</td><td>000</td><td>rd</td><td>0110011</td></tr></table>	0100000	rs2	rs1	000	rd	0110011	$x[rd] \leftarrow x[rs1] - x[rs2];$ (其中 $x[rs1]$ 、 $x[rs2]$ 是 32bits 的补码数) $x[rs2]$ 减去 $x[rs1]$ ，结果写入 $x[rd]$ 。忽略算术溢出。	
0100000	rs2	rs1	000	rd	0110011					
SLL	逻辑左移	<b>rd, rs1, rs2</b> R-type 格式 <table border="1"><tr><td>0000000</td><td>rs2</td><td>rs1</td><td>001</td><td>rd</td><td>0110011</td></tr></table>	0000000	rs2	rs1	001	rd	0110011	$x[rd] \leftarrow x[rs1] <<_l x[rs2][4:0];$ (其中 $x[rs1]$ 是 32bits 的补码数) $x[rs1]$ 左移 $x[rs2]$ 位，空出的位置填入 0，结果写入 $x[rd]$ 。(逻辑左移) $x[rs2]$ 的低 5 位代表移动位数，其高位则被忽略。	
0000000	rs2	rs1	001	rd	0110011					
SLT	小于则置位	<b>rd, rs1, rs2</b> R-type 格式 <table border="1"><tr><td>0000000</td><td>rs2</td><td>rs1</td><td>010</td><td>rd</td><td>0110011</td></tr></table>	0000000	rs2	rs1	010	rd	0110011	$(x[rs1] <_s x[rs2])? rd \leftarrow 1 : rd \leftarrow 0;$ (其中 $x[rs1]$ 、 $x[rs2]$ 是 32bits 的补码数) 比较 $x[rs1]$ 和 $x[rs2]$ 中的数 (有符号数的比较)，如果 $x[rs1]$ 更小，向 $x[rd]$ 写入 1，否则写入 0。	
0000000	rs2	rs1	010	rd	0110011					
SLTU	小则设 1(无符号)	<b>rd, rs1, rs2</b> R-type 格式 <table border="1"><tr><td>0000000</td><td>rs2</td><td>rs1</td><td>011</td><td>rd</td><td>0110011</td></tr></table>	0000000	rs2	rs1	011	rd	0110011	$x[rs1] <_u x[rs2]? rd \leftarrow 1 : rd \leftarrow 0;$ (其中 $x[rs1]$ 、 $x[rs2]$ 是 32bits 的补码数) 比较 $x[rs1]$ 和 $x[rs2]$ 中的数。(比较时均视为无符号数进行无符号数比较) 如果 $x[rs1]$ 更小，向 $x[rd]$ 写入 1，否则写入 0。	
0000000	rs2	rs1	011	rd	0110011					
XOR	异或	<b>rd, rs1, rs2</b> R-type 格式 <table border="1"><tr><td>0000000</td><td>rs2</td><td>rs1</td><td>100</td><td>rd</td><td>0110011</td></tr></table>	0000000	rs2	rs1	100	rd	0110011	$x[rd] \leftarrow x[rs1] \oplus x[rs2];$ $x[rs1]$ 和 $x[rs2]$ 接位取异或，结果写入 $x[rd]$ 。	
0000000	rs2	rs1	100	rd	0110011					
SRL	逻辑右移	<b>rd, rs1, rs2</b> R-type 格式 <table border="1"><tr><td>0000000</td><td>rs2</td><td>rs1</td><td>101</td><td>rd</td><td>0110011</td></tr></table>	0000000	rs2	rs1	101	rd	0110011	$x[rd] \leftarrow x[rs1] >>_l x[rs2][4:0];$ (其中 $x[rs1]$ 是 32bits 的补码数) $x[rs1]$ 右移 $x[rs2]$ 位，空出的位置填入 0，结果写入 $x[rd]$ 。(逻辑右移) $x[rs2]$ 的低 5 位代表移动位数，其高位则被忽略。	
0000000	rs2	rs1	101	rd	0110011					
SRA	算数右移	<b>rd, rs1, rs2</b> R-type 格式 <table border="1"><tr><td>0100000</td><td>rs2</td><td>rs1</td><td>101</td><td>rd</td><td>0110011</td></tr></table>	0100000	rs2	rs1	101	rd	0110011	$x[rd] \leftarrow x[rs1] >>_s x[rs2][4:0];$ (其中 $x[rs1]$ 是 32bits 的补码数) $x[rs1]$ 右移 $x[rs2]$ 位，空出的位置用 $x[rs1]$ 的最高位填充，结果写入 $x[rd]$ 。(算数右移) $x[rs2]$ 的低 5 位代表移动位数，其高位则被忽略。	
0100000	rs2	rs1	101	rd	0110011					
OR	或	<b>rd, rs1, rs2</b> R-type 格式 <table border="1"><tr><td>0000000</td><td>rs2</td><td>rs1</td><td>110</td><td>rd</td><td>0110011</td></tr></table>	0000000	rs2	rs1	110	rd	0110011	$x[rd] \leftarrow x[rs1]   x[rs2];$ $x[rs1]$ 和 $x[rs2]$ 接位取或，结果写入 $x[rd]$ 。	
0000000	rs2	rs1	110	rd	0110011					
AND	与	<b>rd, rs1, rs2</b> R-type 格式 <table border="1"><tr><td>0000000</td><td>rs2</td><td>rs1</td><td>111</td><td>rd</td><td>0110011</td></tr></table>	0000000	rs2	rs1	111	rd	0110011	$x[rd] \leftarrow x[rs1] \& x[rs2];$ $x[rs1]$ 和 $x[rs2]$ 接位取与，结果写入 $x[rd]$ 。	
0000000	rs2	rs1	111	rd	0110011					
同步指令										
FENCE	同步内存和 I/O	<b>pred, succ</b> I-type 格式 <table border="1"><tr><td>0000</td><td>pred</td><td>succ</td><td>00000</td><td>000</td><td>00000</td><td>0001111</td></tr></table>	0000	pred	succ	00000	000	00000	0001111	在后续指令中的内存和 I/O 访问对外部 (例如其他线程) 可见之前，使这条指令之前的内存及 I/O 访问对外部可见。 比特中的第 3,2,1 和 0 位分别对应于设备输入、设备输出、内存读写。 例如 fence r,rw，将前面读取与后面的读取和写入排序，使用 pred = 0010 和 succ = 0011 进行编码。
0000	pred	succ	00000	000	00000	0001111				
断点与调用指令										
ECALL	环境调用	无参数 I-type 格式 <table border="1"><tr><td>000000000000</td><td>00000</td><td>000</td><td>00000</td><td>1110011</td></tr></table>	000000000000	00000	000	00000	1110011	通过引发环境调用异常来请求执行环境。		
000000000000	00000	000	00000	1110011						
EBREAK	环境断点	无参数 I-type 格式 <table border="1"><tr><td>000000000001</td><td>00000</td><td>000</td><td>00000</td><td>1110011</td></tr></table>	000000000001	00000	000	00000	1110011	通过抛出断点异常的方式请求调试器。		
000000000001	00000	000	00000	1110011						

#### 11.3.4 RV32I 的伪指令 (Pseudo Instruction)

RV32I 指令集如上所见。相较于 x64 之类的重量级指令集，RV32I 将指令数精简到了几十条以内。这么少的指令数如何保证功能的完整性？

RV32I 通过基础的指令组合或特殊的指令参数实现了许多其它的指令操作，来达到实现更多功能的目的。这种技巧被称作伪指令，伪指令并不存在于指令集中，但它可以由指令集中的一条或多条原生指令通过组合实现。

下面展示了几个伪指令的例子：

伪指令名	功能	基于 RV32I 的实现
NOP	什么都不做。	ADDI x0,x0,0
SNEZ rd,rs	如果 rs 不为 0，则将 rd 设置为 1，否则设置为 0。	SLTU rd,x0,rs
BEQZ rs, offset	如果 rs 为 0 则跳转。	BEQ rs,x0,offset

3 个 RISC-V 伪指令例子，都可以被替换为实际的 RV32I 指令

表中的这三条伪指令都用到了 x0 寄存器，这也是为什么 RISC-V 要规定 x0 的值总是为 0，因为许多伪指令都需要依赖一个这样的值总为 0 的寄存器。此外，RV32I 将大部分的数据类型都统一为了补码数 (如 ADDI 中的 immediate)，利用补码数的正负也可以实现更多功能的伪指令 (如令 ADDI 中的 immediate 为负数，那么就能实现立即数减法)。

有关 RISC-V 的更多伪指令可以在 RISC-V 手册中了解。(有的伪指令还依赖了 RISC-V 的一些拓展指令集)

伪指令在指令集设计中是一个非常重要的技巧，指令集并不是多多益善，复杂的指令集意味着复杂的 CPU 实现，精简的指令集在实现中能够体现出大量的优势。

指令集也不是越精简就越好。如果指令过少而导致功能缺失，一些功能反而需要更多的指令来组合实现，这将导致程序体积的大大增加，最终降低执行效率。极端的例子就是 OISC，一类只 1 条指令的指令集，这类指令集只有理论研究或教学的意义，缺乏应用价值。

编程时，程序员可以使用伪指令编写程序。但程序代码被编译器 (Complier) 转为二进制编码并导入内存执行时，这些伪指令都会被一个或多个真正的指令替代。

这样，在面向程序员 (或编译器) 时，指令集是功能丰富强大而方便的，因为不同的伪指令提供了相当多的指令功能。面向 CPU 时，指令集却又是精简而易于执行的，因为程序中的伪指令都被替换为了真正的指令或指令序列，而真正的指令集要精简地多。何等巧妙！

## 11.4 认识 RV32I 汇编

### 11.4.1 汇编语言 (Assembly Language) 概述

CPU 只能识别二进制串，因此存储器中的程序也是以二进制串的形式存储的，用二进制串直接编写的程序叫做机器语言。

计算机刚发明时，人们是直接用机器语言 (Machine Language) 在计算机上编程的。但这给大家带来了极大的不便，因为机器语言的可读性非常低。为了提高程序的可读性，大家就用了一些英文缩写来代替原本的二进制串，汇编语言就由此诞生了。

分别用机器语言和汇编语言编写的同一段代码

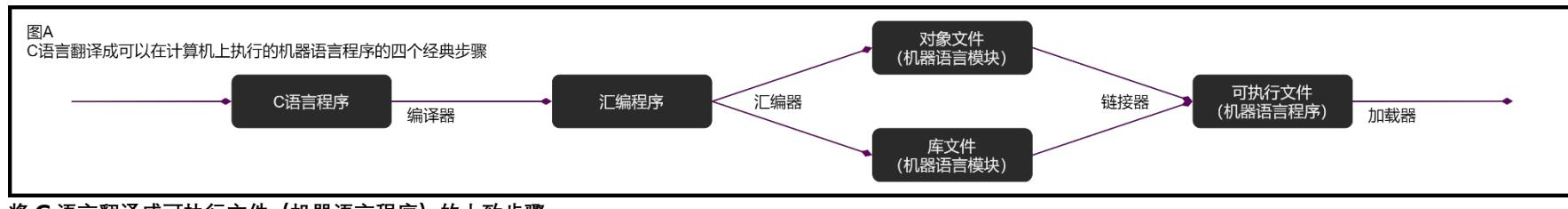
RV32I 机器语言 (二进制)	RV32I 汇编语言	评注
0000000000001 00000 000 01100 0010011	addi x12, x0, 1	将 x12 的值设置为 1。
0000000000001 00000 000 01101 0010011	addi x13, x0, 1	将 x13 的值设置为 1。
0000000 01101 01100 000 01100 0110011	add x12, x12, x13	将 x12 和 x13 的值相加，结果存入 x12。
0000000 01101 01100 000 01101 0110011	add x13, x12, x13	将 x12 和 x13 的值相加，结果存入 x13。

分别用机器语言和汇编语言写的同一段程序

上表展示了分别用机器语言和汇编语言编写的同一段代码，可以看出汇编语言的可读性比机器语言要高不少。

当然，CPU 仍然只能识别二进制代码，所以在执行汇编程序前必须将汇编程序转换为其对应的机器语言，将汇编语言转换为机器语言就用到了汇编器 (Assembler) 和链接器 (Linker)。

### 11.4.2 从高级语言到机器语言



图A  
C语言翻译成可以在计算机上执行的机器语言程序的四个经典步骤

上图展示了将高级语言“c 语言”逐步翻译成 CPU 能直接理解的可执行文件（机器语言程序）的大致过程。

由于汇编语言和机器语言开发在各方面不利与现代编译器的发展，当下已经很少有程序员直接使用汇编语言和机器语言编写程序了，取而代之的是使用一种或多种高级语言来进行程序开发。（如 c、python、java、javascript 等，它们各有不同的优劣势）

高级语言是抽象且对人而言是直观的，也因此高级语言与汇编语言或机器语言的对应关系十分复杂，甚至同一段高级语言编写的代码能通过编译器翻译成数种功能等价的汇编代码。（当然，性能或者稳健性等方面可能会有不同优劣势）

总之，读者应该明白的一个事实就是，当下几乎已经没人直接用汇编语言或者机器语言直接在机器上编程了。也就是说，指令集与 CPU 的服务对象不是人，而是编译器与操作系统。（操作系统是一个权限较高的程序，它主要的职责是管理其它程序与资源）

如果在设计指令集与 CPU 之初不考虑编译器与操作系统 (Operating System) 的存在，就会犯下许多谬误。

从高级语言的源代码生成可执行文件的具体过程涉及到了一些编译原理知识，这里就不展开讨论了。

另外，有关 RV32I 汇编的更多的说明请参考 RISC-V 中文手册的第三章“RISC-V 汇编语言”！（包括编程规范与伪指令等许多东西）

## 11.5 内存中的数据

### 11.5.1 大小端序

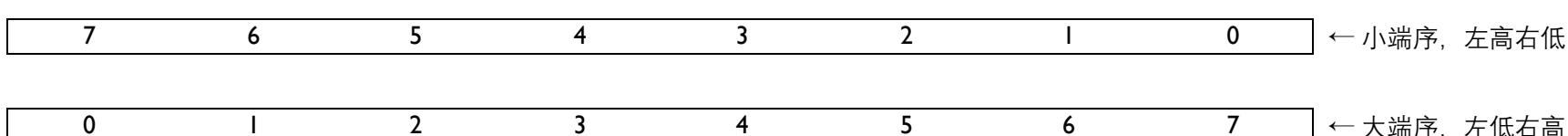
所谓的大小端序就说一个数的最高位在最左边还是最右边的意思。或者结合实际来说，是指内存中数据的排放顺序。

如果 n 个二进制位由高到低从左到右排列，就说它们采用的是小端位序。如果是由高到低从右到左排列的话，就说它们采用的是大端位序。（位序）

同理，如果 n 个字节由高到低从左到右排列，就说它们采用的是小端字节序。如果是由高到低从右到左排列的话，就说它们采用的是大端字节序。（字节序）

其它也同理。

下图直观地展示了大小端序的区别：



不同设备、不同指令集等等，都可能采用不同的大小端序标准。（因为你不可能要求所有人都统一一个标准）如果在处理数据时没采用正确的端序就可能导致错误。如大端序数据的二进制加法是向左进位的，而小端序数据是向右进位的，混淆它们可能导致在计算加法时使用预期外的进位方向。或者对比字符串时也可能把字符串读反，比如把“backwards”读成“sdrawkcab”。

### 11.5.2 数据对齐

前面说过，RV32I是按字节寻址的，因此在内存中一个 $N$ bits长的数据可以从内存的第 $n$ 个字节处开始存放。比如一个32bits的数据可以从内存的第3个字节处开始存放，那么它将存放在内存中的第3、4、5、6这几个字节处。

为了提高数据的存取效率，这些数据在实际情况中并不会随意地摆放在内存中，而是以“对齐”的方式存放在内存中。下表就分别展示了不同大小的对象在内存中的对齐/不对齐情况：

对象宽度	第 $n$ 个字节单元														
	0	1	2	3	4	5	6	7							
1 byte (byte,字节)	对齐了	对齐了	对齐了	对齐了	对齐了	对齐了	对齐了	对齐了							
2 bytes (half word,半字)	对齐了		对齐了		对齐了		对齐了								
2 bytes (half word,半字)		没对齐		没对齐		没对齐		没对齐							
4 bytes (word,字)	对齐了			对齐了											
4 bytes (word,字)	没对齐				没对齐										
4 bytes (word,字)					没对齐			没对齐							
8 bytes (double word,双字)	对齐了														
8 bytes (double word,双字)	没对齐														
8 bytes (double word,双字)					没对齐										
8 bytes (double word,双字)						没对齐									
8 bytes (double word,双字)							没对齐								
8 bytes (double word,双字)								没对齐							
8 bytes (double word,双字)	没对齐														

在字节寻址计算机中，字节、半字、字和双字对象的对齐与未对齐地址。对于每种未对齐示例，一些对象需要两次存储器访问才能完成。每个对齐对象总是在一次存储器访问中完成，只要存储器与对象的宽度相同即可。上表显示的存储器宽度为8个字节。标记各列的字节偏移指定了该地址的低3位（即字节寻址模式下，pc值的最低3位）

对齐的正式定义：大小为 $s$ 字节的对象，其字节地址为 $A$ ，如果 $A \bmod s = 0$ ，则该对象的寻址是对齐的。（即 $A$ 可被 $s$ 整除）

由于存储器通常与一个字或双字的倍数边界对齐，所以非对齐寻址会增加硬件复杂度。一次非对齐的存储器访问可能需要涉及多个对齐的存储器访问。因此即使计算机支持非对齐寻址，采用对齐寻址的程序也可以运行得更快一些。

### 11.5.3 内存空间的划分

### 11.5.4 内存空间的管理

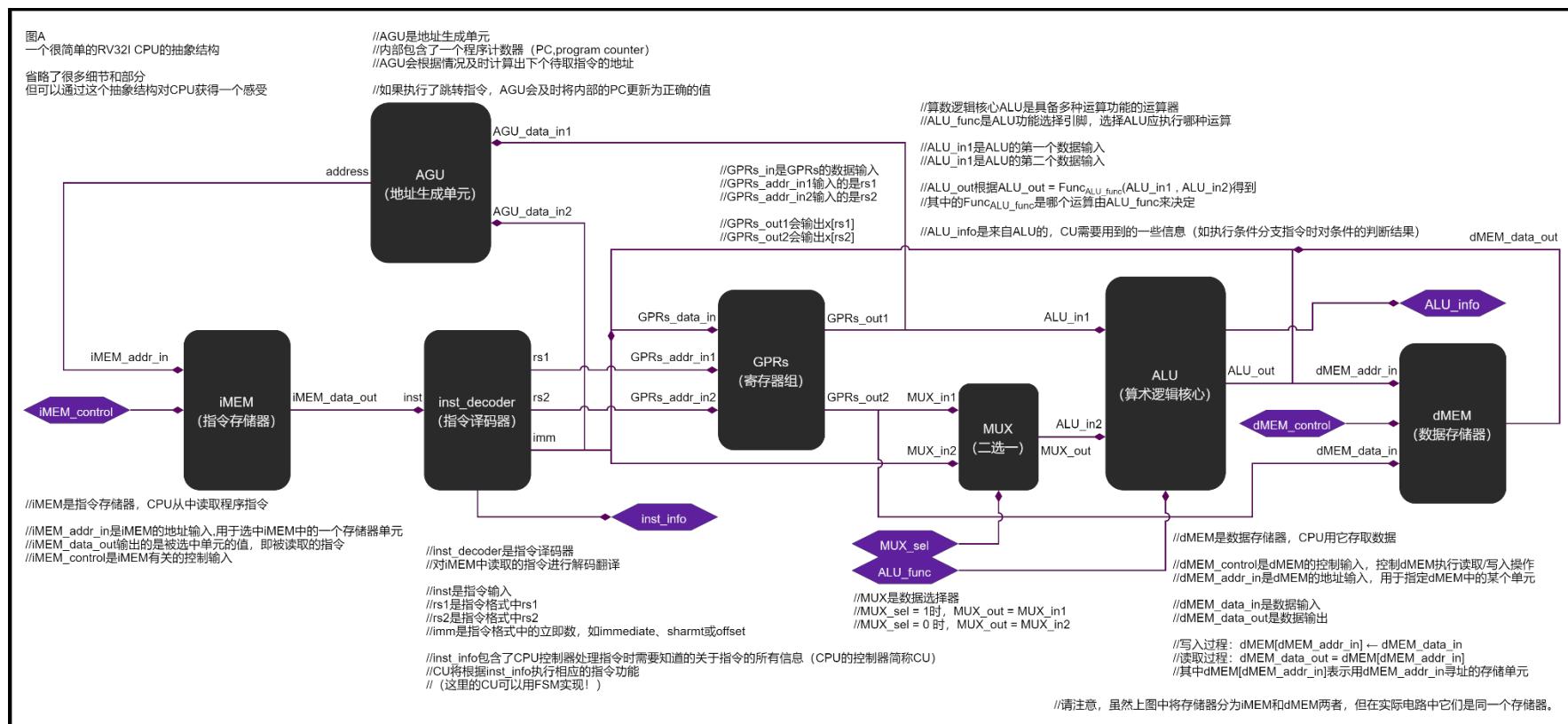
## I2一个简单的 RV32I 处理器

12.1 前言

这章我们会了解一个简单的 RV32I 处理器内部是什么样子的。

## 12.2 一个单周期 RV32I 处理器的内部抽象结构

### 12.2.1 抽象的内部架构视图



一个 **RV32I** 处理器的内部抽象视图。在图中, iMEM 用于存储指令, dMEM 用于存储数据 (iMEM 和 dMEM 本质是同一个存储器, 在实际情况在指令和数据都存储在同一个存储器中)。inst\_decoder 将指令进行解码并输出指令包含的信息。GPRs 是寄存器组。ALU 是负责运算的部件, 运算数据来源于 GPRs 或者指令中的立即数, 结果会保存至 GPRs 中。在图中 ALU 也被用于处理一些其它的东西, 如计算并生成内存地址 (指令地址或数据地址), 或者判断指令的执行条件 (如在指令 BEQ 中就需要判断两个寄存器的值是否相等)。AGU 负责管理程序计数器 pc 的值, 生成待取指令的地址。为了提高地址生成的效率, 通常大家会让 AGU 集成一些运算器以独自完成许多生成指令有关的计算 (而不是将这些计算交给 ALU), 这么做不仅能提高地址生成的效率, 还在一定程度上缓解了 ALU 的数据处理压力。(当然, 提供给 dMEM 的数据地址的有关计算也可以交给 AGU 处理, 如果这么做能带来改善) (如果某些指令刚好要用到 AGU 内部的运算器, 也可以直接让 AGU 内的运算器处理这个指令的相关运算, 这样在 ALU 中就不必实现相关的运算功能, 或者减少 ALU 的运算压力)

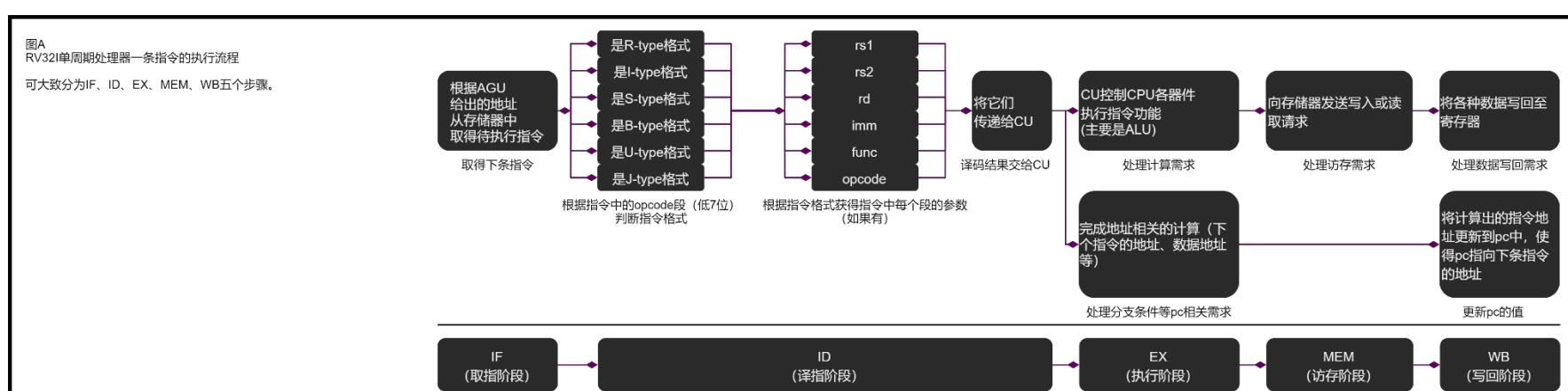
上图就是一个简单的 RV32I 处理器的内部抽象架构视图，它的功能就是从指令存储器中不断读取并执行计算机指令。（RV32I 指令）

图中没有画出处理器的控制器 (CU) 部分，但你应该明白它一直在管理和控制 CPU 的运行，CU 也是 CPU 最复杂的部分。

抽象视图隐藏了内部复杂的电路细节，方便你理解它的高层次结构。实际电路要复杂得多，并且会多出许多其它部分的电路。

接下来我们看看这个处理器的运行过程是什么样的。

### 12.2.2 抽象的执行过程



如上图 A，各指令的执行流程可以被依次划分为三个阶段：

1. IF : 取出待执行的指令。
  2. ID : 翻译指令, 将译码结果作为参数传递给 CU, CU 根据指令配置好各器件的输入参数, 为下一阶段准备。
  3. EX : CU 控制各器件执行功能以完成各种计算和操作。(包括算数指令的计算, 分支指令的条件判断, 跳转地址的计算等)
  4. MEM : 处理访存请求, 向存储器发送需要发送的访存请求。(存储器数据读取/写入请求, 如 LB、LH、SB、SH 等)
  5. WB : 将执行指令产生的各种数据写回。(如将 ALU 的运算结果或存储器数据读取结果写回 rd)

其中 **MEM** 阶段发送访存请求后，一般不会等到访存请求完成后才进入下个阶段，因为存储器访问通常很慢。访存结果可以在将来另一条指令的周期中提交。（提交是指访存请求中的数据已经写回到存储器单元或寄存器单元里了，即对相关单元的读取操作可以得到被写的数据）

**RV32I** 不要求访存必须完成才开始执行下条指令。作为代替品，**RV32I** 提供了 **FENCE** 指令用于强制让 **CPU** 等到所有被指令指定的访存请求类型完成后才开始继续执行 **FENCE** 后的指令，以确保后续指令执行时前面的访存请求都已经提交了。

在上例中，单独设立了一个叫“AGU”的机构来完成地址相关的计算和处理，因此图 A 中地址计算和其它计算很自然地被分开了。

### I2.2.2.1 设计的抽象层次

不同的 **CPU** 可能有不同的阶段划分，一些复杂的处理器甚至划分出了几十个阶段（如 **Pentium 4** 处理器）。更复杂的划分主要是为了让流水线技术更好地运用在庞大复杂的处理器电路上。

但即使不同的 **CPU** 在阶段划分上是一致的，它们也可能在每个阶段的具体操作过程细节中大相径庭。再进一步地，即便确定了每个阶段是如何操作的，**CPU** 也可以在电路实现上存在很大区别。

更高层次的抽象（如 **CPU** 的执行阶段模型）不能直接落地成为实际的成品（如一个 **CPU** 的实际电路图），但高层次的抽象能够指导低层次应该如何实现。人的注意力是有限的，所以通过一个高层次的抽象来感受并调整一个工程的整体很有必要。

## I2.3 CPU 的更多细节

### I2.3.1 CPU 与计算机上的数据交换

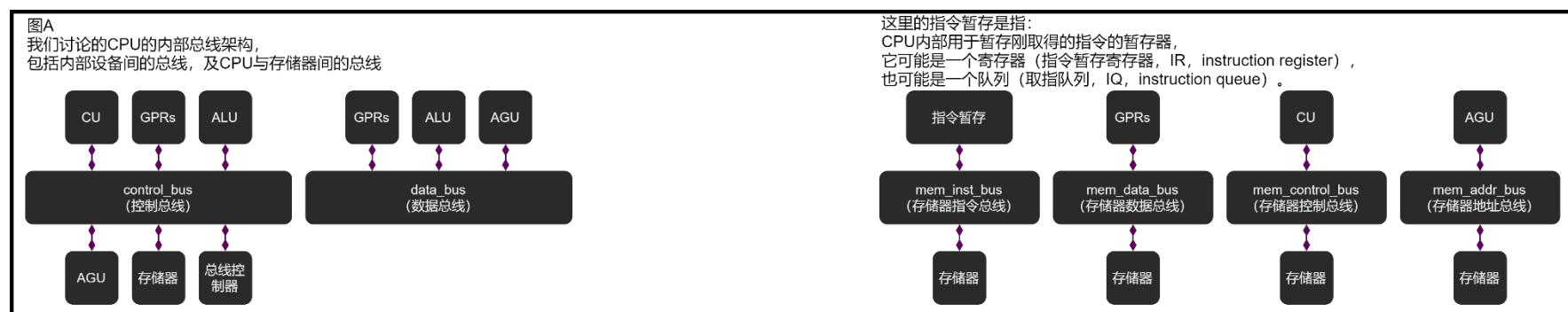


各个部件通过中间的总线 **BUS** 交换数据

总线（**BUS**），可以理解为计算机各种功能部件之间传送信息的公共通信干线，或者说是计算机中的交换节点。如上图 A 就通过了一个“总线”连接了一个 **CPU** 中的几个部件，而这些部件可以是 **CPU** 中的 **ALU**、**GPRs**、**AGU**、指令译码器等。这样，我们就可以通过控制总线来处理数据交换，这么做能大大降低 **CPU** 数据通路电路的复杂度，并提高 **CPU** 数据流动的可分析性。

实际的计算机中存在着各种总线，它们负责为不同器件提供数据交换服务。例如 **CPU** 可以有内部数据总线，负责在 **CPU** 内部组件间传递数据，比如在 **GPRs** 和 **ALU** 间传递运算输入输出。或者 **CPU** 也可以有所谓的地址总线，在 **CPU** 与 **RAM** 间传递内存地址。亦或者所谓的控制总线，即负责传递控制信号的总线。

现在我们就为之前讨论的 **RV32I** 处理器设计它的总线架构：



CPU 的总线架构。包括 CPU 内部的总线，与 CPU 和存储器间的总线。

如上图，我们用数据总线 **data\_bus** 来处理 **CPU** 内部的运算数据交换、用 **control\_bus** 来处理 **CPU** 内部的控制信号交换。然后在 **CPU** 与存储器间，我们分别用存储器指令、数据、控制、地址总线来交换 **CPU** 与存储器间的信号。图中的总线架构非常简单，因为我们讨论的这个 **CPU** 很简单。

细心的同学可以看到，我们的控制总线 **control\_bus** 上挂了一个总线控制器，这个总线控制器就是用来控制总线时序的，我们现在来讨论它。

### I2.3.2 计算机总线的时序控制

### 12.3.3 计算机中的 I/O 接口与外设



CPU 通过 I/O 接口与外设链接。外设就是外部设备，外设可以是键盘、鼠标、显示器、耳机、音响、扩展坞、显卡等等。

I/O 是 input/output 的缩写，翻译过来就是输入/输出，I/O 接口就是 CPU 与外部环境（外设）输入输出的地方。上图高度抽象地展示了 CPU、I/O 接口与外设之间的关系。

外设与 CPU 通常因为各种原因不能直接连接，这时就需要一个“I/O 接口”沟通它们。

I/O 接口通常要负责许多杂活，如：

1. 设置数据的寄存、缓冲逻辑，以适应 CPU 与外设之间的速度差异，接口通常由一些寄存器或 RAM 芯片组成，如果芯片足够大还可以实现批量数据的传输。
2. 转换数据格式，如串行信号转并行信号、并行信号转串行信号、数字信号转模拟信号、模拟信号转数字信号...
3. 协调 CPU 和外设两者在信息的类型和电平的差异，如电平转换驱动器、数/模或模/数转换器等。
4. 协调时序差异。
5. 地址译码和设备选择功能。
6. 设置中断和 DMA 控制逻辑，以保证在中断和 DMA 允许的情况下产生中断和 DMA 请求信号，并在接受到中断和 DMA 应答之后完成中断处理和 DMA 传输。

...

总之，I/O 接口的意义就是沟通 CPU 与外设，I/O 接口挺复杂的，接下来我们来看一个稍微具体点的例子。