计算机系统概论

Chapter 1 Introduction

1 Two major themes

1.1 Abstraction

Abstract **low-level details** into **high-level interfaces** that are easy to understand and use.

An underlying assumption: everything about the details is fine.

It should be careful about the components below the abstractions when combining multiple components into a larger system.

将关注的内容从一个复杂的系统中提炼出来，省略内容的具体实现，视其为一个黑箱，而更加关注如何使用这样一个黑箱、这样一个黑箱在各种情况下有怎样的行为。

然而这并不意味丢弃抽象程度更低的细节，也并不意味着没有必要了解底层系统。当我们做一些需要溯源的工作（比如追踪错误、优化分析等），我们就需要能够对“抽象”进行“解构”，以在更“低”，或者说更底层的角度去分析问题。

1.2 Hardware vs Softwear

Both are equally important, don't separate them.

2 A computer system

- CPU : operating data
- MEM : storage data
- I/O : Input & output devices
    - Input : Type input informations to CPU by keyboard
    - Output : Show the outcome done by CPU on monitor

3 Two Very Important Ideas

3.1 High-level and Low-level

All computers are capable of computing exactly the same things if they are given enough time and enough memory. That is, anything a fast computer can do, a slow computer can do as well.

高级计算机能够解决但低级计算机无法解决的问题，一般来说可以通过给低级计算机更多的计算时间或基于更多的内存来实现。所有计算机的计算能力是一样的，只是所需要的时间不同而已。

3.2 Language and Electrons

We describe our problems in English or so on. Yet the problems are solved by electrons running around inside the computer. It is necessary to transform our problem from the language of humans to the voltages that influence the flow of electrons.

计算机没法直接理解自然语言，所以需要用一种计算机能够“执行”（而非“理解”，计算机不需要理解只需要执行，因此之后说的东西叫“指令”）的“语言”去指导计算机解决问题。

4 Universal Computational Devices

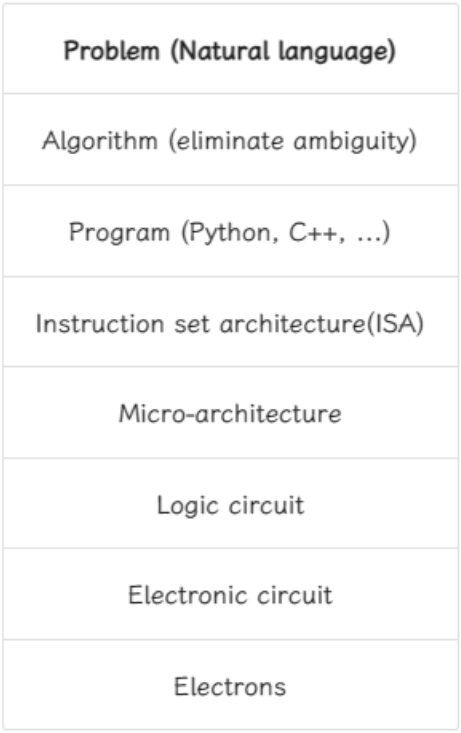
You can tell a computer how to add numbers, multiply, alphabetize a list or perform any computation you like. When there is a new kind of computation, you do not have to design a new computer, but just give the old computer a new set of instructions (a program) to carry out the new computation. This is why we say the computer is a universal computational device. It is believed that anything that can be computed by a computer if you provided it enough time and memory.

Turing machine:

One way to construct a machine more powerful than any particular Turing machine was to make a machine U that could simulate all Turing machines.

5 Level of Transfomation

hierarchy: Problems -> electrons



5.1 Problems

Natural languages may have ambiguity. However, The statement of the problem should have **No Ambiguity**.

5.2 Algorithm

Algorithm is a procedure step by step.

- **Definite** : no ambiguity.
- **Effective computability** : every step can be successfully carried out.
- **Finiteness** : the procedure will terminate.

5.3 Programs

Transform the **algorithm** into one of the **programming languages** precisely.

- **high-level languages:**

  they are independent of the computer on which the programs will execute. We say the language is“machine independent. (Python, C++,...)

- **low-level languages**:

  they are tied to the computer on which the programs will execute. There is generally one such low-level language for each computer. That language is called the assembly language for that computer. (x86)

5.4 Instuction Set Architecture (ISA 指令集架构)

为更好的控制计算机科学的复杂度，需要在不同层次间抽象出接口以提高系统的迭代效率。指令集架构就是其中非常典型的例子，它是软硬件间的结构抽象。

其本质而言，就是一系列的指令集合。其中每个指令可以看做是提前定义好的一个明确的小的硬件功能点（如：四则运算、寄存器读写等）。所有计算机支持的指令加总起来，就被称为一套指令集。如果把编程看做写作的话，那么ISA就是单词表。

e.g.

汽车的 ISA 就是人需要知道他能让车做什么，以及车需要做到人指定的任务的规范。
对于一辆车的踏板，人知道如果他踩下去，那么这辆车会刹车。车知道如果踏板受到了压力，车的硬件会让车停下。ISA 的作用就是将人踩刹车和车停下对应起来。

x86(1979 8086, 286, 386, 486), Power-PC, Sparc ISA contains:

- **指令码（opcode）**：用于描述操作的代码。
- **数据类型（data type）**：操作数（operand）的表示形式。
- **寻址模式（addressing mode）**：计算机用于查找操作数地址的机制。
- **储存能力（Address ability）**：每个内存槽位（memory-slot）的字节数。

5.5 Microarchitecture

所有的汽车都有相同的 ISA，例如所有的汽车中三个踏板的定义完全相同，即中间的是刹车、右边的是油门、左边的是离合器。 而将 ISA 实现的具体组织（微结构）是指车盖板下的“内容”。所有的汽车，其制造和模型都不尽相同，这取决于设计者在制造之前所做的权衡决策，如有的制动系统采用刹车片，有的采用制动鼓；有的是八缸发动机，有的是六缸，还有的是四缸；有的有涡轮增压，有的没有。我们称这些差异性的细节为一个特定汽车的“微结构”，它们反映了设计者在成本和性能之间所做的权衡决策。

e.g.

对同样指令集 x86, 他的微结构从 8086, 80286, 80386, 80486... 直到如今的 Skylake.

ISA 将程序转化为 01 字符串（类似于汇编中的机器码 如E8 01代表 jmp），而微结构是其对应的物理实现（电路）。因此对于同样的 01 字符串，其实现的功能相同，但可以有各种不同的物理实现；而一套微结构只能实现一类 ISA，但一类 ISA可以由不同微架构实现。

5.6 Logic Circuit

逻辑电路，通过基本的逻辑门电路去实现各种功能模块，每个模块有不同的实现方式.

编译器：高级语言→ISA

汇编器：低级语言→机器码

Chapter 2 Bits, Data Types and Operations

1 Bits, Bytes

- **bit(位)** : only 1/0
- **bytes(字节)** : 1byte = 8 bits

2 Data Type

We say a **particular representation** is a data type if there are operations in the computer that can operate on information that is encoded in that representation.

2.1 Integer

2.1.1 Unsigned Interger (非负数)

n bits, can represent 2^n numbers. range : $[0, 2^{n-1}]$

2.1.2 Signed Interger (有正有负)

- **signed-magnitude(原码) :** 符号位 + 绝对值, 将最高为作为符号位, **0为正, 1为负**.
- **1's Complement(反码)** : **按位取反**　　range : $[-2^{n-1}+1, 2^{n-1}-1]$
- **2's Complement(补码)** : **按位取反加一**　range : $[-2^{n-1}, 2^{n-1}-1]$
- 原码不能直接计算，例如001 + 101，需要进行复杂的转换。计算机中能够方便的取到反码，一个数加上自己的反码得到全1，再加上1恰好为0，所以补码能保证相反数之和为0.

e.g.

01111(15), 反码10000, 相加得到11111, 再加1得到100000, 将溢出的1舍去得到00000.

Representation	Value Represented			
	Unsigned	Signed Magnitude	1's Complement	2's Complement
00000	0	0	0	0
00001	1	1	1	1
00010	2	2	2	2
00011	3	3	3	3
00100	4	4	4	4
00101	5	5	5	5
00110	6	6	6	6
00111	7	7	7	7
01000	8	8	8	8
01001	9	9	9	9
01010	10	10	10	10
01011	11	11	11	11
01100	12	12	12	12
01101	13	13	13	13
01110	14	14	14	14
01111	15	15	15	15
10000	16	−0	−15	−16
10001	17	−1	−14	−15
10010	18	−2	−13	−14
10011	19	−3	−12	−13
10100	20	−4	−11	−12
10101	21	−5	−10	−11
10110	22	−6	−9	−10
10111	23	−7	−8	−9
11000	24	−8	−7	−8
11001	25	−9	−6	−7
11010	26	−10	−5	−6
11011	27	−11	−4	−5
11100	28	−12	−3	−4
11101	29	−13	−2	−3
11110	30	−14	−1	−2
11111	31	−15	−0	−1

Figure 2.1 Four representations of integers.

2.1.3 Overflow

本质上是太大或太小

- **正 + 正 = 负** : the sign bit becomes 1 after adding.
- **负 + 负 = 正** : the sign bit becomes 0 after adding.
- 正数和负数相加不会溢出.

2.1.4 Extension

两个位数不同的数据进行计算时需要对齐，否则会出错.

如 34-2=32，而 0010 0010(34) + 1110(-2) = 0011 0000(48)。所以需要将位数少的数据扩展到相同位数后再进行计算。有以下两种扩展方式 :

- 无符号扩展(zero extension): 在高位补 0，如 1100 扩展为 8 位得到 00001100。通常用于将数据看作无符号数的情况.

- 有符号扩展(sign extension): 在高位补符号位的值，如1100符号位为1，则扩展成11111100，0100符号位为0，则扩展成00000100.

e.g.

0010 0010(34) + 1111 1110(-2) = 0010 0000(32)

2.1.5 Conversion between binary and decimal

用2乘十进制小数，将积的整数部分取出，再用 2 乘余下的小数部分，再将积的整数部分取出，如此直到积中的整数部分为 0/1，此时 0/1 为二进制的最后一位。或者达到所要求的精度为止。然后把取出的整数部分按顺序排列起来，先取的整数作为二进制小数的高位有效位，后取的整数作为低位有效位。

2.2 Logical Variables

- bit vector : number string of 0/1.对于多位二进制逻辑运算，可将其看作一个比特向量，对于每一位进行逻辑运算.
- functions : AND, OR, XOR(异或, 相同为0), NOR(或非, 全0为1), NAND(与非, 全1为0), Equivalence

$X\ AND\ Y \Leftrightarrow X \cdot Y$

$X\ OR\ Y \Leftrightarrow X + Y$

$X\ XOR\ Y \Leftrightarrow X \oplus Y$

2.3 Floating Point Number

	S(符号位)	exp(指数位)	frac(尾数位)	位数
Float	1bit	8bits	23bits	32bits
Double	1bits	11bits	52bits	64bits

- S(Sign) : 符号位，0正1负。
- exp(Exponent) : 指数位，表示浮点数的指数部分。指数位的值需要进行偏移后才能得到实际的指数。例如，对于32位浮点数，通常采用偏移值为127。
- frac(Fraction) : 尾数位，表示浮点数的小数部分。尾数位通常采用二进制表示法，并且规定尾数位的最高位为1。因此在表示时可以省略最高位的1，只存储剩余的有效位数。

数据格式:

- Normalized Form : $N = (-1)^S \times 1.frac \times 2^{exp-127}$ ， $exp \in [1, 254]$.
- Subnormal Form : $N = (-1)^S \times 0.frac \times 2^{-126}$ ， $exp = 0$.
- Infinity : $exp = 255, frac = 0$ 即为无穷，用符号位的0/1区分正负无穷.
- 0 : $exp = 0, frac = 0$ 即为0.
- 其他情况为 NaN.

转换方法:

e.g.1 0 01111110 10100000000000000000000

- S = 0
- exp : 01111110 = 126 exp = 126 - 127 = -1 != 0
- frac = 101

- $N = (-1)^0 \times 1.101 \times 2^{-1} = 0.1101 = 0.8125$

e.g.2 104.375 转二进制

整数部分判断奇偶，除2向下取整，奇1偶0

- 104 偶0 52 偶0 26 偶0 13 奇1 6 偶0 3 奇1 1 奇1

  104 -> 110 1000

小数部分乘2，判断是否小于1，小于取0大于等于取1，大于等于判断完减去1

- 0.375 × 2 = 0.75 < 1 0

  0.75 × 2 = 1.5 >= 1 1

  0.5 × 2 = 1 >= 1 1

  1 - 1 = 0

  0.375 -> 0.011
- 104.375 -> 110 1000.011

e.g.3 1010.101转十进制

$B = b_m b_{m-1} \cdots b_1 b_0 . b_{-1} b_{-2} \cdots b_{-n}$ **转化为十进制为** $N = \sum_{i=-n}^{m} b_i \times 2^i$

- 1010 -> 10
- 0.101 -> $2^{-1} + 2^{-3} = 0.625$
- 1010.101 -> 10.625

2.4 ASCII码

用八位二进制数记编码字符

2.5 Hexadecimal

以 x开头用以识别，对于二进制数据可以使用十六进制数据进行表示，四位二进制串可以得到一个十六进制数

如0000 0101 1101 1001表示为 x05d9.

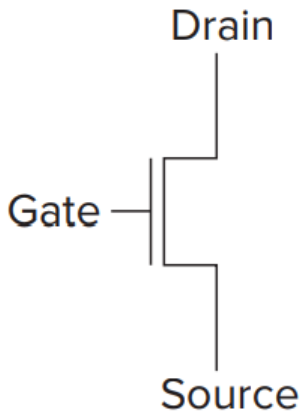
0000	0101	1101	1001
0	5	d	9

Chapter 3 Digital Logic Structures

1 Transitors(晶体管)

Gate为栅极，Drain 为漏极，Source 为源极

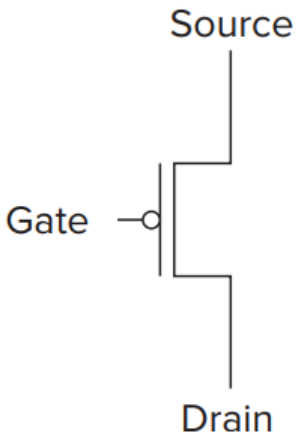
- **N-Type**



Gate施加电压时，Drain和Source导通(short circuit)

Gate无电压时，Drain和Source断开(open circuit)

- **P-Type**



Gate施加电压时，Drain和Source断开(open circuit)

Gate无电压时，Drain和Source导通(short circuit)

N-type : N断P通

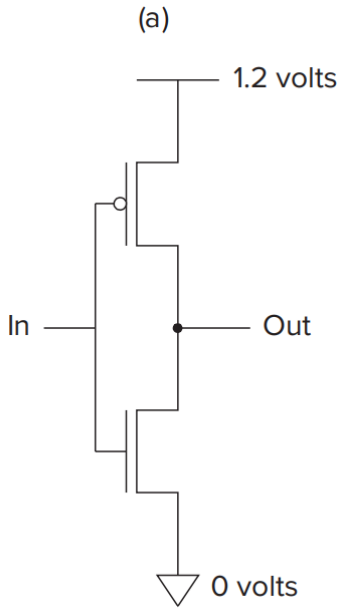
P-type : P断N通

其中 P 指 Positive(1.2V/接电源正极), N 指 Negtive(0V/接地/接电源负极)

- P-Type只能连高电压，N-Type只能连低电压.
- 不能存在输入使得高电压端和低电压端直接连接.
- 无论输入为高电压还是低电压，输出必须直接和高电压端或者低电压端连接.

2 Logical Gates

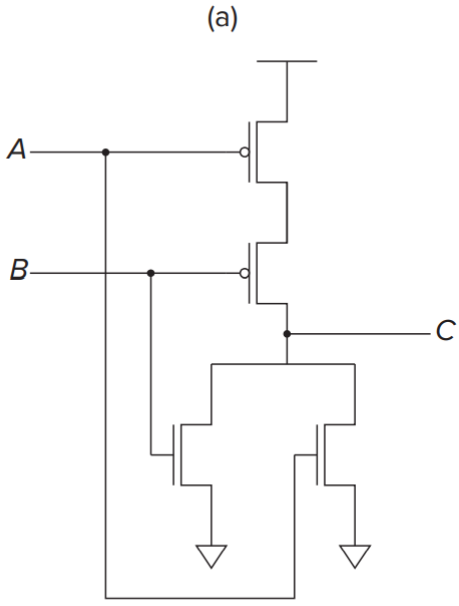
2.1 Not Gate(Inverter) 非门



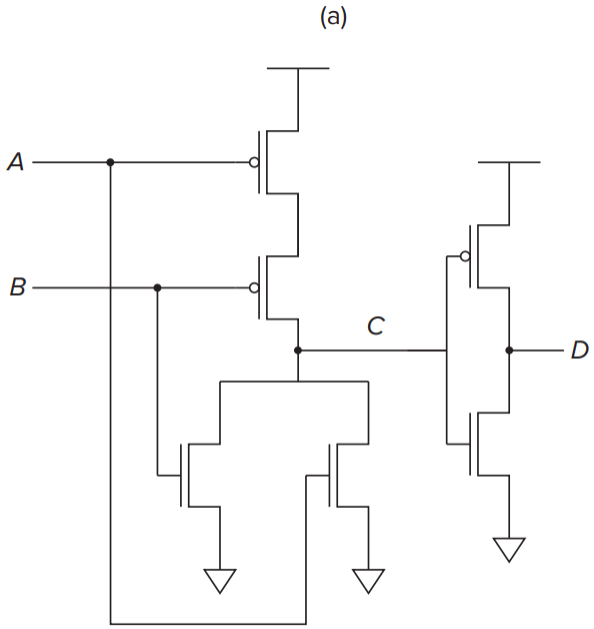
In	Out
0	1
1	0

2.2 OR and NOR Gates 或门

NOR Gate



OR Gate



A(in)	B(in)	C(NOR)	D(OR)
0	0	1	0
0	1	0	1
1	0	0	1
1	1	0	1

2.3 AND and NAND Gates 与门

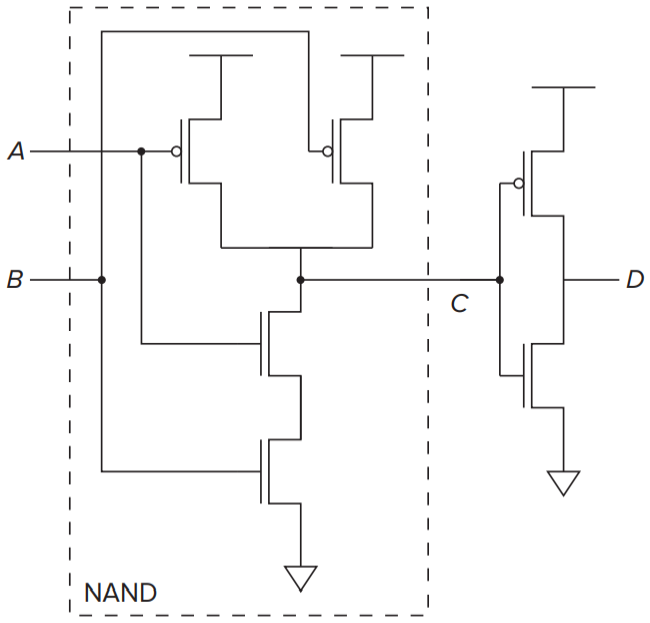


Figure 3.8 The AND gate.

A(in)	B(in)	C(NAND)	D(AND)
0	0	1	0
0	1	1	0
1	0	1	0
1	1	0	1

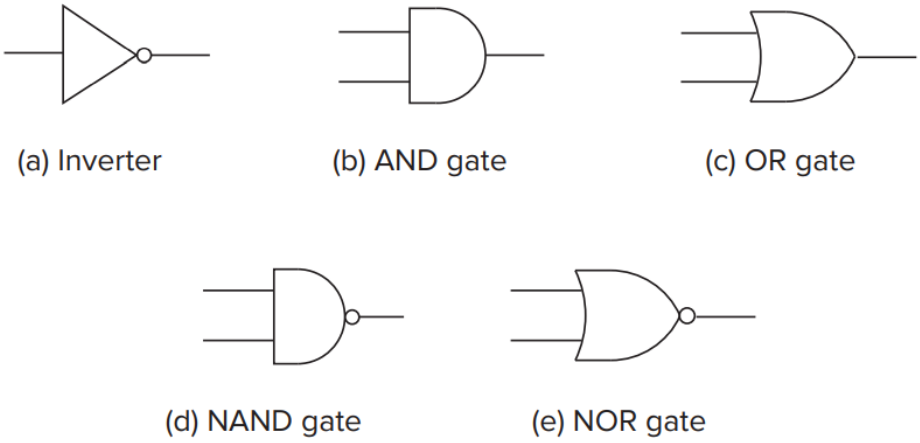
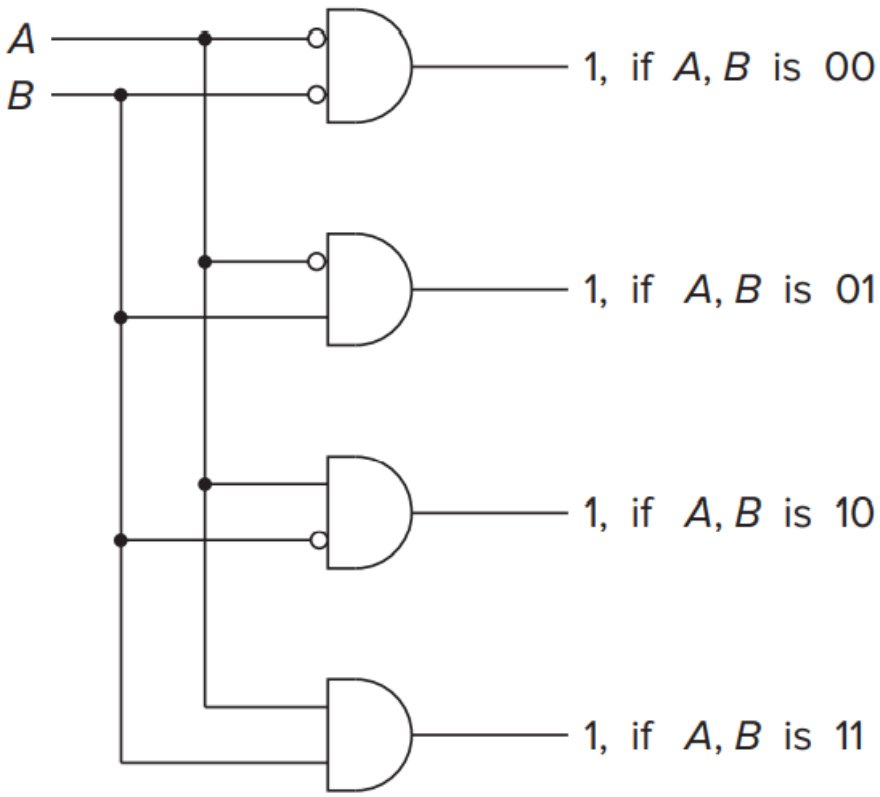


Figure 3.9 Basic logic gates.

3 Combinational Logic Circuits 组合逻辑电路

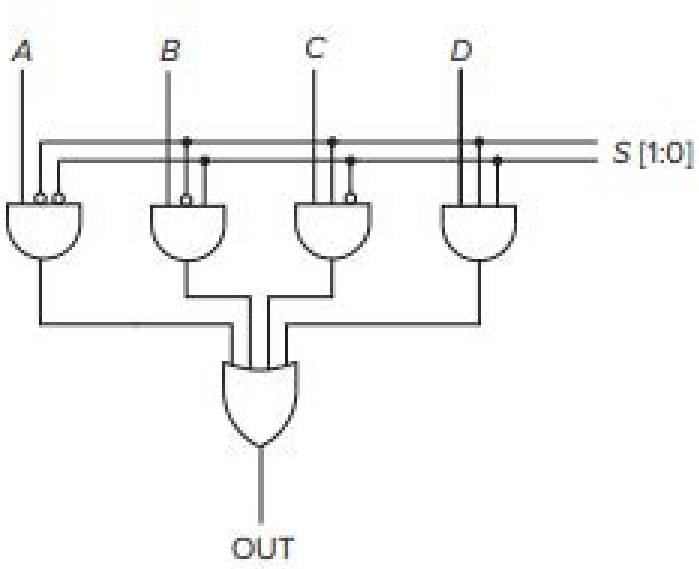
3.1 Decoder 译码器



对于所有输入的情况，只有一个输出结果为1，其他结果为0。则可以根据具体是哪个输出结果为1来判断信号源。

比如当第二个与门输出为1时AB为01。nbits输入对应 2^n bits输出。

3.2 MUX (Multiplexer 数据选择器)

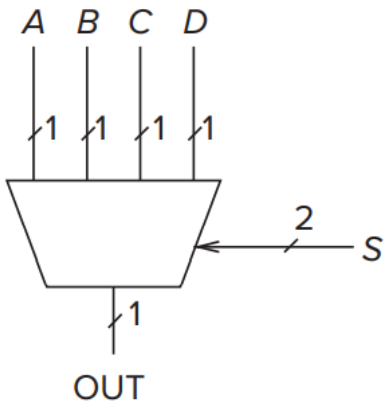


对于所有输入情况，只选择一个进行输出。

比如根据 S[1:0] 对 ABCD 进行选择，若 S 为 10，ABCD 为1 0 0 1，则输出数据 A。

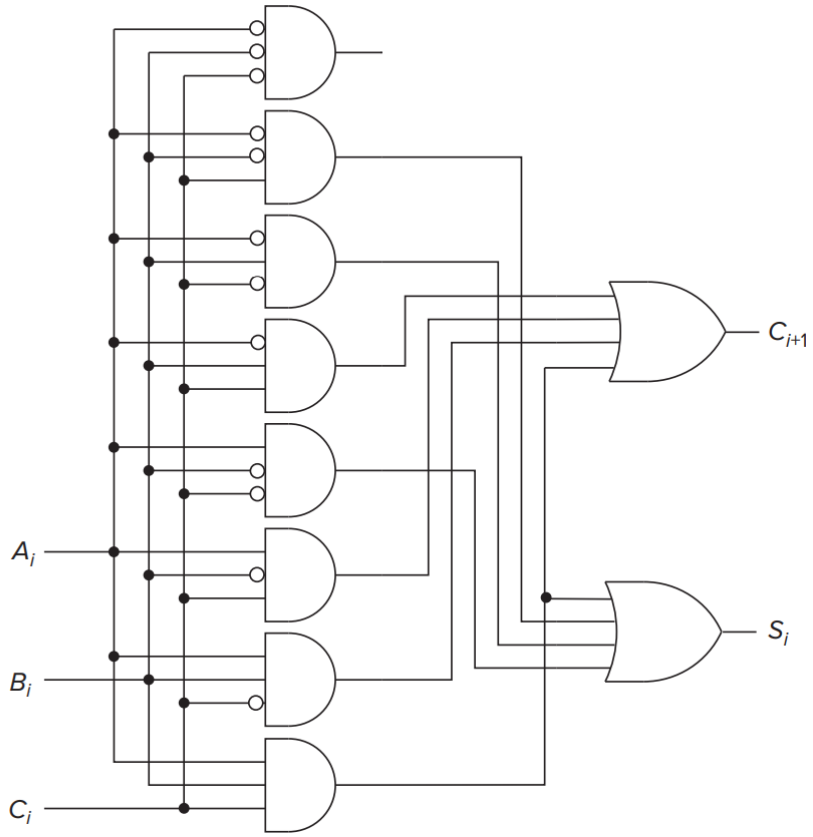
可以理解为对选择信号先进行译码，然后再将译码后的结果和所有待选择的输入作一个与操作，最后再将所有与的结果进行或操作得到最终的结果。

数据选择器用提醒表示，一根线上面接的数字表示数据比特数。



3.3 One-Bit Adder 全加器

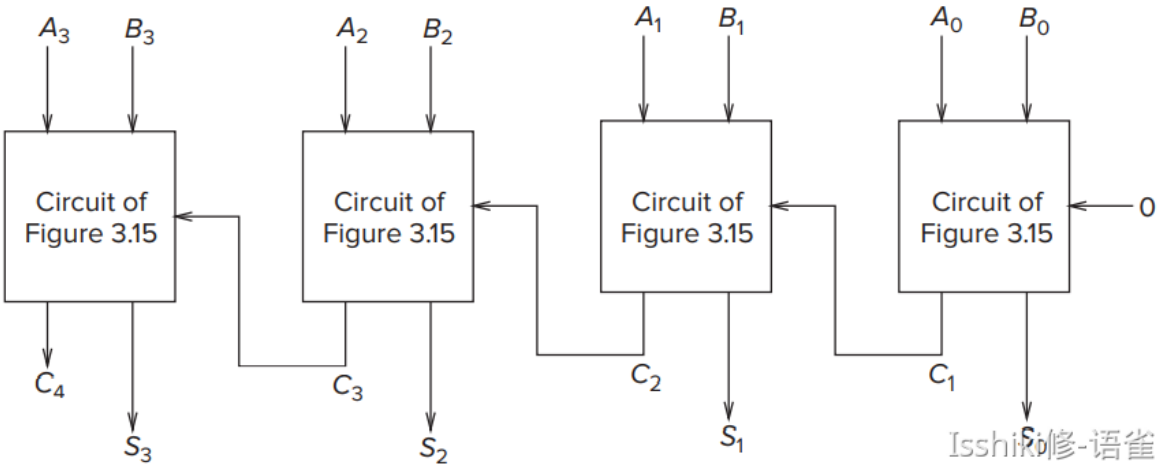
全加器输入两个相加的数据对应位的比特及前一位的进位，输出本位的数据以及下一位的进位。



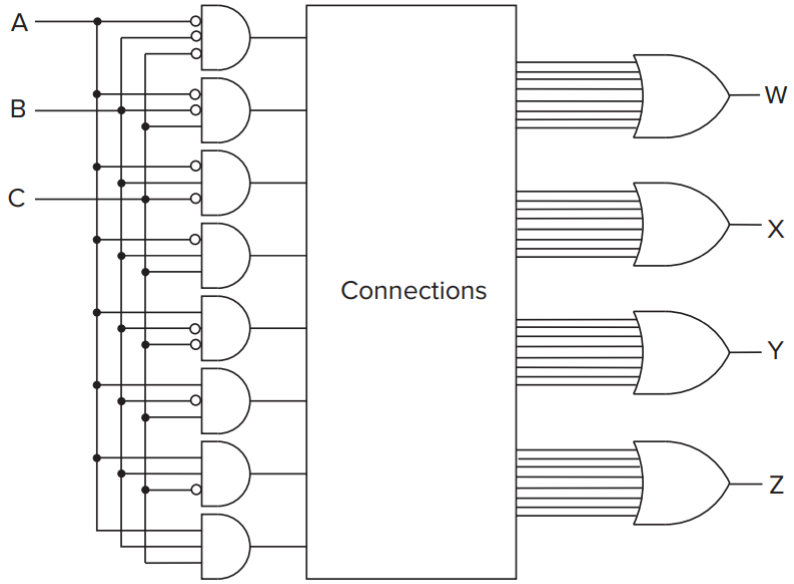
如两个四位二进制数相加，则第 i 个全加器输入的是 A_i, B_i以及前一位的进位 C_{i-1}，输出的结果是本位的结果 S_i 和输入到下一位的进位 C_i。

A_i	B_i	C_{i-1}	C_i	S_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

将上述加法器连接即可得四位加法器：



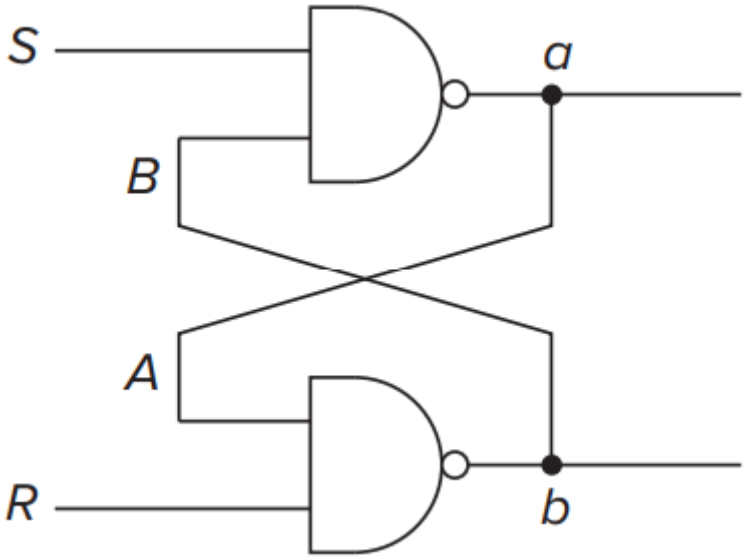
3.4 PLA(Programmable Logic Array) 可编程逻辑阵列



A programmable logic array.

这个电路是可以编程的，可以通过编程将各个与门输出连接到或门的输入中。

3.5 RS Latch 锁存器



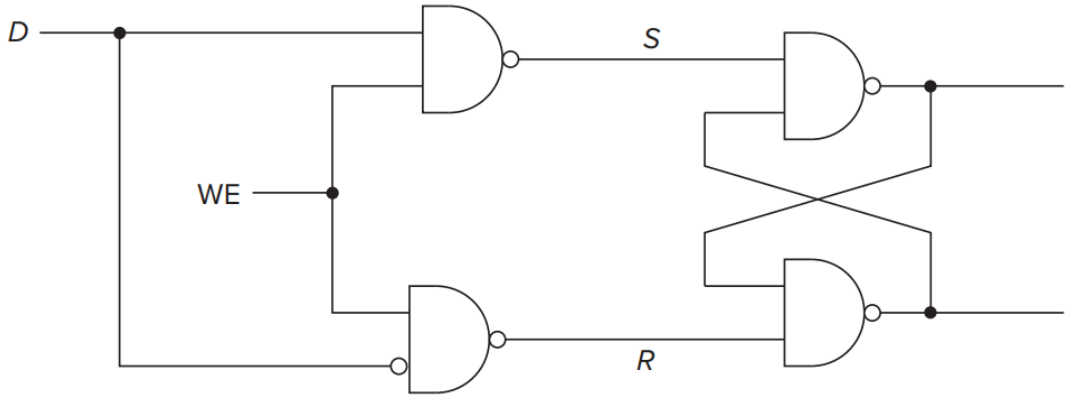
S R为输入，a b为输出，a b值一定相反，其中a 是储存、输出的值。

- 起初 S = R = 1，ab不确定

- 将 S(Set) 拨到 0，则a -> 1，成功将 1存入。再将 S 拨到 1，则 S=R=1，只要 S R不变，则储存的值不变。
- 将 R(Reset) 拨到 0，则a -> 0，成功将 0存入。再将 R 拨到 1，则 S=R=1，只要 S R不变，则储存的值不变。

未定义行为：S = R = 0，但若想保持 a = b = 1，需要将 S 或 R 拨为 1，但两者不可能同时变成 1，若S先变成 1，则a = 0。若 R 先变成 1，则 a = 1，由于不能确定 R S 谁先变化，所以 a 处于不确定的状态。最后保持的值取决于锁存器的电路性质。

3.6 Gated D Latch (D 锁存器)



WE : 写使能 write enable

- WE = 1 (可写) 时将D处的值写入锁存器中
- WE = 0 (只读) 时保持当前锁存器的值

4 The Concept of Memory 内存

可以理解为一大块存储着数据的东西，它可以通过地址来访问一个特定位置的数据，类似于C语言的数组，能够读取某个位置的数据，或者将数据写入到某个位置。

- **Address Space**

  地址为与每个内存位置关联的唯一标识符。**Address Space为唯一可识别位置的总数**

  address space表示的是编码内存位置的能力，与实际的内存位置数量无关，看线的数量即可，无需管具体画了几个内存。

- **Addressability**

  每一个内存位置存储的比特的数量即为**Addressability**

假设玉湖七幢 4 楼为一块内存, 其中这一楼一共有 60 间寝室, 这就是内存空间 (因为每间寝室都有独一无二的寝室号)。而每间寝室都是四人间, 因此可寻址能力是 4, 表示最小的寻址单元。

对于一个 2^n-by-m-bit memory，其Address Space为 2^n，Addressability为m. 几根线就是2的几次方。

e.g. 2^2-by-4-bit memory

- the data at address 2 is 0001

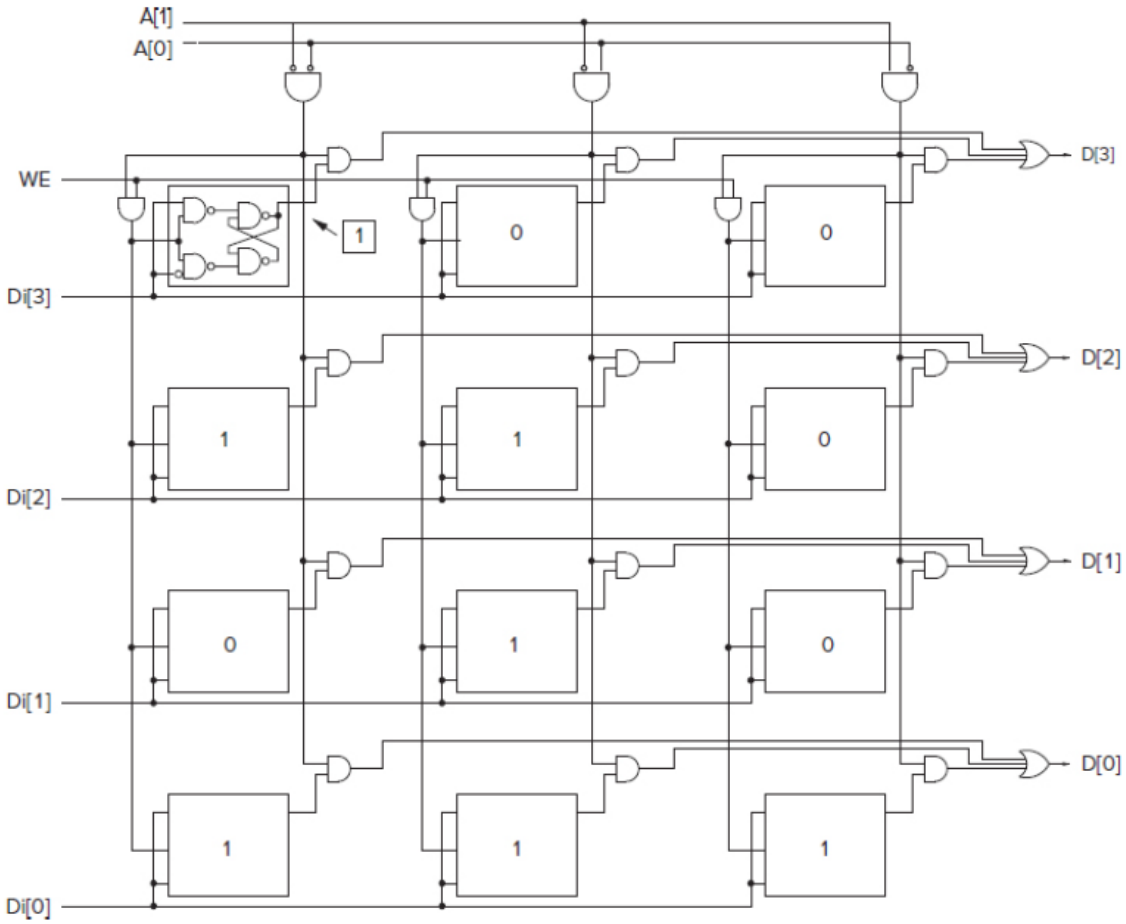
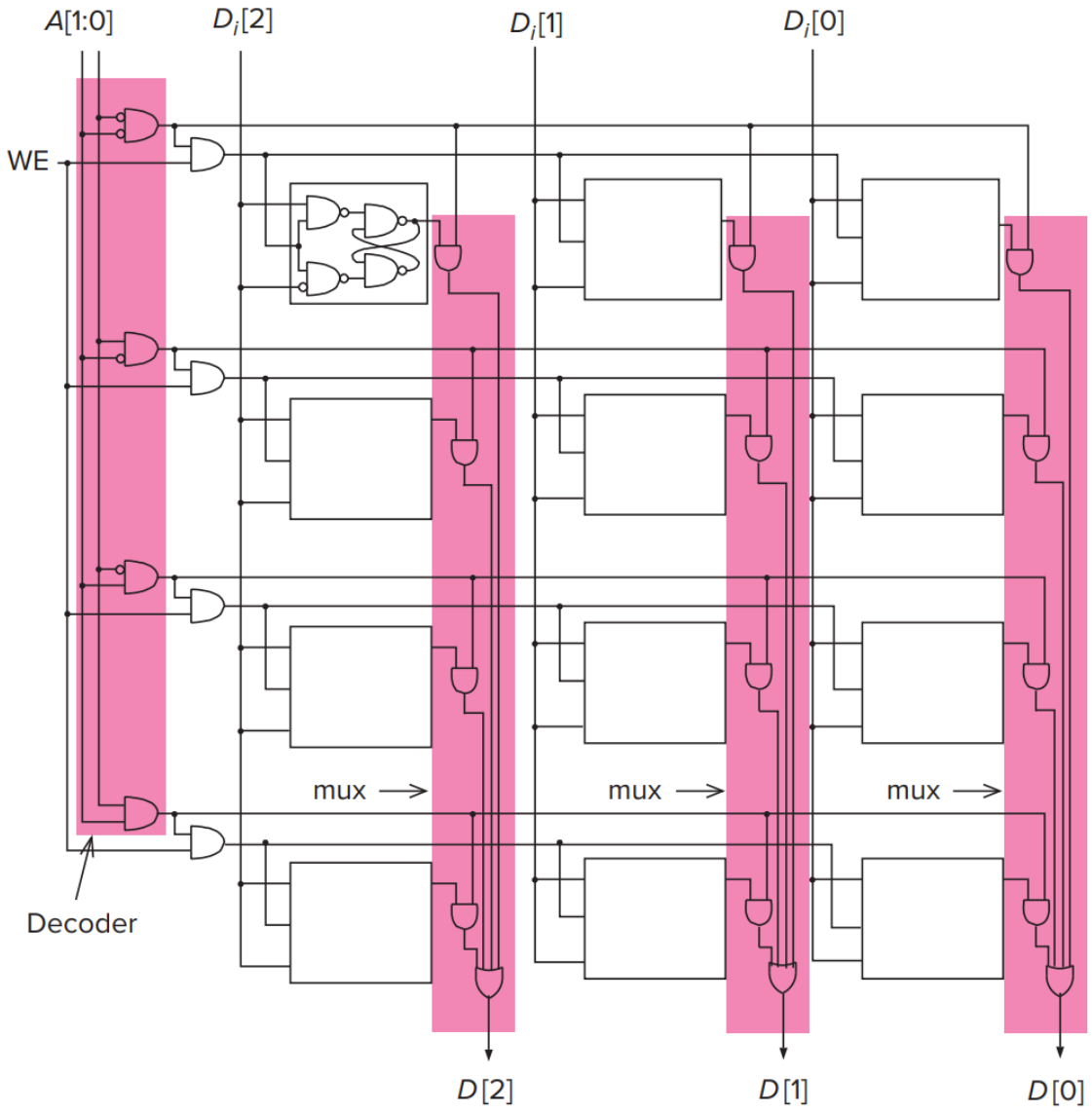


Figure 3.45 Diagram for Exercise 3.40.

e.g. 2^2-by-3-bit memory



其中 A[] 表示地址, D[] 表示数据.

- 从 A[1:0] 中输入地址, 通过 Decoder 对应某一行, 选择出对应的内存位置. 其中每一行为 3 bits, 因此有三个存储器.
- WE=0 时读取数据, 此时每个 bit 中的存储器 (R=S=1, 只能读) 将其中的值读出, 通过 MUX 最后输出得到 D[2:0].
- WE=1 时写入数据, 此时每个 bit 中为 D 锁存器, 直接写入 D 的值, 即 D_i[2:0]. 此时也会有输出.

5 Sequential Logic Circuits 时序逻辑电路

之前的电路均为组合逻辑电路，根据特定的输入得到特定的输出。

时序逻辑电路为根据**当前的输入**以及**当前的储存信息**得到输出的电路。

5.1 State

一个系统的状态为每个时间点对整个系统所有相关的快照。计算机的状态通常为锁存器等储存元件种储存的二进制值，在适时的时间发生转换，产生相应的输出。

e.g.

- 一个贩卖机只卖一种可乐，售价为 1.5 ¥

- 只能投 1￥ 或 0.5￥ 硬币，一次只能投一个
- 投够 1.5￥ 硬币后输出一瓶可乐，多的钱作为找补输出

贩卖机的状态：

1. 没有投
2. 共收到 0.5￥
3. 共收到 1￥
4. 共收到 1.5￥
5. 共收到 2￥

其状态会发生转移：

- 1 收到 0.5￥ 进入 2，收到 1￥ 进入 3
- 2 收到 0.5￥ 进入 3，收到 1￥ 进入 2
- 3 收到 0.5￥ 进入 4，收到 1￥ 进入 5
- 4 无条件跳到 1 并输出一瓶可乐
- 5 无条件跳转到 1 并输出一瓶可乐和 0.5￥ 找补

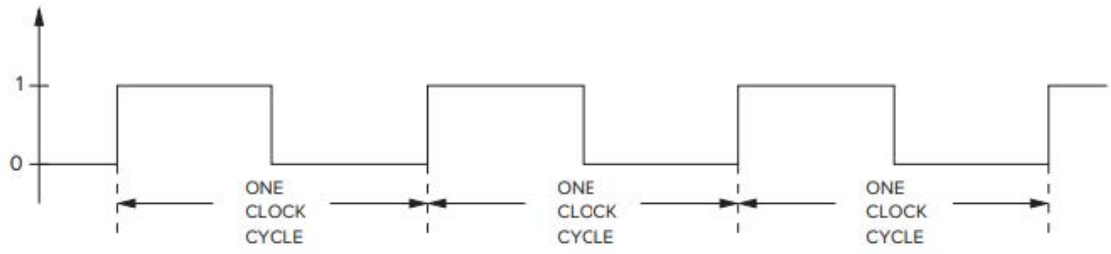
5.1.1 Finite State Machine 有限状态机

有限状态机有以下特点：

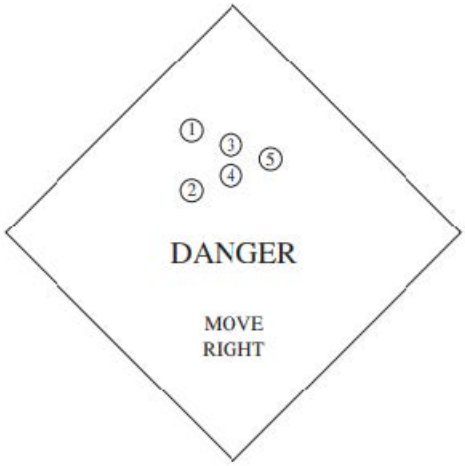
- 有限数量的状态
- 有限数量的外部输入
- 有限数量的外部输出
- 所有状态转换都有显式的规范
- 有显式的规范来决定每个外部输出的值

5.1.2 同步有限状态机

每隔一定时间就根据**当前时间点的输入**以及**当前状态**进行转移。**通过时钟信号控制同步有限状态机。**



e.g. 危险指示灯

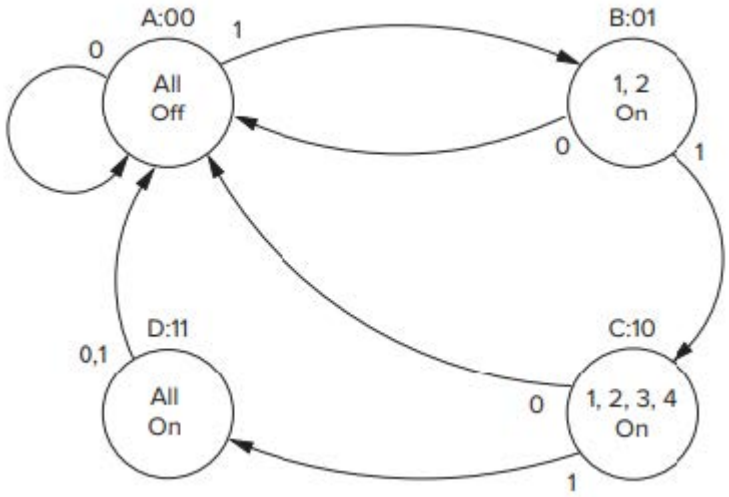


此指示灯在打开时会自动重复：

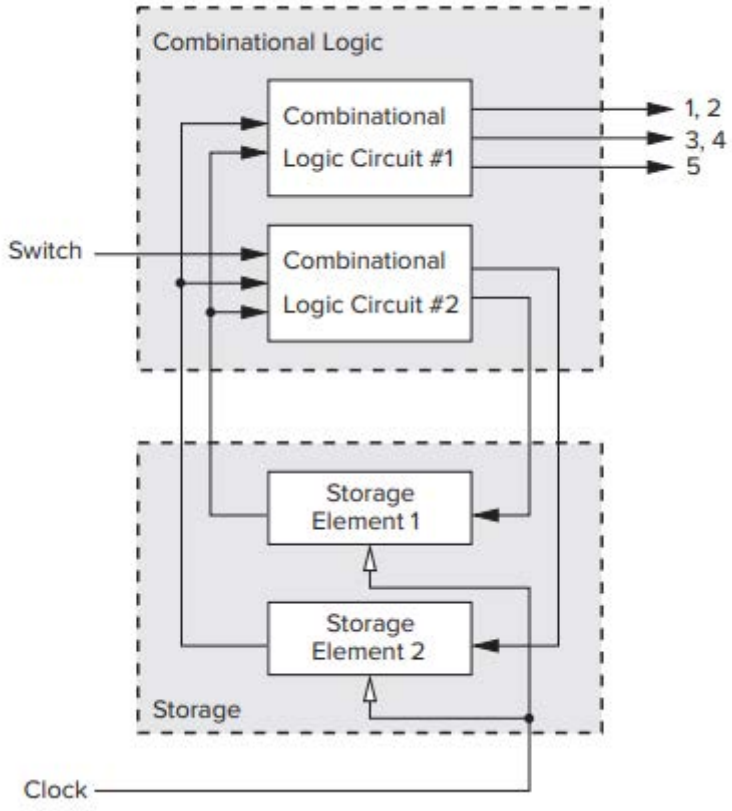
- 灯全灭
- 1，2开启
- 1，2，3，4开启
- 1，2，3，4，5开启

关闭时在现已状态改变的时间点会关闭所有灯

上述四种情况即为四个状态，可以用两位二进制来表示，得到**状态图 (State Diagram)**

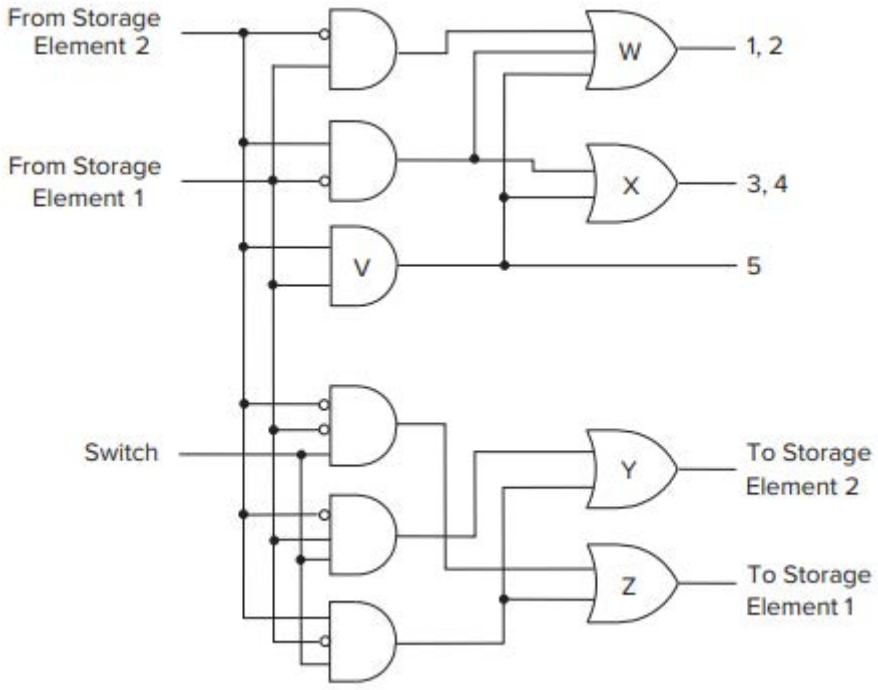


此时状态可以用储存结构进行储存，同时需要组合逻辑电路，根据**当前的状态及输入**决定**当前输出和下一状态**



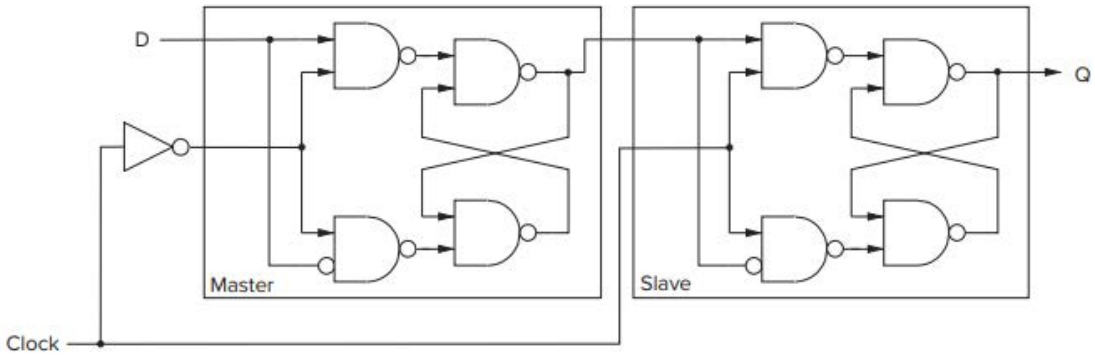
对于组合电路 1，信号灯的亮灭只与当前储存的状态有关，输入值为当前状态，输出值为五个灯是否开启。

对于组合电路 2，输入值为当前状态和开关值输入，输出值为下一个状态的值。



5.1.3 Flip-Flop 存储状态的介质

之前的储存结构只能做到将特定的数据储存，不能根据信号的变换选择是否储存。因此需要通过触发器进行数据储存



其时序图如下：

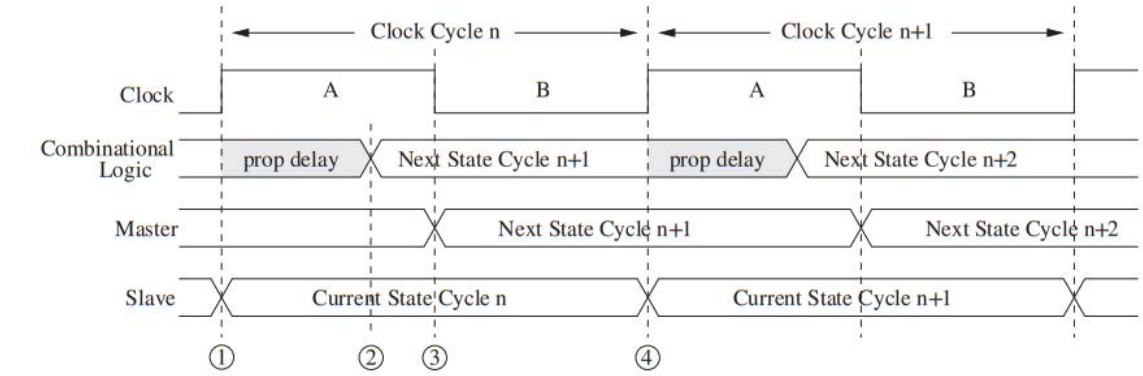


Figure 3.34 Timing diagram for a master/slave flip-flop.

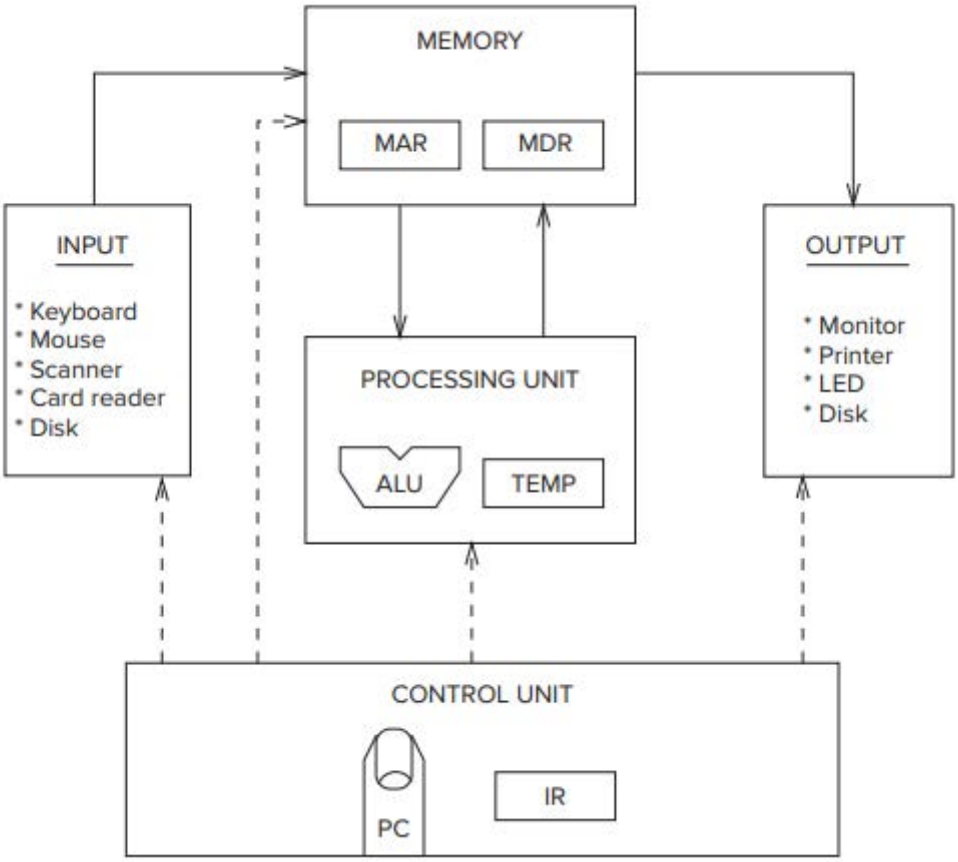
WE = 1, SLAVE 读 MASTER 中的数据

WE = 0, MASTER 读下一个数据

- In the first half A(WE=1), the slave read the value from the master and update the value by combinational circuits(Q) while the master remain the same.
- In the second half B(WE=0), the value updated by combinational circuit is stored in master while the slave remain the same.
- Timing diagram
    1. the slave load the value from the master.
    2. the combinational logic finishes its update/calculation(proper delay). So it enters the next cycle.
    3. the master load the updated value, so it also enters the next value.
    4. repeat 1

Chapter 4 Von Neumann Model

1 Basic Components



1.1 Memory

即内存，可用于储存数据（通常为需要执行的命令），通过地址信号来访问某一具体位置，读取内容或写入数据。

LC-3 中，**地址为16 位二进制数**，每个地址位置储存 **16bits。**从地址 **x0000 到 xffff**

Register 寄存器

和访问内存相关的有两个重要的寄存器：

- **MAR (memory's address register)**

  存放当前访问的内存地址

- **MDR (memory's data register)**

  将从内存中读取的数据放在此处；将写入的内存的数据放在此处

1.2 Processing Unit 处理单元

包含能进行计算的单元和临时存储单元：

- **计算单元**

  包括 **ALU** (Arithmetic and Logic Unit)，能进行各种算术（加减法）以及逻辑运算（与或）

- **临时储存单元**

  包括通用寄存器，作为储存指令所需数据的位置

  LC-3 有八个通用寄存器，为 R0-R7，可用三位二进制位对其进行编码

1.3 Input/Output

在 LC-3 中，输入为键盘输入，输出为屏幕显示内容。

1.4 Control Unit 控制单元

包括所有控制计算机进行执行和处理的结构。

- **PC (program counter)**

  PC 寄存器中储存当前指令执行完成后下一条需要被执行的指令储存在内存中的地址。

- **CC (condition codes)**

  CC 编码中储存 nzp 3bits数据，某些特定的指令在执行完后其结果会影响 nzp 的值。

  n z p 分别对应 positive zero negative，特定指令执行完后会根据据其结果的正负或零与否设置对应位的 1，而其他位为 0。例如计算结果是正数时 p 为 1，z n为0

  带加号的指令会根据其结果影响 CC 的值

2 LC-3 Instruction

2.1 Instruction Processing

LC-3 中有三种类型的指令：

- operates：进行算术运算和逻辑运算。
- data movement：将数据在 processing unit，memory，I/O 之间移动的指令。
- control：用于改变指令的执行顺序，类似分支跳转指令 (像C语言里面的goto)

每一条指令的执行会经历以下六个阶段（不一定全都经历）：

- **Fetch**

  将 PC 的值输入到 MAR 中并**将 PC 加一**，从内存中读出要执行的指令并写入 MDR 中，将 MDR 中的指令传输到 IR 中 (**instruction register，指令寄存器**)

- **Decode**

  此阶段会根据opcode分析出各个部分的控制信号，从而控制整个微架构的运行

- **Evaluate Address**

  计算需要被这条指令访问的内存位置的地址。由于不是所有指令都需要访问内存，所以此阶段不是必要的。

- **Fetch Oprands**

  即读取操作数阶段，例如从内存中读取数据，或者从通用寄存器中读取数据等

  LD 会在此阶段访问内存

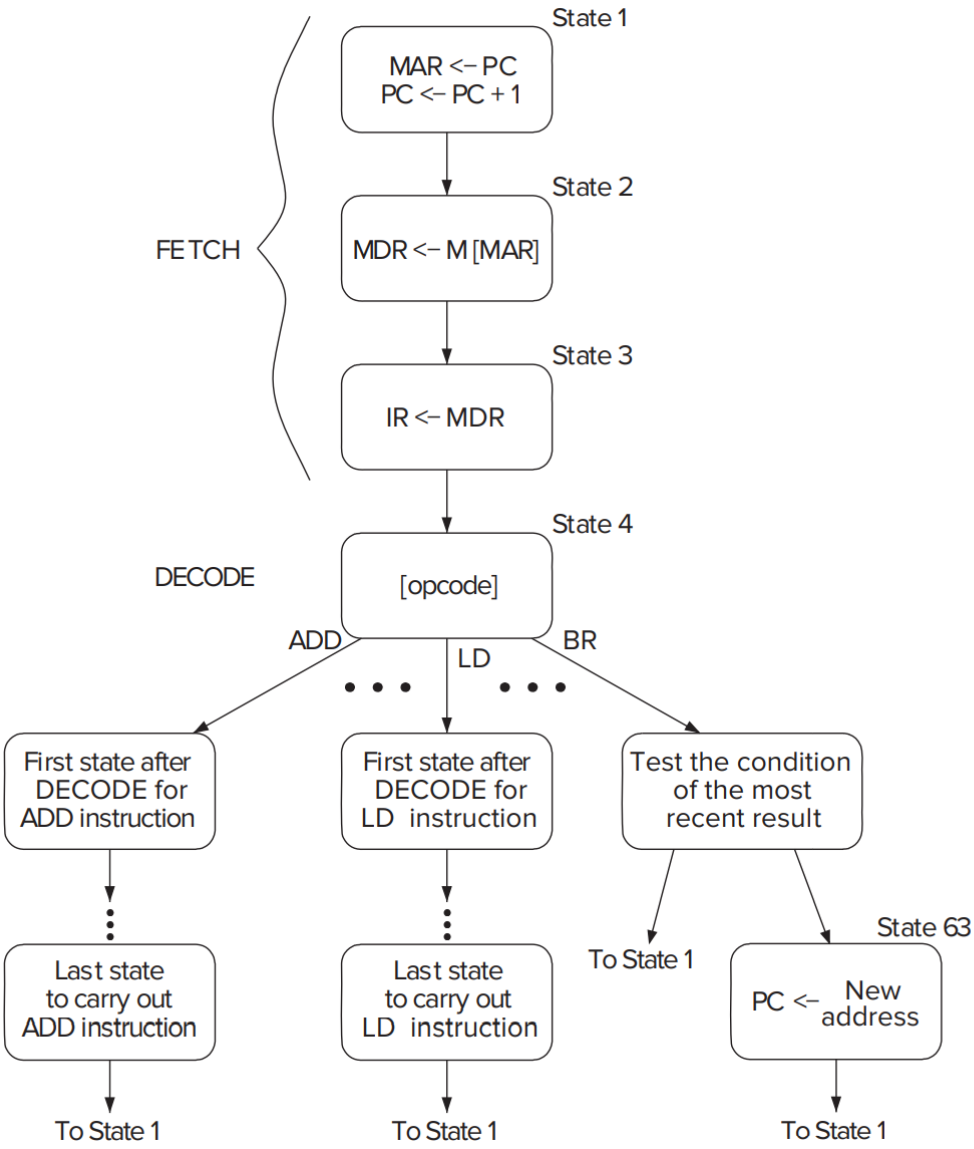
- **Execute**

  执行算术逻辑运算，如加法，按位与等

  跳转类指令都是在此阶段修改 PC 值

- **Store Result**

  将计算得到的结果存储到其设计的目标中，如目的寄存器，内存等。



Chapter 5 LC-3

1 ISA (Instruction Set Architecture 指令集架构)

1.1 Memory organization

- **Address Space**

  地址为与每个内存位置关联的唯一标识符。**Address Space为唯一可识别位置的总数**

  How many locations can be addressed ? 2^{16}

- **Addressability**

  每一个内存位置存储的比特的数量即为 **Addressability**

  how many bits per locaiton ? $16\ bits$

1.2 Register set 寄存器

temperary storage, accessed in a single machine cycle

- 8个通用寄存器 R0-R7
- 不可直接寻址但由指令使用：PC、CC

1.3 Opcode 操作代码

15 opcodes

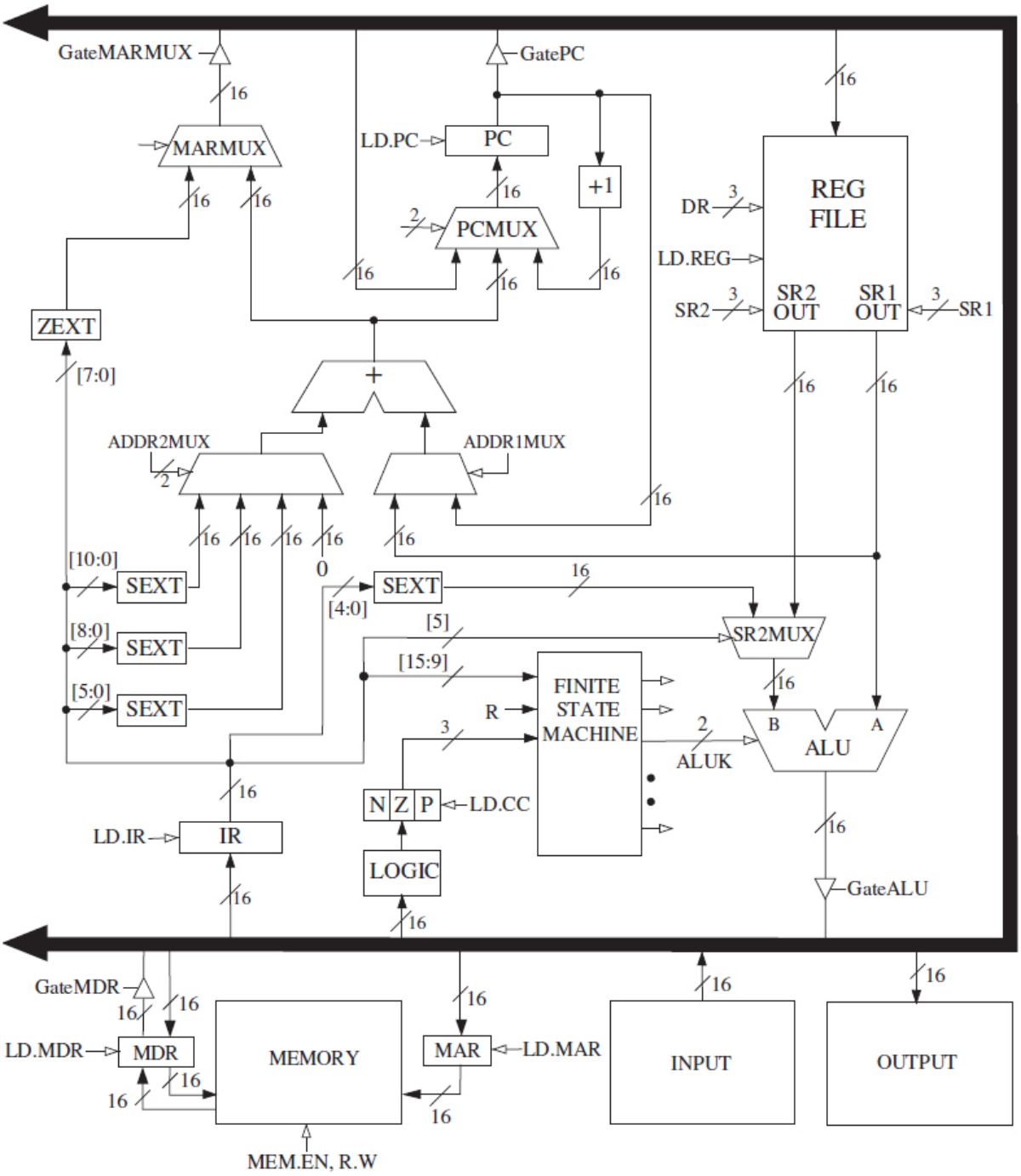
- **Operate instructions**: ADD, AND, NOT
- **Data movement instructions**: LD, LDI, LDR, ST, STI, STR, LEA
- **Control instructions**: BR, JSR/JSRR, JMP, RTI, TRAP
- **可以改变CC值的**：ADD, AND, NOT, LD, LDR, LDI, LEA

1.4 Addressing Modes 寻址模式

How is the location of an operand specified?

- **non-memory addresses**: immediate, register
- **memory addresses**: PC-relative, indirect, base+offset

2 Data path 数据通路组成



2.1 Global bus

A special set of wires that carry a 16-bit signal to many components

- 仅在启用时踩在总线上放置信号
- 任意时刻仅能启用一个信号
- 控制单元决定哪个信号驱动总线
- 任意数量的组件都可以读取总线
- 寄存器仅在控制单元允许写入的时候读取总线数据

2.2 Memory

Control and data registers for memory and I/O devices

memory: MAR, MDR (also control signal for read/write)

2.3 ALU 算术逻辑单元

Accepts inputs from register file and from sign-extended bits from IF (immediate field 立即字). Output goes to bus. used by condition code logic, register file, memory

2.4 PC and PCMUX

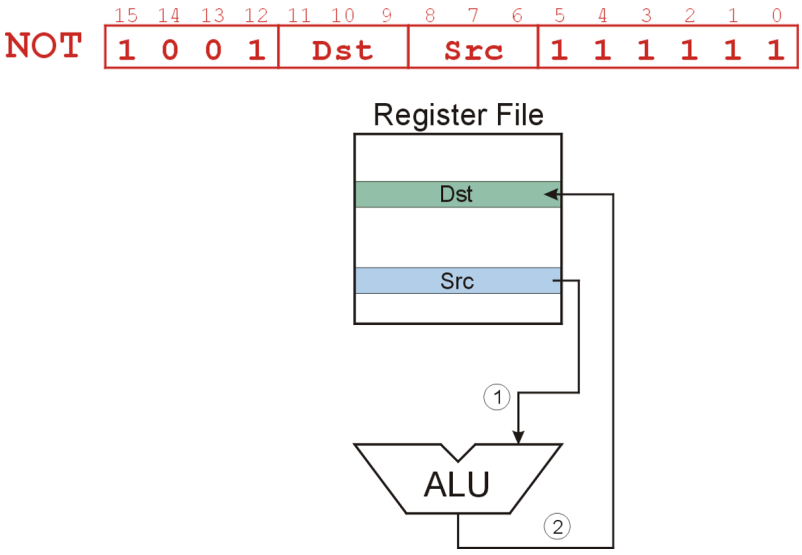
Three inputs to PC, controlled by PCMUX

- **PC+1 – FETCH stage**
- **Address adder – BR, JMP**
- **bus – TRAP (discussed later)**

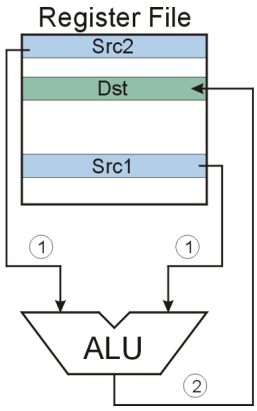
3 Data Path 数据通路

3.1 NOT, ADD, AND, LEA

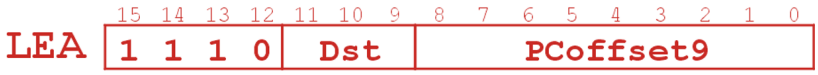
- NOT

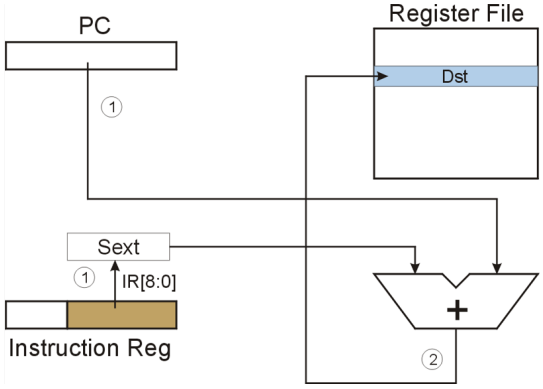


- ADD & AND



- LEA



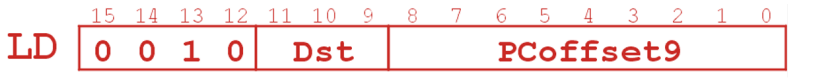


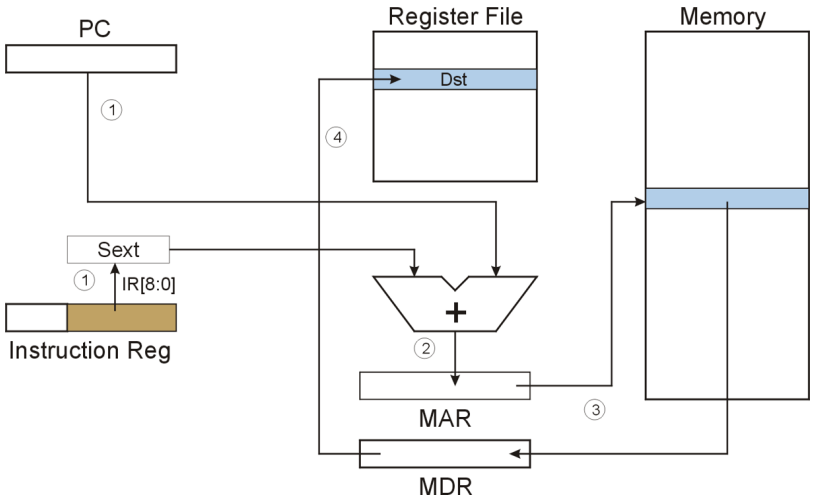
3.2 LD ST

- LD

```
x3002: 0010 001 1 1111 1100
       LD   R1     #-4
```

将地址为 x3002 + x0001 - x0004 = x2FFF 处的数据储存在 R1 寄存器中

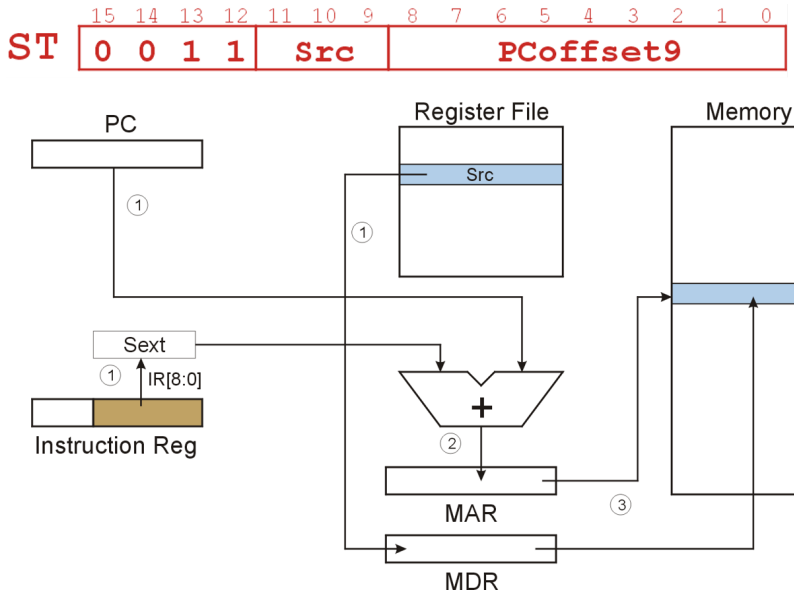




- ST

```
x3001: 0011 001 0 0000 0011
       ST   R1     x3
```

将 R1 寄存器的值储存在 x3001 + x0001 + x3 = x3005 处

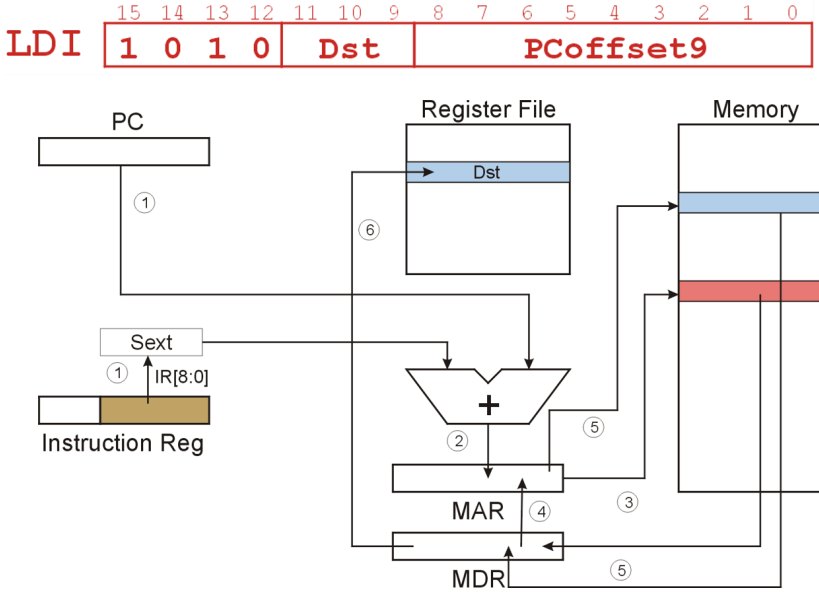


3.3 LDI STI

- LDI

```
x3001: 1010 001 0 0000 0101
       LDI  R1      x5
```

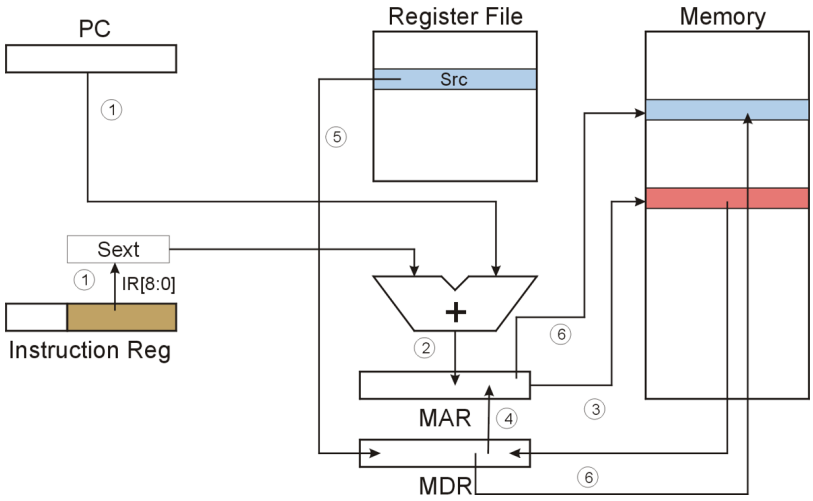
地址为 x3001 + x0001 + x0005 = x3007 处的数据为 xAAD3，将 xAAD3 处的数据写在 R1 中



- STI

```
x3008: 1011 001 1 1111 1100
       STI  R1     #-4
```

地址为 x3008 + x0001 -x0004 = x3005 处的数据为 xAAD3，将 R1 中的数据写到 xAAD3 处

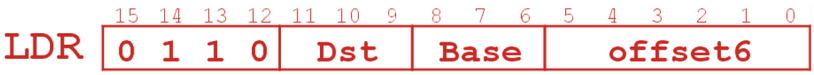


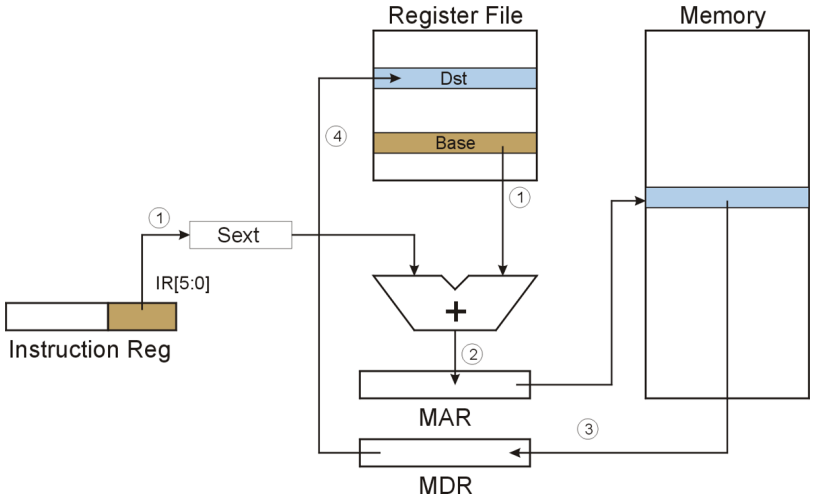
3.4 LDR STR

- LDR

```
x3000: 0110 001 010 00 1100
       LDR  R1  R2    xC
```

若现在 R2 的值为 x3010，则将地址为 x3010 + xC = x301C 处的数据读到 R1 中。

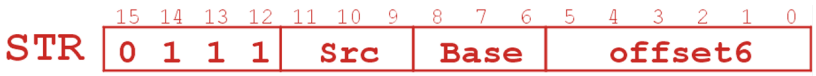


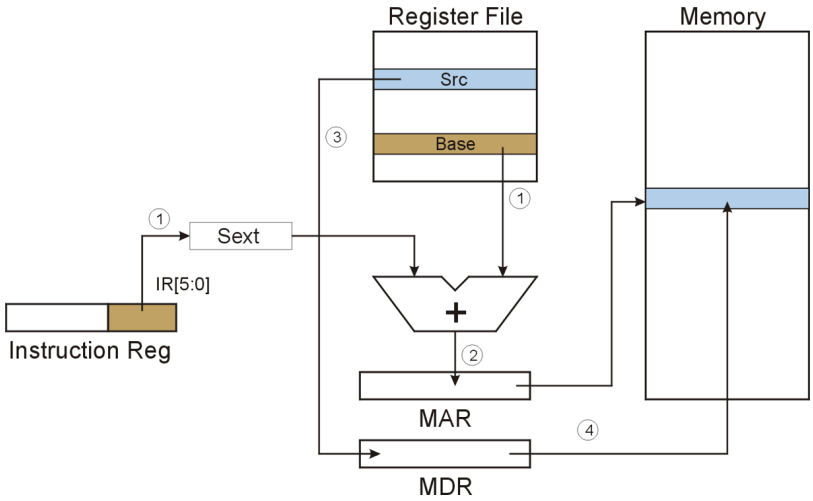


- STR

```
x3000: 0111 001 010 00 1100
       STR  R1  R2    xC
```

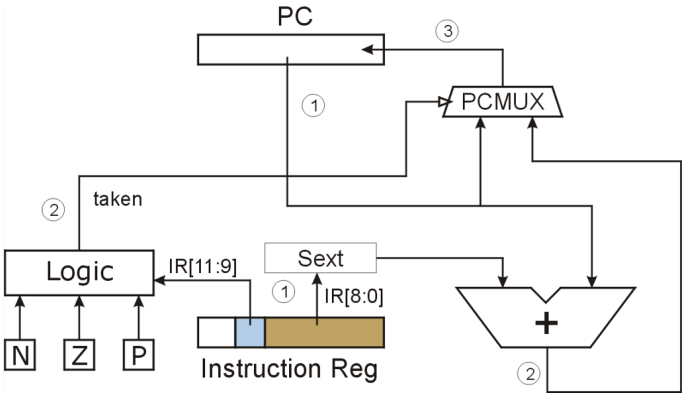
若现在 R2 的值为 x3010，则将 R1 中的数据写到地址为 x3010 + xC = x301C 处。



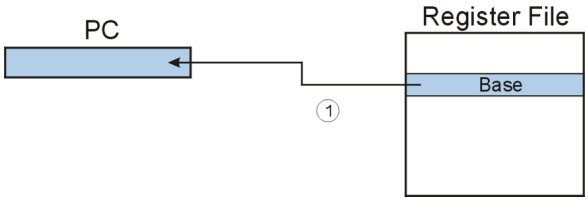


3.5 BR, JMP

- BR



- JMP



Chapter 6 Programming

The larger tasks are systematically broken down into smaller ones, which is called *systematic decomposition.*
There are basically 3 constructs for doing this.

- **sequential construct 顺序构造**: carry out the first subtask completey, *then* go to the second. Never go back.

- **conditional construct 条件构造**: the task consists of doing one of 2 subtasks but not both, depending on some condition.
- **iterative construct 迭代构造**: the task consists of doing a subtask a number of times as long as some condition is true.

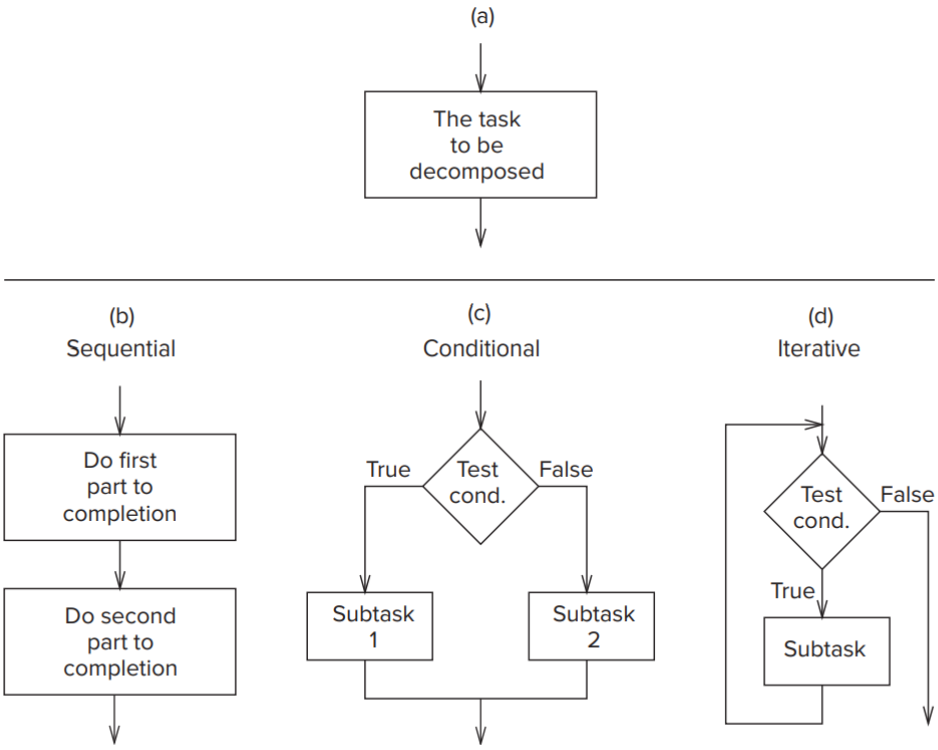


Figure 6.1 The basic constructs of structured programming.

Chapter 7 Chapter 7 Assembly Language

- From ISA to language.
- Assembly language is a low-level language.
- High-level languages tend to be ISA independent.
- Low-level languages are very much ISA dependent. In fact, it is usually the case that each ISA has only one assembly language.
- The translation program is called an assembler and the translation process is called assembly.

汇编语言转为机器码的过程叫**汇编**，而不是**编译**。编译是对高级语言（如 C）而言。

1 Instructions

每条汇编指令的格式如下所示：

```
Label Opcode Operands
```

其中 Opcode 和 Operands 是必须的，Label 是可选的。
汇编指令的 Opcode 和 Operands 都是符号化表示的，例如 ADD 可表示加法指令的 opcode。
operands 可以从以下位置获取：

- 寄存器，可以用 R1，R2 等来表示

- 内存，可以使用 label 进行表示
- 立即数，立即数可以使用十进制，二进制，十六进制进行表示，

  # 十进制，b 二进制，x 十六进制

Chapter 8 Data Structures

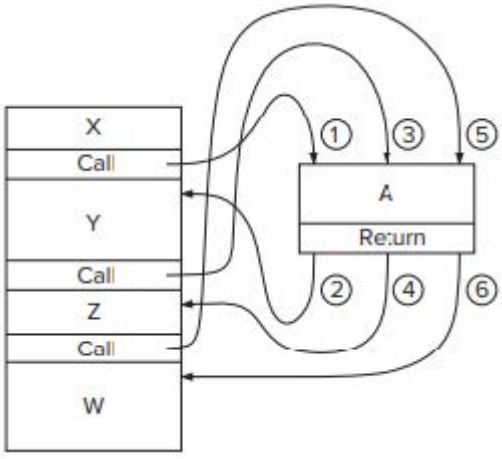
1 Subroutines 子程序

subroutines = procedure = function

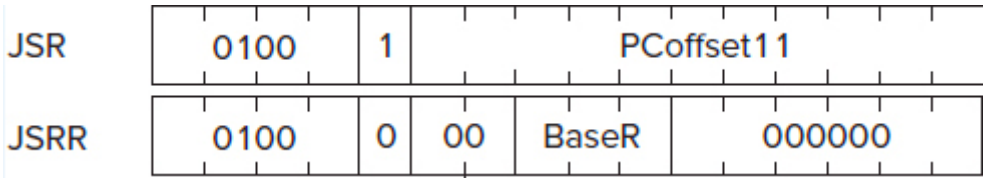
所以这个概念就是类似于C语言里的函数。

1.1 The Call/Return Mechanism 调用/返回机制

子程序执行顺序结构如下：

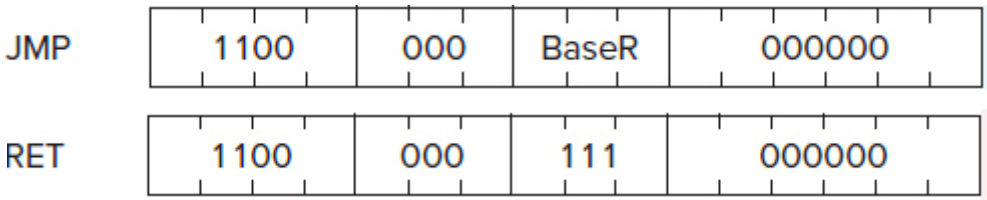


为实现函数的调用，需要有**跳转到函数对应指令**以及执行完函数指令后**回到调用函数指令的下一条指令**的能力，此机制由 JSR/JSRR 和 JMP/RET 实现：




```
JSR LABEL
JSRR R1
```

- JSR 跳转到 LABEL 所指的位置
- JSRR 跳转到 R1 所存地址对应的位置。
- 将当前的 PC 值（JSR/JSRR 的下一条指令的地址）储存在 R7 中




```
JMP R1
RET
```

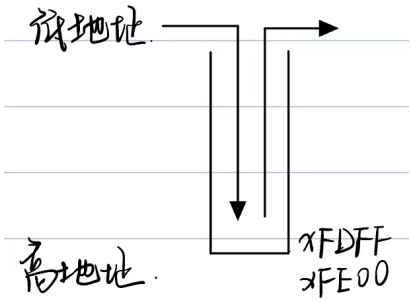
- JMP 将 R1 值作为下一条指令的地址
- RET 等价于 JUM R7

2 Stack

Stack (栈) 是一种 ADT (Abtract Data Type 抽象数据类型)，其由特定的对象，以及对应的具体操作构成。栈满足 LIFO 原则，即 Last in First out。

栈有以下两种独有的操作：

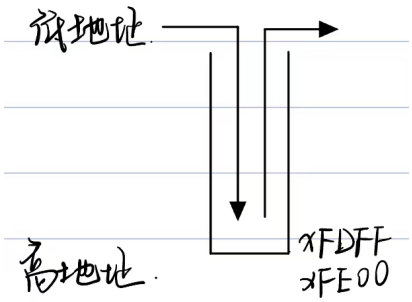
- Push : 将一个数据压入栈顶
- Pop : 从栈中获得最近加入的数据并删除



2.1 Implementation in Memory 内存中的实现

- 栈顶：最新加入栈的数据的位置
- 栈底：栈中最旧数据所在位置
- 栈指针：用 **R6** 在记录栈顶

在 LC-3 中，栈底是高位地址，栈会向更小的地址成长。根据权限等级的不同，LC-3 中的栈分为用户栈和系统栈，系统栈底为 x2FFF，用户栈底为 xFDFF（xFE00 到 xFFFF的储存空间给 IO 外设）结构如下：



具体实现：

- 在程序开始之前设置 R6 的值为栈底

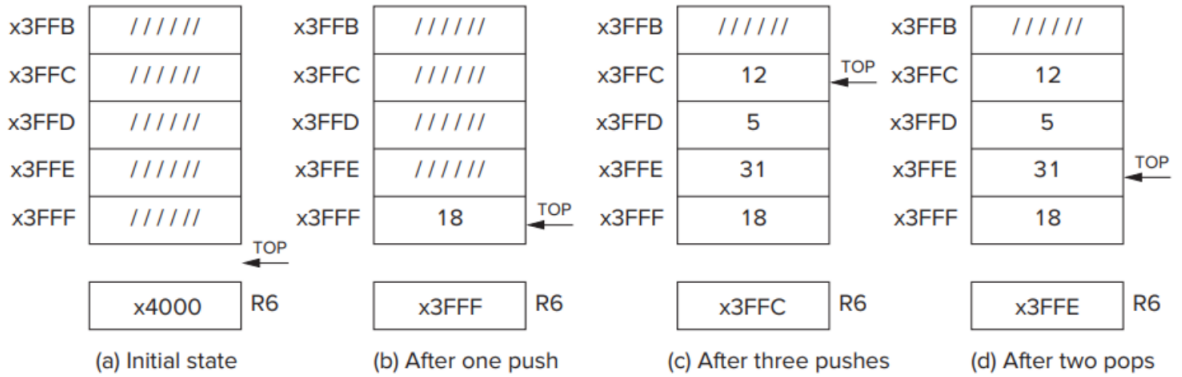
```
LD R6, stack_ptr;R6存储栈底地址
stack_ptr .FILL xFE00
```

- Push 操作

```
ADD R6, R6, #-1 ;将指针指向下一个内存空间
STR R0, R6, #0  ;将R0的值压入栈中
```

- Pop 操作

```
LDR R0, R6, #0  ;将栈顶的值压入R0
ADD R6, R6, #1  ;更新栈顶，旧的值在新的值压入栈顶后会被覆盖
```



2.2 Overflow/Underflow 上溢/下溢

- **Overflow 只发生在 Push 过程中**

  当栈中的数过多时，数据会将内存中的代码覆盖产生错误，因此需要设置一个栈指针的最小值。当栈指针指向最小值表示栈已满，此时再 Push 数据就会产生 Overflow

- **Underflow 只发生在 Pop 过程中**

  当栈为空时，调用 Pop 则会 Underflow

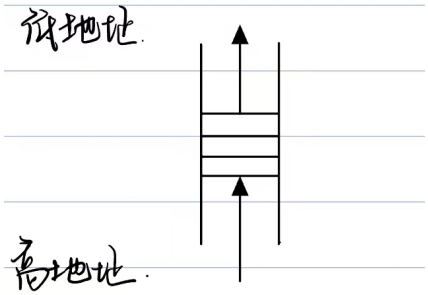
若要检测 Overflow 和 Underflow 只需要再 Push 和 Pop 时 检测顶指针的值，判断其是否为满或为空。

3 Queue

遵循 FIFO 原则，First in first out。每次只能取出最早进入的数据。

队列有两种操作：

- Enqueue 入队：将一个数据加入队列放在队列的最后面
- Dequeue 出队：将队列的第一个数据取出并删除



3.1 Implement in memory

队列的实现需要两个指针 REAR 和 FRONT，数据遵循从低地址向高地址增长

- REAR：队列中最后一个数据的地址
- FRONT：队列中最前面一个数据再前面一个的地址

例如一个使用内存空间 x8001 到 x8005 来构成一个队列，其结构如下：

```
x8001: x0000 <- FRONT
x8002: x0001
x8003: x0002 <- REAR
x8004: x0003
x8005: x0004
```

此时队列中储存的数据即为 x0002 和 x0001。如果队列为空，则 FRONT 和 REAR 初始均指向 x8000。出队和入队的操作，假设 REAR 存在 R4 中，FRONT 存在 R3 中：

- Enqueue，将 R0 的值加入队尾

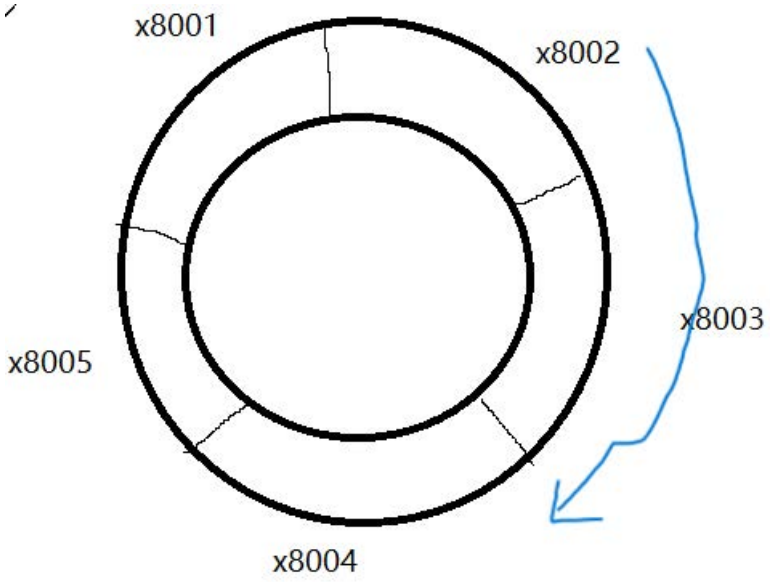
```
ADD R4, R4, #1
STR R0, R4, #0
```

- Dequeue，将队首的值存入 R0

```
ADD R3, R3, #1
LDR R0, R3, #0
```

3.2 Wrap-Around 循环队列

由于队列操作时两个指针的移动方向相同，为避免影响其他区域，使用 Wrap-around 循环队列思路，其核心思想为指针若移动后超过分配给地址空间下界时，则将其赋值为队列空间的上界



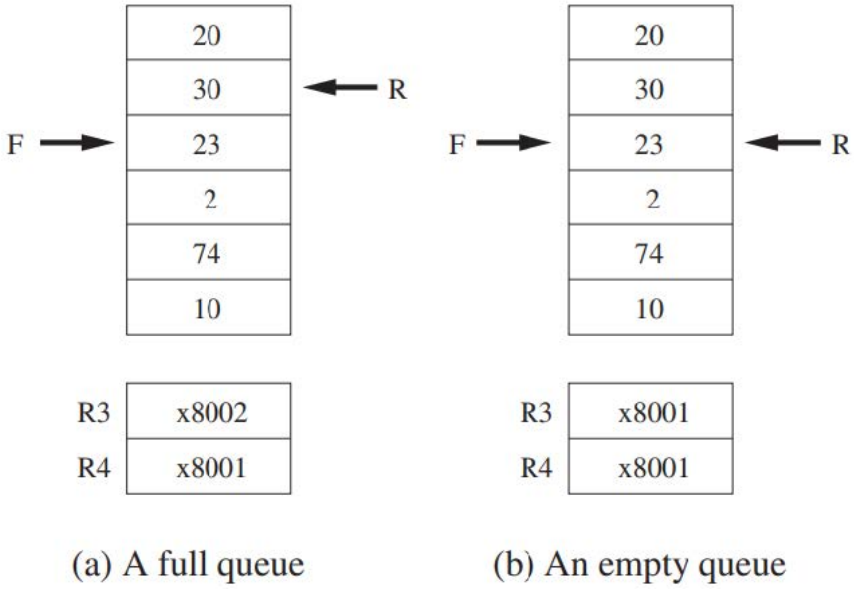
- Enqueue，将 R0 的数据加入队尾

```
lower_bound .FILL x8001
upper_bound .FILL x8005
    ADD R4, R4, #1
    LD R1, upper_bound
    NOT R1, R1
    ADD R1, R1, #1
    ADD R1, R4, R1
    BRnz else
    LD R4, lower_bound
else:
    STR R0, R4, #0
```

- Dequeue，将队首的数据存入 R0

```
lower_bound .FILL x8001
upper_bound .FILL x8005
    ADD R3, R3, #1
    LD R1, upper_bound
    NOT R1, R1
    ADD R1, R1, #1
    ADD R1, R3, R1
    BRnz else
    LD R3 lower_bound
else:
    STR R0, R3, #0
```

循环队列的满队列和空队列如下：



(a) A full queue　　(b) An empty queue

对于一个能储存 n 个数据的队列，当队列满时仅能储存 n-1 个数据，若将 n 个位置全部占满则对导致 REAR 和 FRONT 指向同一个位置，造成满队列和空队列的混淆

4 Recursion 递归

求第i个斐波那契数的C语言程序如下所示

```
int fib(int n) {
    if (n == 0 || n == 1) {
    return n;
    }
    return fib(n - 1) + fib(n - 2);
}
```

当调用fib(3) 来求第3个斐波那契数时，其函数调用路径如下所示

```
fib(3) {
    调用fib(2) {
        调用fib(1) {
            返回值为1
```

```
        }
        调用fib(0) {
            返回值为0
        }
        返回值为1
    }
    调用fib(1) {
        返回值为1
    }
    返回值为2
}
```

观察递归程序执行的过程，可以发现一个情况：当前未执行完成的函数中，最后被调用的函数最先执行完成并返回。
这和栈的执行逻辑相似，所以递归的核心为将函数的状态暂存到栈中，可以采用callee save的方式进行，即为函数刚刚开始执行时先将当前寄存器的状态保存到栈中，在函数即将结束时从栈中恢复数据。
fib 函数的LC-3汇编代码如下所示

```
.orig x3000
    ld r6, stack_ptr
    and r1, r1, #0
    add r1, r1, #3
    jsr fib
    add r2, r0, #0 ; x3004
    halt
    stack_ptr .fill xfe00
; 假设函数的返回值存在R0中，传入函数的参数为R1，栈指针为R6，R7存着函数的返回地址
fib:
    ; 将会进行修改的临时寄存器的值放在栈中进行保护
    add r6, r6, #-1
    str r1, r6, #0
    add r6, r6, #-1
    str r7, r6, #0
    add r6, r6, #-1
    str r2, r6, #0
    and r0, r0, #0
    ; 判断当前输入是否为0或者1
    add r2, r1, #0
    BRz fib_end
    add r0, r0, #1
    add r2, r2, #-1
    BRz fib_end
    ; 递归调用
    add r1, r1, #-1
    jsr fib ; fib(n - 1)
    add r2, r0, #0 ; x3015
    add r1, r1, #-1
    jsr fib ; fib(n - 2)
    add r0, r2, r0 ; x3018
    fib_end:
    ; 将被保护的值从栈中取出
    ldr r2, r6, #0
    add r6, r6, #1
    ldr r7, r6, #0
    add r6, r6, #1
```

```
    ldr r1, r6, #0
    add r6, r6, #1
    ; 返回
    ret
.end
```

Chapter 9 I/O

1 Privilege 权限

即执行程序的权力。在 LC-3 中有两种权限等级：

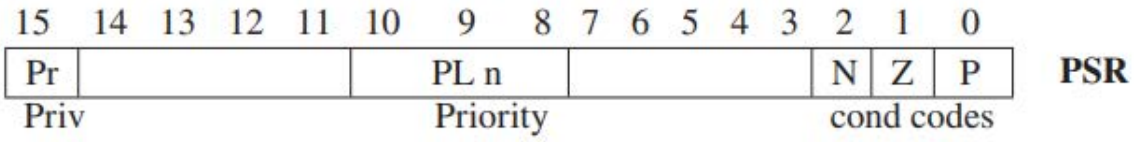
- supervisor privilege mode 管理员权限
- unprivileged mode 普通权限

2 Priority 优先级

即一个程序执行的紧迫程度，由于优先级的存在，使得更高优先级的程序去中断（interrupt）优先级更低的程序执行。可分为 0-7 八档

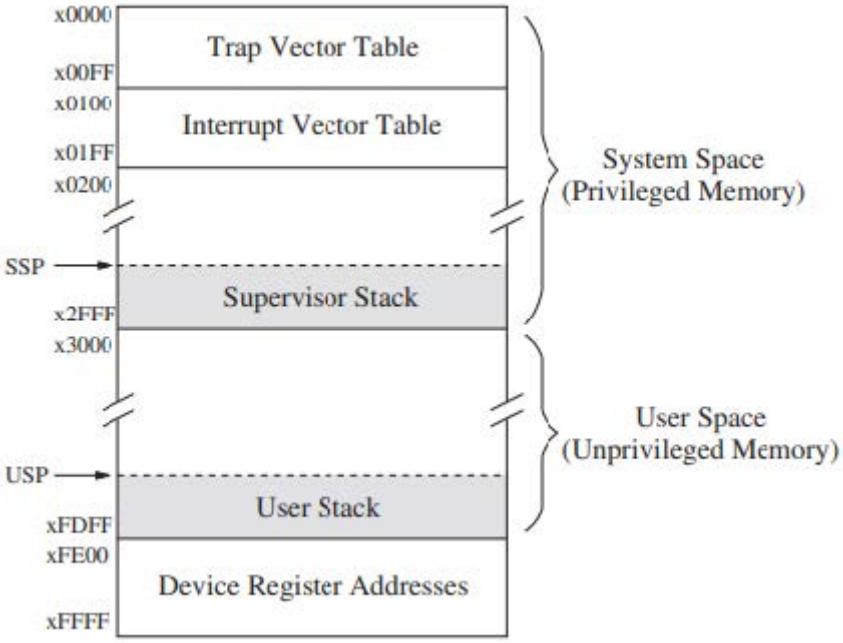
- 用户编写的程序优先级为 0
- 键盘中断程序优先级为 4

2.1 Processor State Register (PSR) 处理器状态寄存器



- PSR[15] 储存优先级
  - PSR[15] = 0 表示当前为管理员权限
  - PSR[15] = 1 表示当前为普通权限
- PSR[10:8] 表示当前执行程序的优先级
- PSR[2:0] 储存当前的 CC

3 Organization of Memory



- x0000 到 x2fff 为系统空间，Privileged
- x3000 到 xFDFF 为用户空间，Unprivileged
- xFE00 到 xffff 为 IO 设备区域，Privileged

4 Basic Characteristic of I/O

4.1 Memory-Mapped I/O vs Special I/O Instructions 内存映射和I/O指令

- **Special I/O instruction**： 一些计算机会设计一些专门的指令来访问IO设备。
- **Memory-Mapped I/O**： 大多数计算机会将 IO 外设映射到内存的特定地址，然后使用和访问内存相同的访存指令来修改 IO 设备寄存器的值，从而控制 IO 设备。

4.2 Asynchronous vs Synchronous 异步和同步

- **Asynchronous 异步 （Main）**

  计算机中有多个时钟信号即为异步。由于 IO 设备相对于 CPU 为慢速设备，因此通常要为其设置更慢的时钟来控制，为保证两种时钟能够协同工作，需要使用类似握手的机制，例如使用 flag 信号来判断 IO 设备是否完成工作。完成工作则将 flag 设为 1，通过 CPU 检测 flag 为 1 来确定 IO 工作完成。这种 flag 信号的机制也被称作 Synchronization，可以理解为异步设备的同步化机制。

- **Synchronous 同步**

  计算机中只有一个时钟信号为同步，如需要访问慢速设备，则设置多个周期来保证 IO 正常工作。

4.3 Interrupt-Driven vs. Polling 中断驱动与轮询

- **Interrupt-Driven 中断操作**

  当 IO 需要相关操作时就会发送一个中断信号，CPU 在检测到中断信号后就会终端当前程序的执行，跳转到需要处理的程序，处理完后继续主程序的执行。

- **Polling 轮询**

即重复查看 IO 外设的状态，使用循环的方式重复查询，当查询到外设的相关 flag 被设置时就跳出循环

在 LC-3 中需要考虑的外设为键盘输入和显示器输出

5 Keyboard

和键盘相关的寄存器有 KBSR 和 KBDR 两种，使用内存映射的方式，将 xFE00 赋值给 KBSR，将 xFE02 赋值给 KBDR，其结构如下所示：

Figure 9.3 Keyboard device registers.

KBSR[15] 为 flag，KBDR[7:0] 储存键盘输入的字符 ASCII 码，当键盘产生输入时自动将输入字符的ASCII码导入到 KBDR[7:0] 中，在完成后将 KBSR[15] 设置为 1。当检测到 LC-3 读取 KBDR[7:0] 的值时自动将 KSBR[15] 设置为 0：

- KBSR[15] == 1：表示当前 KBDR[7:0] 有新的未被读取的数据。
- KBSR[15] == 0：表示当前 KBDR[7:0] 中没有新的数据。

使用轮询的方式，重复判断 KBSR[15] 的值，直到其为 1 时跳出循环，读出 KBDR[7:0] 的数据。

6 Monitor

显示器的两个寄存器为 DSR 和 DDR，分别映射到内存的 xFE04 和 FE06

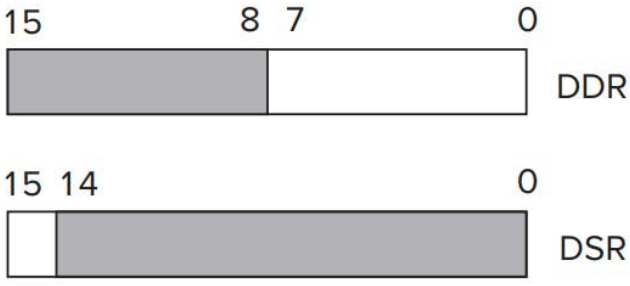


Figure 9.5 Monitor device registers.

DSR[15] 为 flag，DDR[7:0] 储存要显示的数据，将要显示的数据写入 DDR[7:0] 时会自动将 DSR[15] 设为 0，当字符成功被显示的时候，即将 DSR[15] 重新设为 1

- DSR[15] == 0，显示器正在显示一个字符，不能进行显示新字符的操作。
- DSR[15] == 1，当前的显示器处于空闲状态，可以进行显示字符的操作。

使用轮询的方式，不停判断 DSR[15] 的值，直到它为 1 时跳出循环，将要显示的字符的ASCII码 写入到DDR[7:0]中。

IO设备映射在内存中的微架构

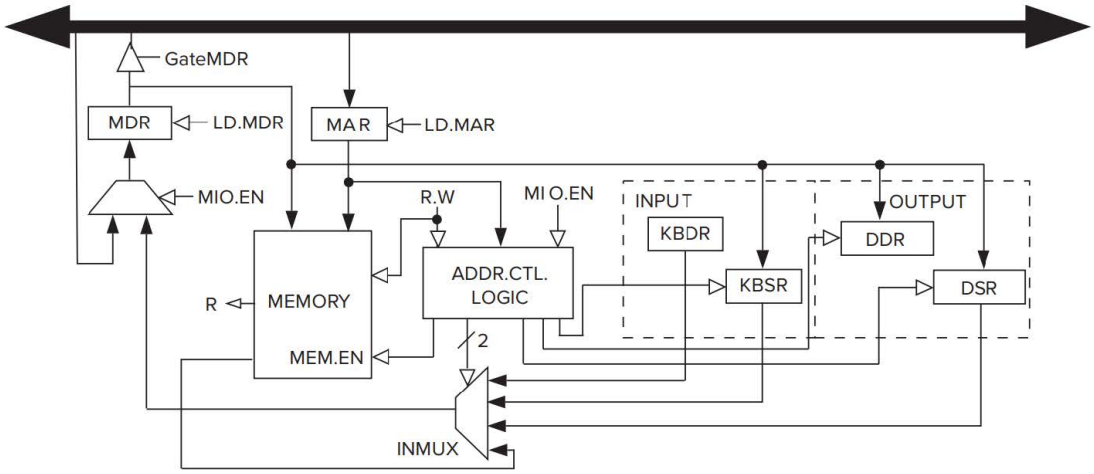


Figure 9.8 Relevant data path implementation of memory-mapped I/O.

7 TRAP

为更好地管理硬件资源以及和 IO 相关的寄存器等资源，需要操作系统来进行管理。

操作系统有supervisor privileged，可以随意访问映射到内存的IO寄存器，而跑在用户模式的代码不能访问到 IO 相关的寄存器。而为了处理 IO，操作系统给用户程序留了接口，通过接口即可使用操作系统的程序来访问 IO 相关的特权寄存器。

此操作系统的接口在LC-3中即为trap 机制。trap 机制由以下几个部分构成：

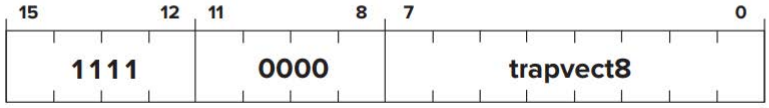
- 操作系统函数的程序，其分布在地址x2fff 之前的操作系统内存区域中。
- 一个关于各种程序的起始地址的表，其范围是x0000 - x00ff ，则可以使用8个比特进行索引 trap 指令，其结构如下所示。
- 执行完操作系统程序后能回到用户原本代码的能力。

TRAP System Call

Assembler Format

TRAP trapvector8

Encoding



根据指令结构可知，trap 指令中编码了 8 位的 trapvector，表示从起始地址第几个位置找到要执行代码的地址并执行，例如 trap x25 表示从内存 x0025 处找到一个地址，其为halt 程序的起始地址，此时就能找到对应的 halt 程序。

为让操作系统程序执行完后能回到原本的程序，则需要保存调用 trap 的程序的 PC 和 PSR。这里用栈来进行保存。

TRAP 指令的执行过程

```
TEMP = PSR
if (PSR[15] == 1) {
// 从普通模式转为特权模式，需要将R6的值从用户指针转变为栈指针
Saved_USP=R6         // 保存用户栈指针
R6 = Saved_SSP       // 将系统栈指针导入R6
PSR[15] = 0          // 将权限等级设为特权模式
}
PUSH TEMP，PC in system stack     // 先存TEMP(原PSR)，再存PC
PC = mem[ZEXT(trapvect8)]        // 将PC设为查询trap程序起始地址表中查到的地址
```

由于操作系统程序也可能会使用trap调用其他程序，所以在进行模式转换以及栈指针转换时，需要进行是否发生权限转换的检查。

注意上述行为中存储的PC是trap指令的下一条指令的地址，因为在trap指令的fetch阶段已经将PC进行加一了。

由于trap 指令返回原来的指令时和简单的JSR 调用函数的逻辑不同，所以需要专门的返回指令RTI 。其行为如下所示

```
if (PSR[15] == 1) {
//报告一个异常，因为在非特权模式不能使用该指令
} else {
    PC = mem[R6]
    R6++
    TEMP = mem[R6]
    R6++                  // 这四条指令为将原先的PC和PSR的值从栈中pop出来
    PSR = TEMP
    if (PSR[15] == 1) {
// 此时程序会从特权模式回到普通模式，则要切换栈指针
        Saved_SSP = R6
        R6 = Saved_USP  // 所以就算调用trap时R6中的值不是用户栈指针，由于trap的保护机制，
也能保证R6的值不被改变
    }
}
```

上述两种行为中 Saved_SSP 和 Saved_USR 都是特殊的寄存器。

8 Interrupt 中断

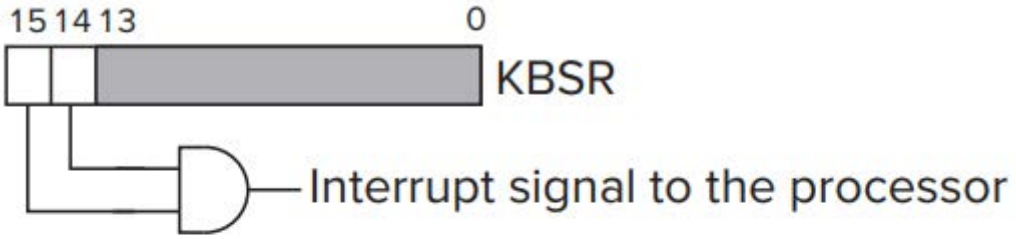
由于采用轮询的方式处理 IO 的话会消耗 CPU 很多时间循环和读取 KBSR 的值，造成时间的浪费，而中断只在有 IO 请求需要处理时才跳转到对应的程序处理，节约了轮询花费的时间。

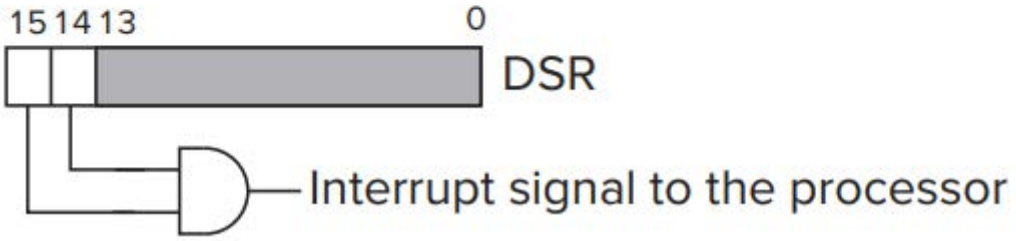
8.1 Process

使用中断控制 IO 需要两个部分

8.1.1 允许 IO 设备产生中断的机制

- 中断使能信号机制

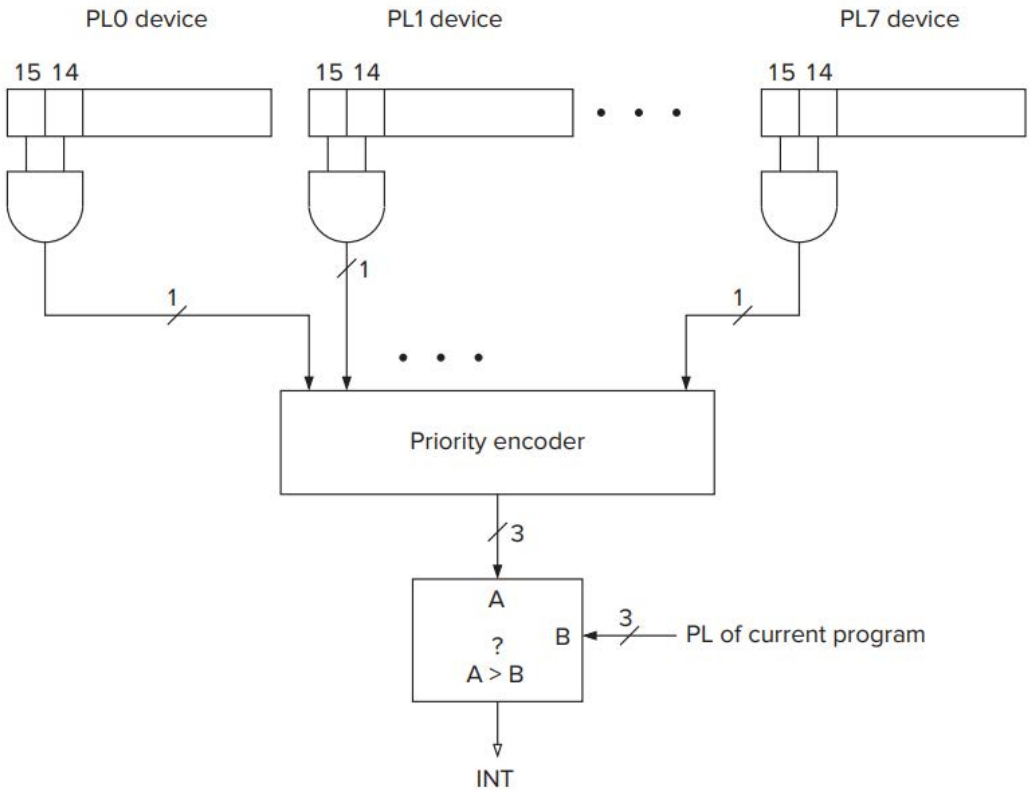




当 KBSR[14] 为 1 时，键盘才能才生中断

当 DSR[14] 为 1 时，显示器才能产生中断

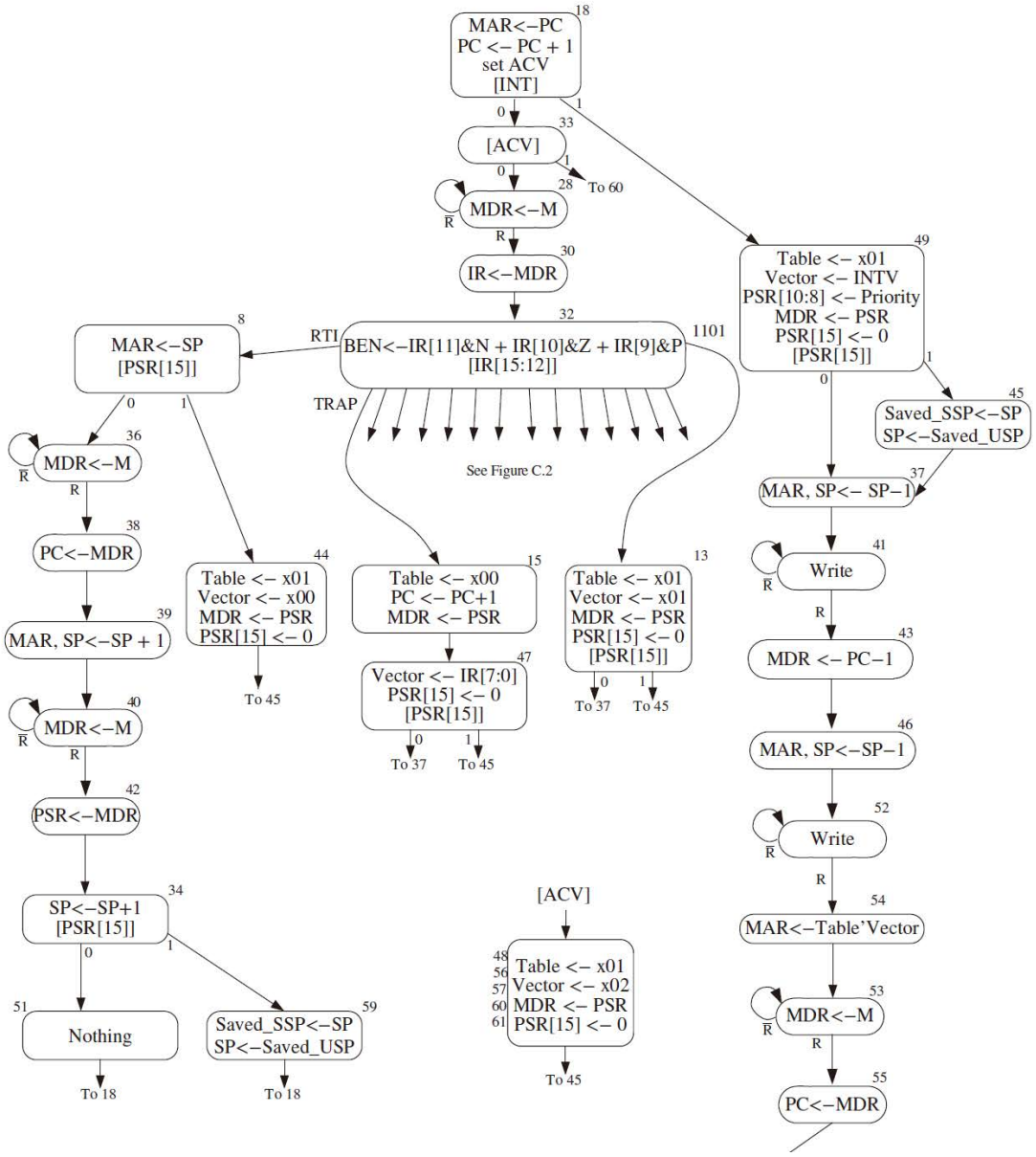
- 优先级机制



先将所有的中断信号传到一个编码器中，得到该中断对应的优先级，随后和现在的优先级进行比较，若中断程序优先级高于当前程序的优先级，则中断信号为1.

8.1.2 处理中断请求的机制

中断类似 TRAP，也有一个中断向量表，其存储在内存的 0x100 - 0x1ff 处，中断也有一个 8 位的中断向量，其表示该中断对应的处理程序在中断向量表的哪个位置，例如中断向量为 0x80 ，则中断处理程序的起始地址存在内存的 0x0180 的位置，进入中断的过程。



中断程序执行完后，使用 RTI 离开中断处理程序。

不是所有中断都由 IO 设备产生，当计算机运行到一些特定的情况时，如果这个情况会产生中断，且对应的优先级高于当前程序的优先级，那就会产生中断。

Exception (异常)也是 CPU 内部一种会产生中断的机制。其为当 CPU 的运行产生相应的错误后就会产生一个中断，随后进入对应的中断处理程序。在 LC-3 中可能有的异常只有一下三种

- 在普通模式下使用 RTI
- 非法 opcode(1101)
- Access Control Violation (ACV) Exception，访问了权限不够的内存位置。