## 第二章：Bit、数据类型及其运算

- 计算机的本质（不过是一个电子设备而已）
- 计算机如何表示信息

## 第三章：数字逻辑

- 如何表示0，1
- 门电路、组合电路、
- 寄存器、内存、时序电路
- 有限状态机

- 第四章：冯诺依曼计算机模型
  - 模型简介
  - 举例：小型计算机LC-3
  - 程序如何运行、指令、程序跳转、停机

- 第五章：LC-3详解
  - 指令集合
  - 内存寻址
  - LC-3机器指令编程

- 第六章：如何求解问题
  - 结构化程序设计
  - 调试

- 第七章：汇编语言
  - 从机器指令到汇编
  - 汇编程序如何编译至机器指令程序
  - 可执行文件格式、多目标文件、链接

## 第八章：输入输出设备

- 硬件结构、同步/异步IO
- 键盘如何输入信息、显示器输出信息
- 中断IO

## 第九章：TRAP、子程序

- 系统调用TRAP及实现
- 子程序原理及实现

## 第十章：栈、全面总结LC-3

- 动机、内存实现
- 中断驱动IO、嵌套
- 数据转换
- 栈的使用举例

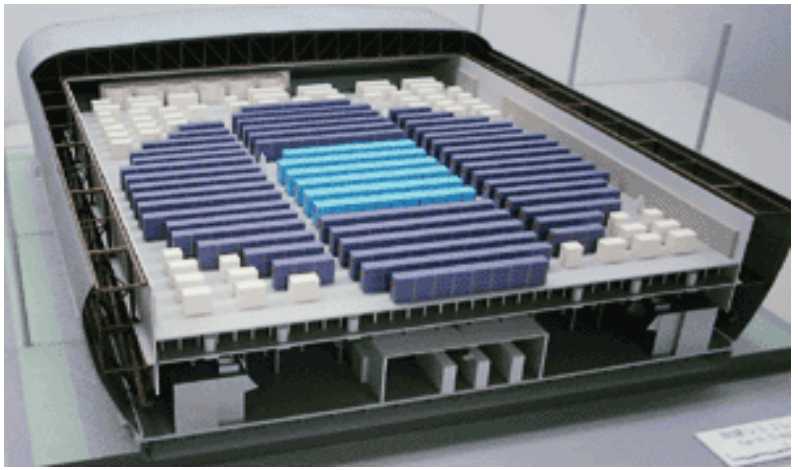
# Chapter-1

# 一个重要思想

- 计算设备具有通用性。对一个计算设备而言，只要具备足够的资源（内存，时间...）,它可以解决任何问题。


=

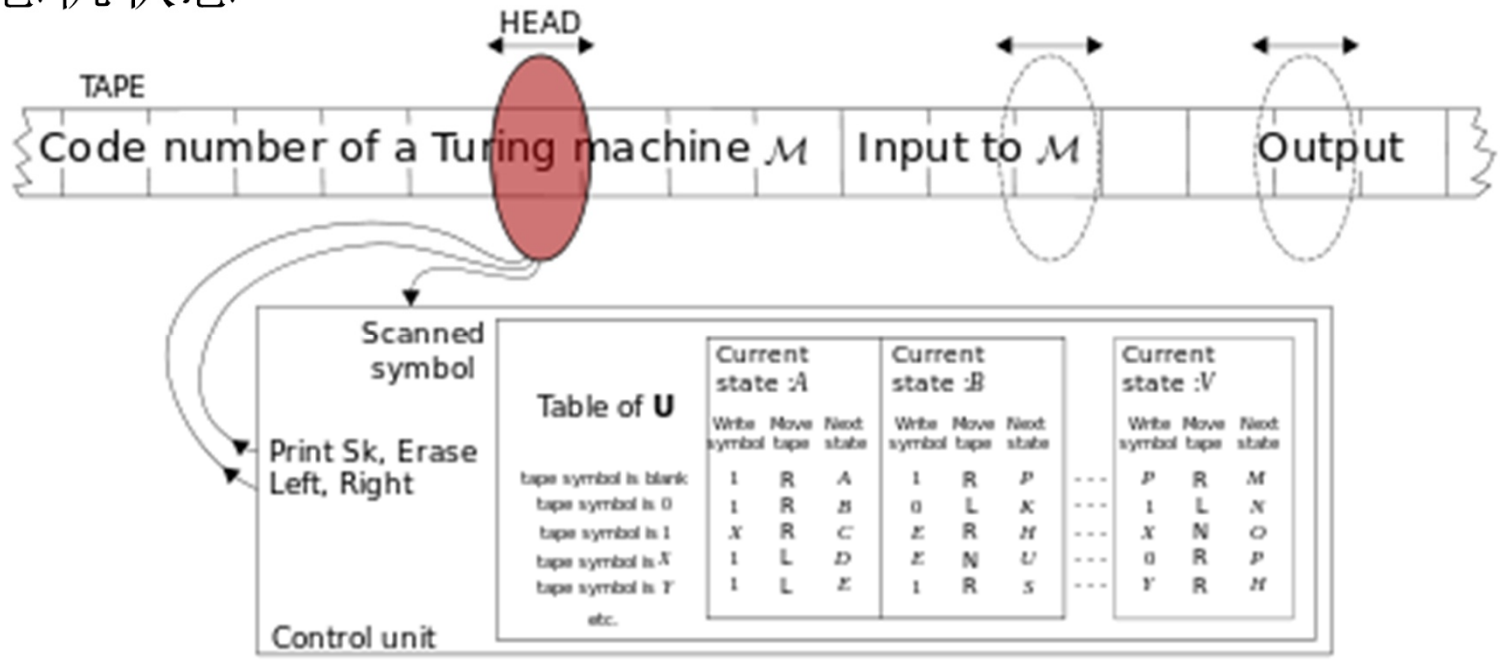

=



Supercomputers

# 计算机的数学模型 —— 图灵机

- Alan Turing：1937年提出
  - 输入输出带
  - 有限状态机状态

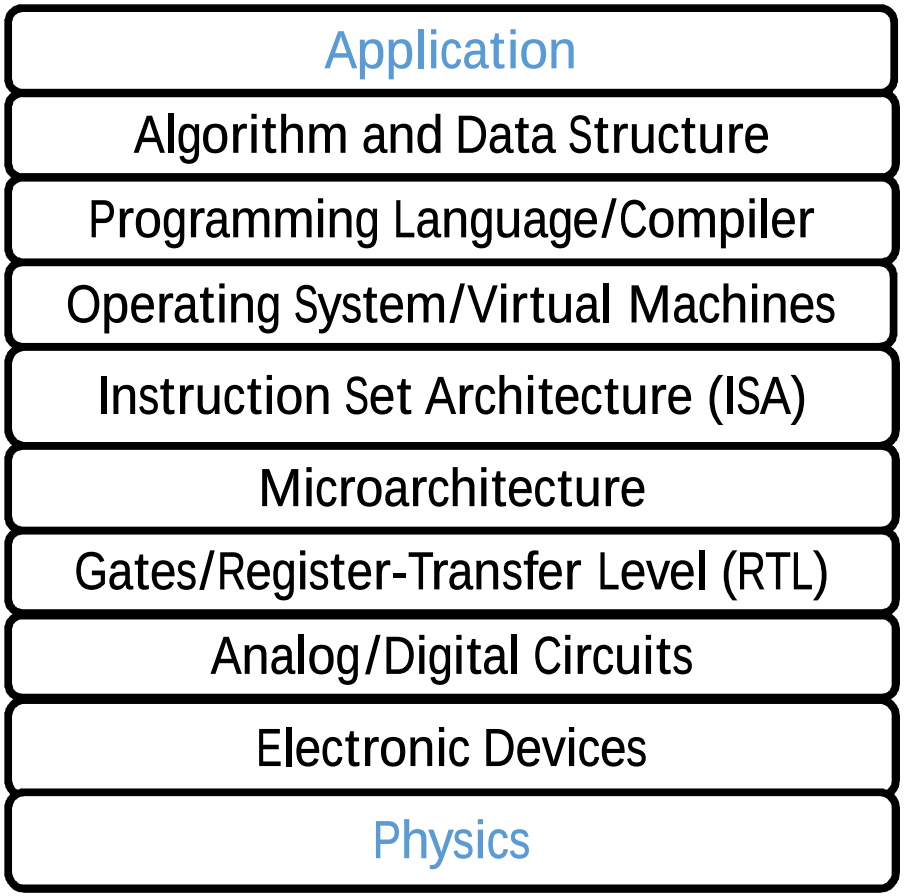


# 第二个重要思想：分层化

- Abstraction helps us Manage Complexity ： 计算机体系结构的层次化结构

- 使用计算机求解问题，问题虽然使用自然语言描述，到计算机使用一系列电子设备以高电压、低电压方式解决该问题，中间经过了一系列的变换，每一个变换抽象成一个层。

# Abstraction Layers in Modern Systems




**Related Courses**

算法基础/数据结构
程序设计/编译技术
操作系统/虚拟机
计算机组成
数字逻辑
集成电路
微电子

# Chapter-2

# 计算是采用二进制的数字系统

数字系统

- 变量具有有限个状态，
- 用有限个符号表示

二进制数字系统:

- 最简单的数字系统
  - has two states: 0 and 1


- 二进制系统最基本的信息单元：位 *bit*.
- 用多个位来表示超过两个状态的数值。
  - 两位（ two bits）二进制可以表示4个状态的数值:
    00, 01, 10, 11
  - 三位（ three bits）二进制可以表示8个状态的数值:
    000, 001, 010, 011, 100, 101, 110, 111
  - *n bits 可以表示$2^n$ 个状态的数值.*

# 3 常用定点数表示方法

- 原码表示法
- 补码表示法
- 反码表示法

## 1 符号的处理

### 1）无符号数

只能表示正整数.($0<=N<=2^n-1$) n:数的位数

### 2） 符号数:

它的最高位被用来表示该数的符号位, 不表示数值位。

在计算机中一个数的数值部分和符号都要用0、1编码。通常, 用数的最高位（MSB—Most Significant Bit）表示数的正负

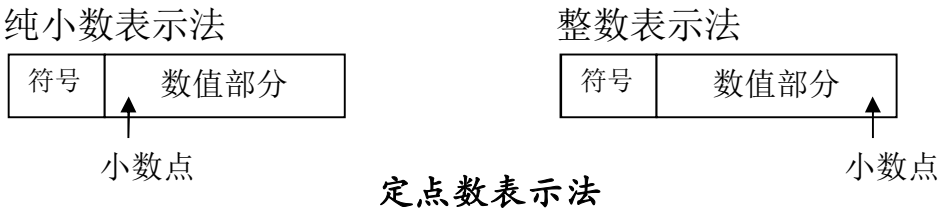
MSB = 0, 表示正数, 如＋1011表示为01011;

MSB = 1, 表示负数, 如－1011表示为11011;

# 2 小数点：定点与浮点表示法

## 定点数表示法

- 在计算机中，图示的小数点“．”实际上是不表示出来的，是事先约定好固定在那里的。对一台计算机来说，一旦确定了一种小数点的位置，整个系统就不再改变。
- 计算机中采用的定点数表示法通常把小数点固定在数值部分的最高位之前，或把小数点固定在数值部分的最后。前者用来表示纯小数，后者用于表示整数。
- 只能处理定点数的计算机称为定点计算机。在这种计算机中机器指令访问的所有操作数都是定点数。




定点数表示法

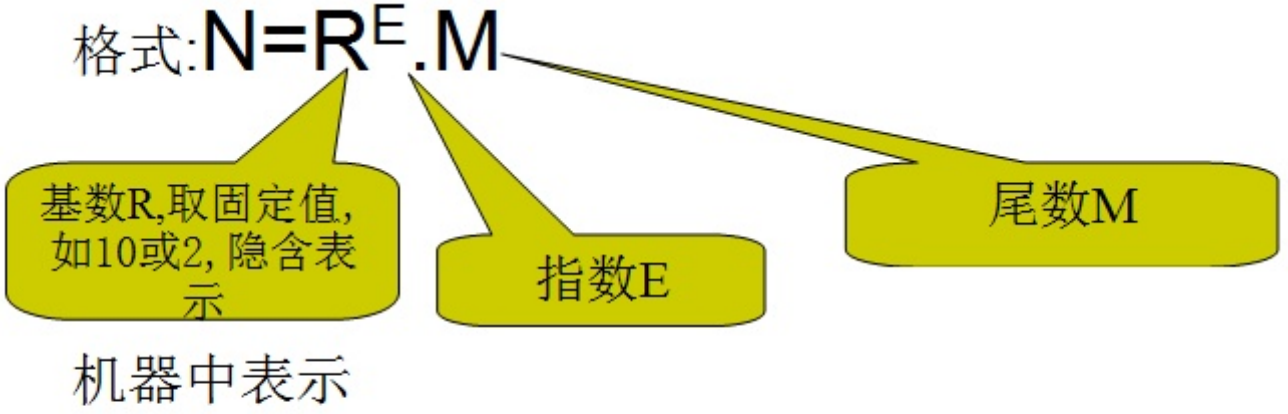
定点数要选择合适的比例因子，确保初始数据、中间结果和最终结果都在定点数的表示范围之内，否则就会产生“溢出”。

# 2.3 浮点数表示法

- 浮点表示法
  (来源于科学计数法)

  电子质量(克)： $9\times10^{-28}=0.9\times10^{-27}$
  太阳质量(克)： $2\times10^{33}=0.2\times10^{34}$

- 小数点的位置可按需浮动,这就是浮点数。



# 5 二进制逻辑运算

一个二进制数位可以用来表示一个二值逻辑型的数据，但逻辑型数据并不存在进位关系。

这里的与、或、非逻辑可以用与门、或门、非门电路实现。

| X | Y | X与Y | X或Y | X非 | X异或Y |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 | 0 |

# Chapter-3

# 基本逻辑门

- 如何利用电子开关实现基本的逻辑操作: AND, OR, NOT.

- 逻辑值与模拟电压:
  - 用不同范围的模拟电压来表示‘0’和’1’

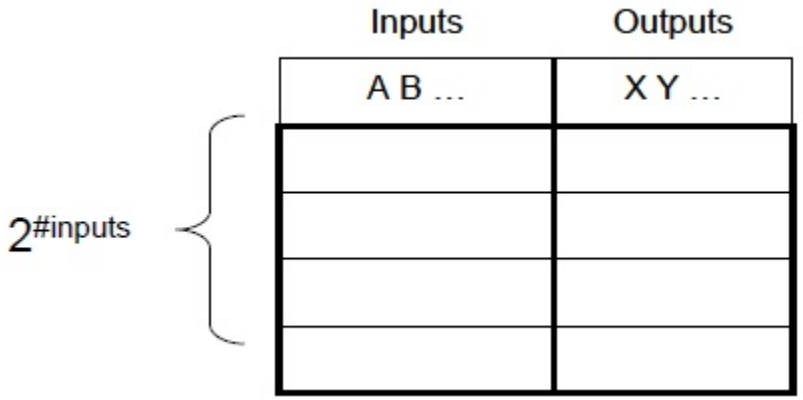

  - 模拟电压范围取决于制造晶体的工艺
  - 通常代表 “1”的高电压有: +5V, +3.3V, +2.9V,+1.8V,+1.35V,+1.0V
    - 教材教学使用 +2.9V

# 真值表：Truth Table

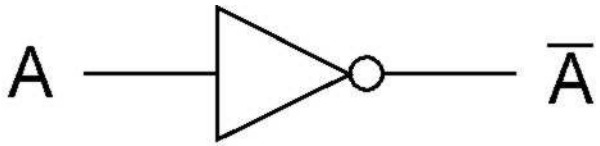
逻辑函数的一种最基本的表达方法

用列表的方式列出所有输入和输出的对应值

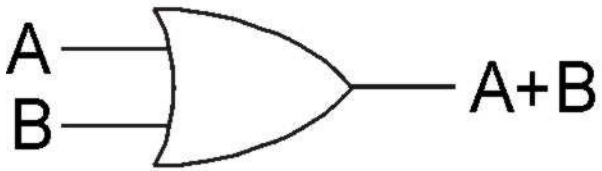
真值表的行数怎么确定?



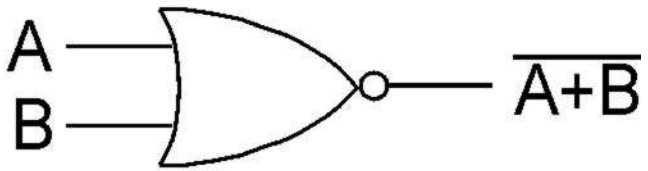
# 基本逻辑门符号



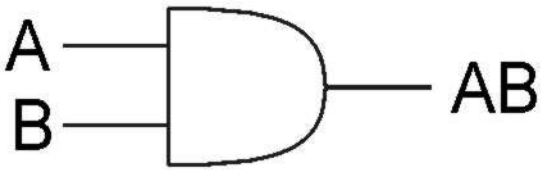

*NOT*



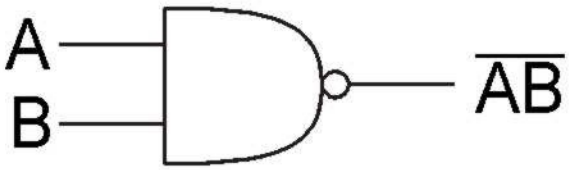

*OR*




*NOR*




*AND*




*NAND*

# 与或的转换：摩根定律（反演律）

- 摩根定律（反演律)：

  $\overline{A+B}=\overline{A}\cdot\overline{B}$　　$\overline{A\cdot B}=\overline{A}+\overline{B}$　　（证明：真值表）

- 可扩展

$$\overline{A+B+C}=\overline{A}\cdot\overline{B+C}=\overline{A}\cdot\overline{B}\cdot\overline{C}$$

$$\overline{A\cdot B\cdot C}=\overline{A}+\overline{B\cdot C}=\overline{A}+\overline{B}+\overline{C}$$

$$\overline{A_1+A_2+\mathbf{L}+A_n}=\overline{A_1}\cdot\overline{A_2}\cdot\mathbf{L}\cdot\overline{A_n}$$

$$\overline{A_1\cdot A_2\cdot\mathbf{L}\cdot A_n}=\overline{A_1}+\overline{A_2}+\mathbf{L}+\overline{A_n}$$

# 利用基本门电路设计功能电路

- *组合逻辑电路*
  - 输出只依赖当前的输入
  - 无状态（无存储电路）

- *时序逻辑电路*
  - 输出不仅依赖当前的输入还取决于电路过去的状态
  - 利用存储电路存储电路过去的状态信息
  - *时序逻辑电路=组合逻辑电路+存储电路*

- 我们先学习些常用的组合逻辑的功能电路,然后再学习时序电路。

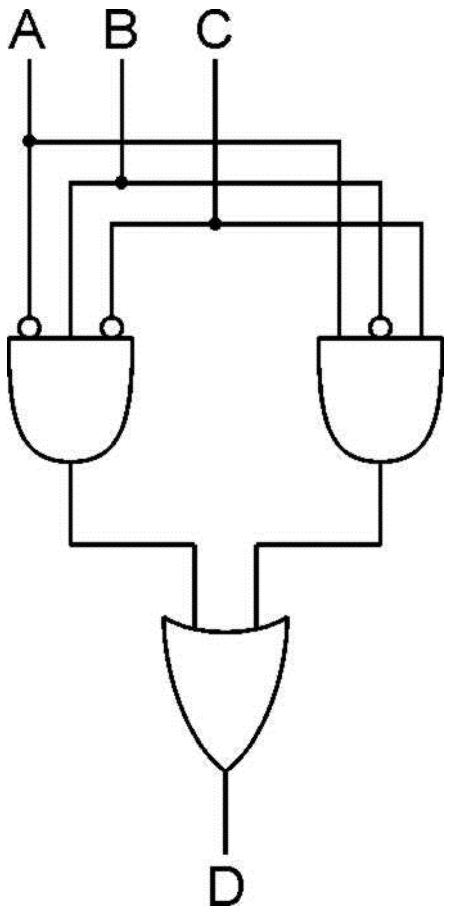
# 例子：

真值表 **à** 逻辑表达式的方法

可以将任意真值表转换成基于AND, OR, NOT操作的逻辑表达式(逻辑完备性)

- $D=A\overline{B}\overline{C}+\overline{A}B\overline{C}+\overline{A}\overline{B}C$

| A | B | C | D |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 |




1. AND combinations that yield a "1" in the truth table.

2. OR the results of the AND gates.

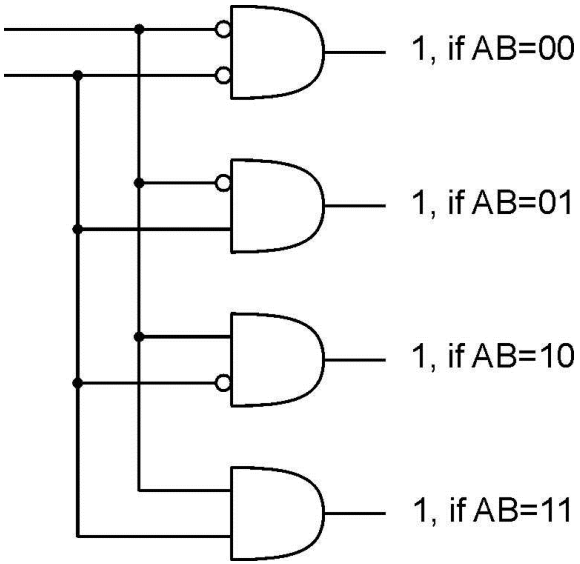
## 必须掌握的能力

【例】某厂有A、B、C三个车间和Y、 Z两台发电机。如果一个车间开工，启动Z发电机即可满足使用要求； 如果两个车间同时开工，启动Y发电机即可满足使用要求；如果三个车间同时开工，则需要同时启动Y、 Z两台发电机才能满足使用要求。 试仅用与非门和异或门两种逻辑门设计一个供电控制电路，使电力负荷达到最佳匹配。

**解** 用“0”表示该厂车间不开工或发电机不工作，用“1”表示该厂车间开工或发电机工作。为使电力负荷达到最佳匹配， 应该根据车间的开工情况即负荷情况，来决定两台发电机的启动与否。 因此，此处的供电控制电路中，A、B、C是输入变量，Y、Z是输出变量。由此列出电路的真值表如表1所示。

表1

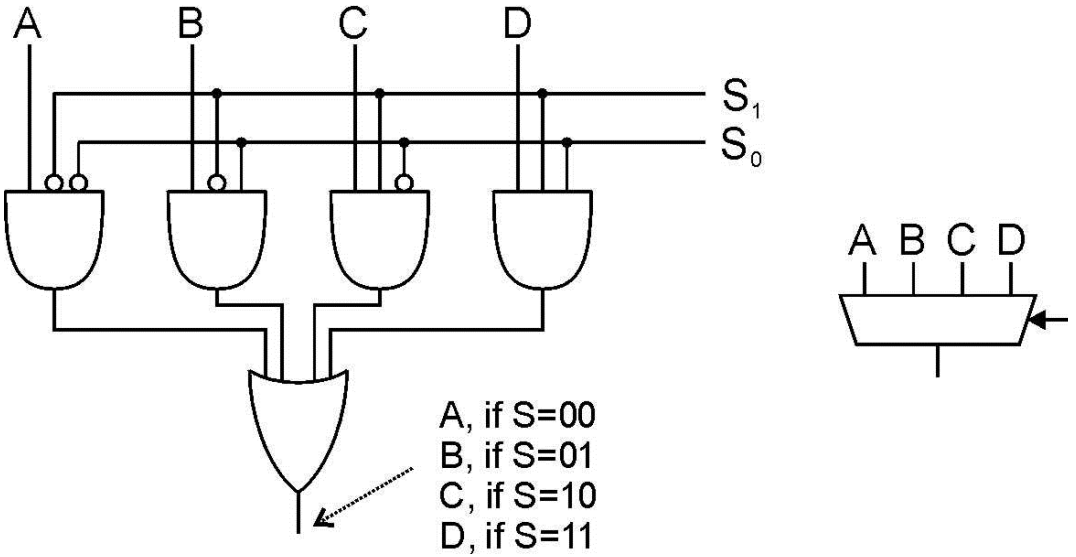
| A | B | C | Y | Z |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |



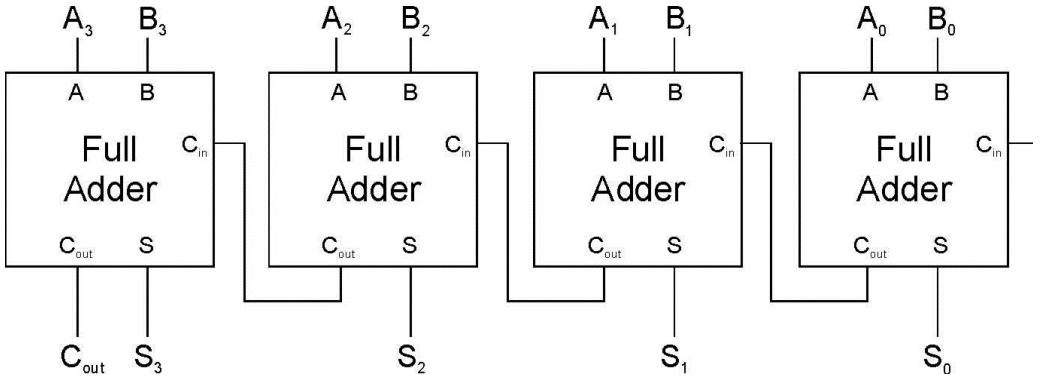

译码器

| A | B | $C_{in}$ | S | $C_{out}$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

一位全加器的概念




多路选择器，MUX




四位串行进位全加器的概念

# 时序逻辑电路


在对电路功能进行研究时，通常将某一时刻的状态称为**“现态”**，Q的位数决定了状态数量.

1位: Q0, 电路有两个状态 0,1

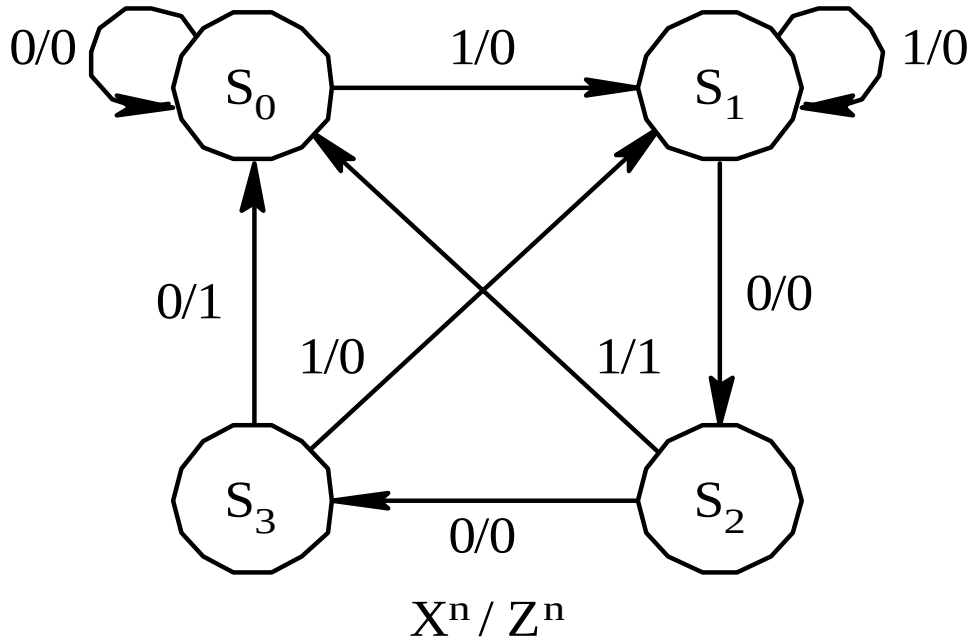
2位: Q0,Q1,电路有四个状态00,01,10,11

.....

将在某一现态下，外部信号发生变化后到达的新的状态称为**“次态”**。

# 使用状态转换图表示现态-次态转换关系

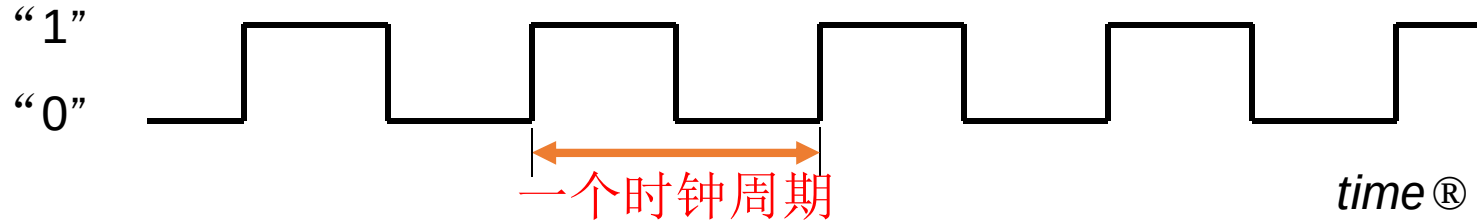
【例】某时序逻辑电路的状态图如图所示。假定电路现在处于状态$S_0$，试确定电路输入序列为X=1000010110时的状态序列和输出序列，并说明最后一位输入后电路所处的状态。




状态图

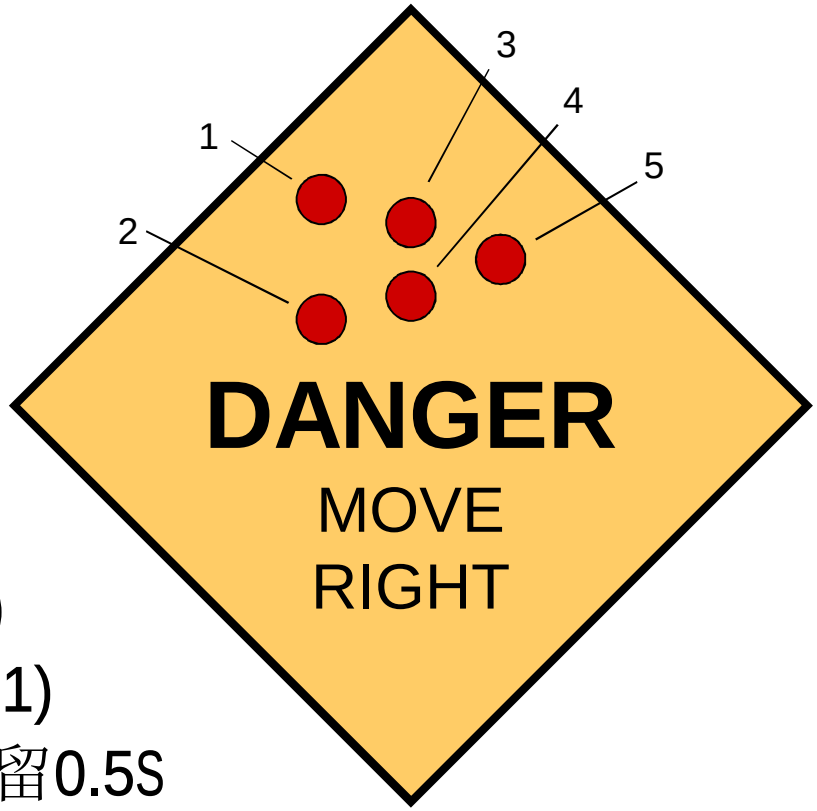
# 时钟

- 通常，状态转移是通过时钟来触发的。




- 在每个时钟周期的开始，基于当前状态和外部输入，状态机进行转换。

# 例子-课本-P53 （应熟练掌握）




交通警告牌:当通电工作时候的状态
(Switch=on)

- S00: No lights on (Z=0,Y=0,X=0
- S01:light1 & 2 on (Z=1,Y=0,X=0)
- S02:light1, 2, 3, & 4 on(Z=1,Y=1,X=0)
- S03:light1, 2, 3, 4, & 5on(Z=1,Y=1,X=1)

开关接通时反复循环，每个状态停留0.5S
(Switch=off)

- S00 开关断开时状态

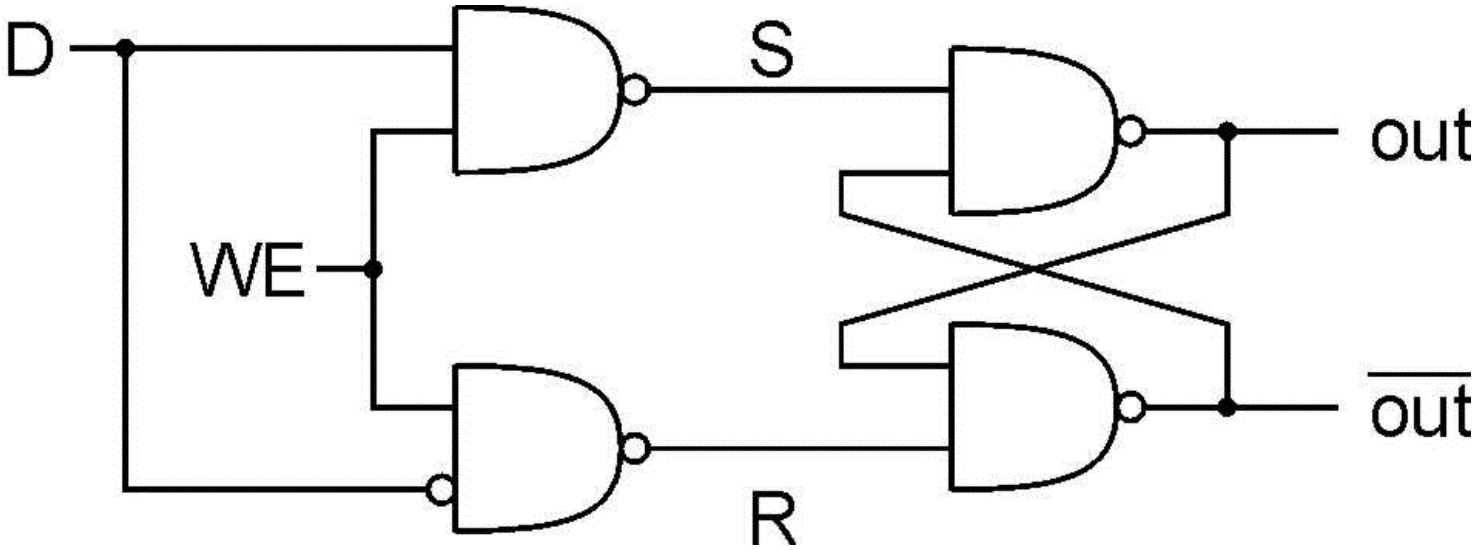
LED1,2 controlled by Z
LED3,4 controlled by Y
LED5 controlled by X

# 存储单元的概念

- 2个输入： D (data)， WE (write enable)
  - 当WE = 0,锁存器保持。(此时D信号变化不影响输出)
    - S = R = 1
  - 当WE = 1 (此时D信号变化影响输出,锁存器输出为D的值)

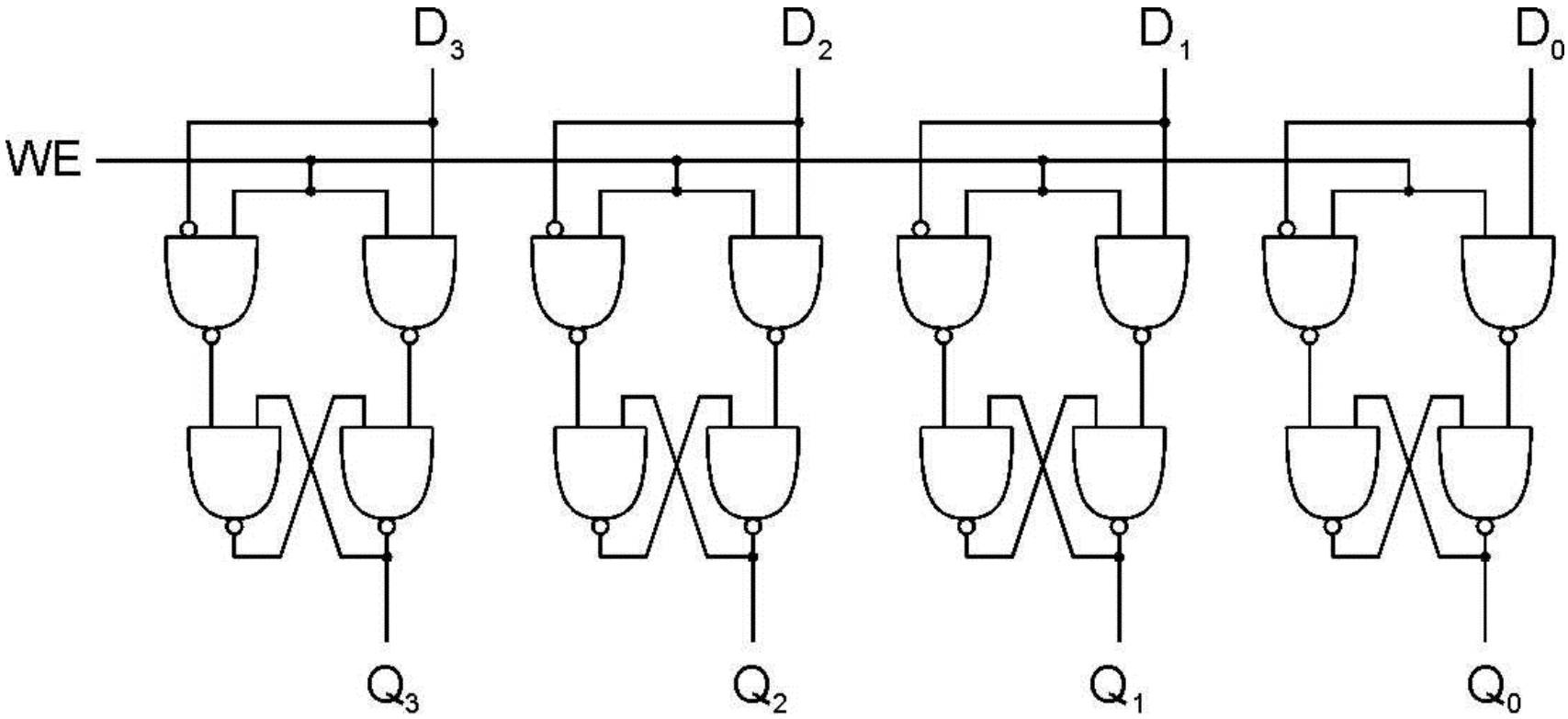
    D=0: S=1 ,R=0 out=0

    D=1: S=0, R=1 out=1 消除了约束。



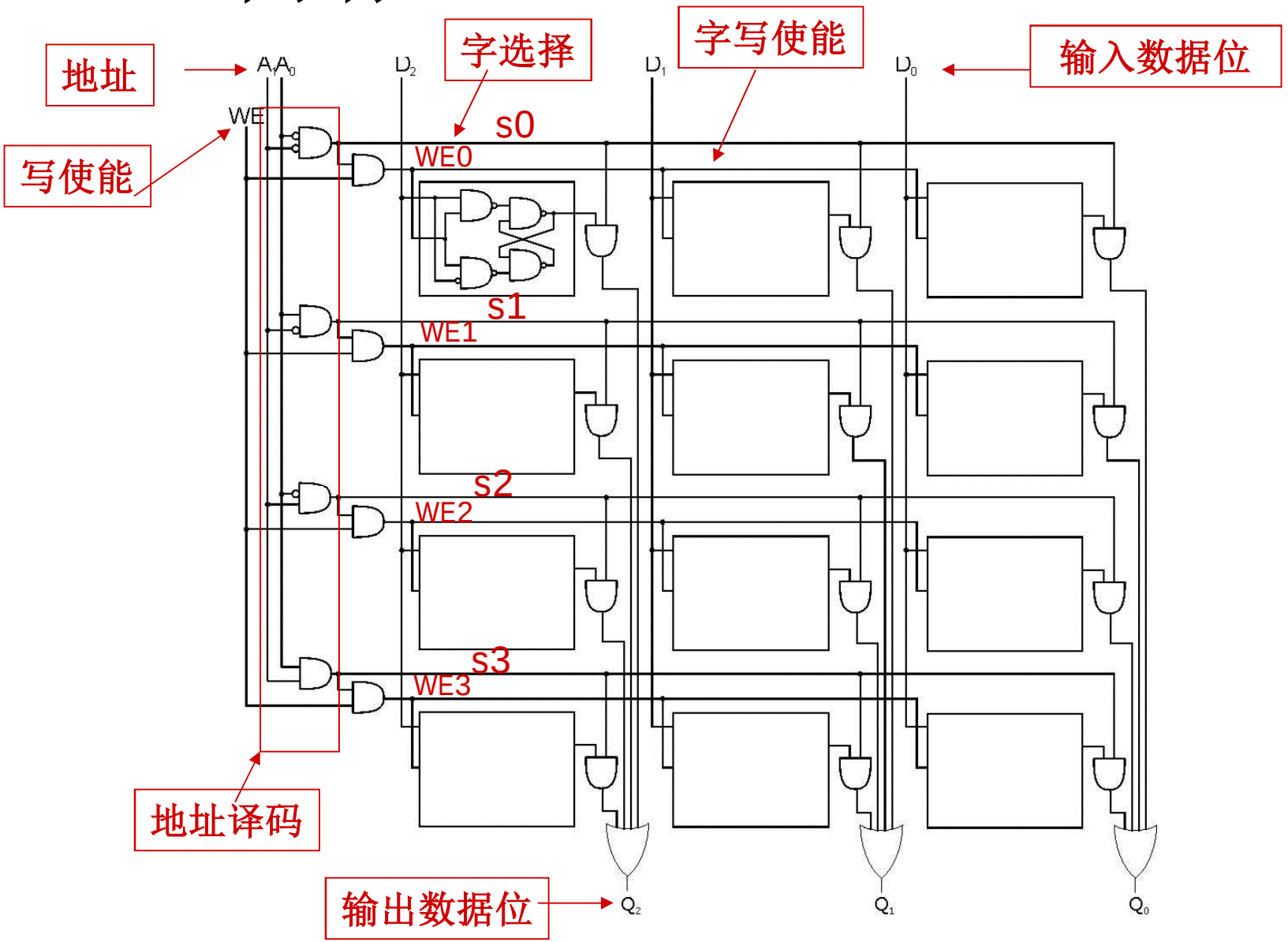
# 寄存器

- 寄存器存放多位数据，有多个锁存器组成。



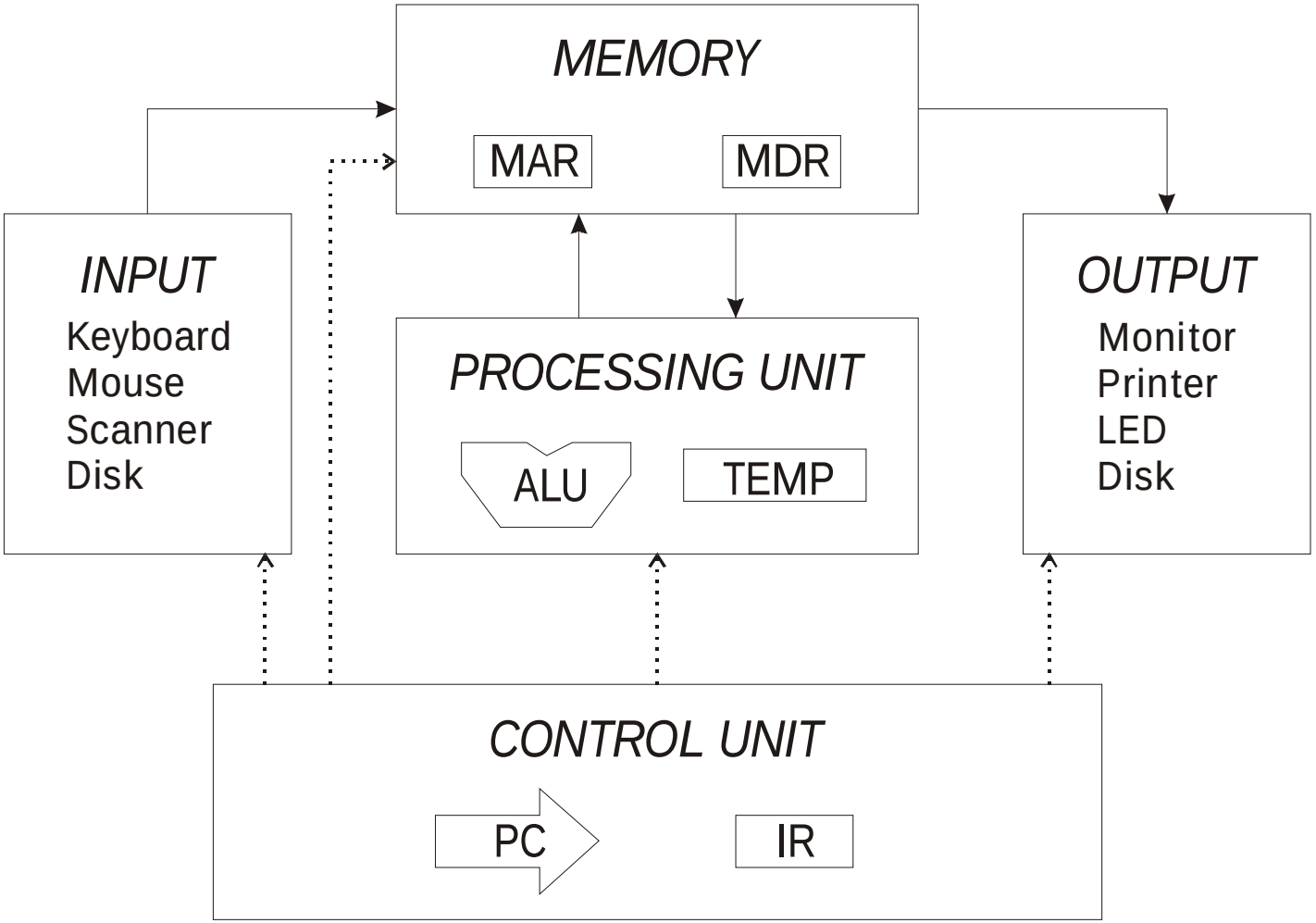

4-bit寄存器

# $2^2$ x 3-bit内存


地址
A1A0
字选择
字写使能
输入数据位
D2
D1
D0
WE
写使能
s0
WE0
s1
WE1
s2
WE2
s3
WE3
地址译码
输出数据位
Q2
Q1
Q0

# Chapter-4-5

# 3. 诺伊曼模型


MEMORY
MAR
MDR
INPUT
Keyboard
Mouse
Scanner
Disk
PROCESSING UNIT
ALU
TEMP
OUTPUT
Monitor
Printer
LED
Disk
CONTROL UNIT
PC
IR

# 5大部件

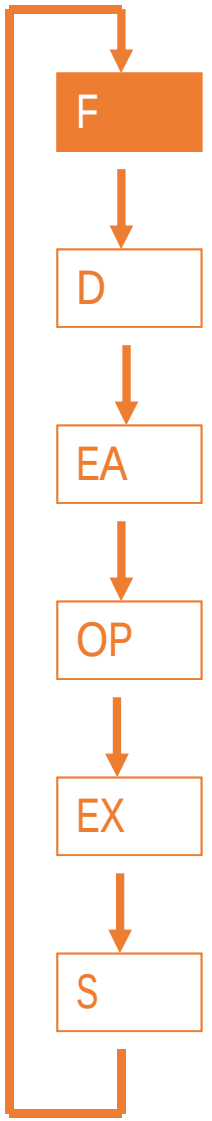
- 内存：包括存储单元，以及16位的MAR和16位的MDR寄存器。$2^{16}$x16bit
- 处理单元: 包括ALU和8个16位的寄存器(R0-R7)。
- 控制单元：PC,IR寄存器和控制逻辑有限状态机FSM
- 输入和输出单元：键盘和显示器。
- 部件的连接: 总线 ,同一时间只允许一个主设备使用

# 指令周期的六个步骤


Fetch ： 取指令
Decode： 译码
Evaluate address： 地址计算
Fetch operands ： 取操作数
Execute： 执行
Store result： 存放结果

# 指令周期：FETCH

- 从内存中取下一条指令(地址存放在PC寄存器) 到指令寄存器 (IR)。
  - 把PC中存放的内容拷贝到 内存的MAR寄存器中.
  - 给内存发读信号。
  - 拷贝MDR的内容到IR寄存器中。
  - PC=PC+1,PC指向下一条待执行的指令



# Instruction Set Architecture(ISA):指令集结构

- ISA = 向以机器语言编程的程序员提供有关控制机器所需要的所有必要信息。包括内存组织方式、寄存器组、指令集等信息。
  - 内存组织方式
    - 寻址空间 – 有多少个存储空间?
    - 寻址能力 – 每个存储空间有多少位?
  - 寄存器组
    - 有多少? 存储数据长度? 怎么使用?
  - 指令集
    - 操作码
    - 数据类型
    - 寻址模式

# LC-3 Overview: 内存组织和寄存器

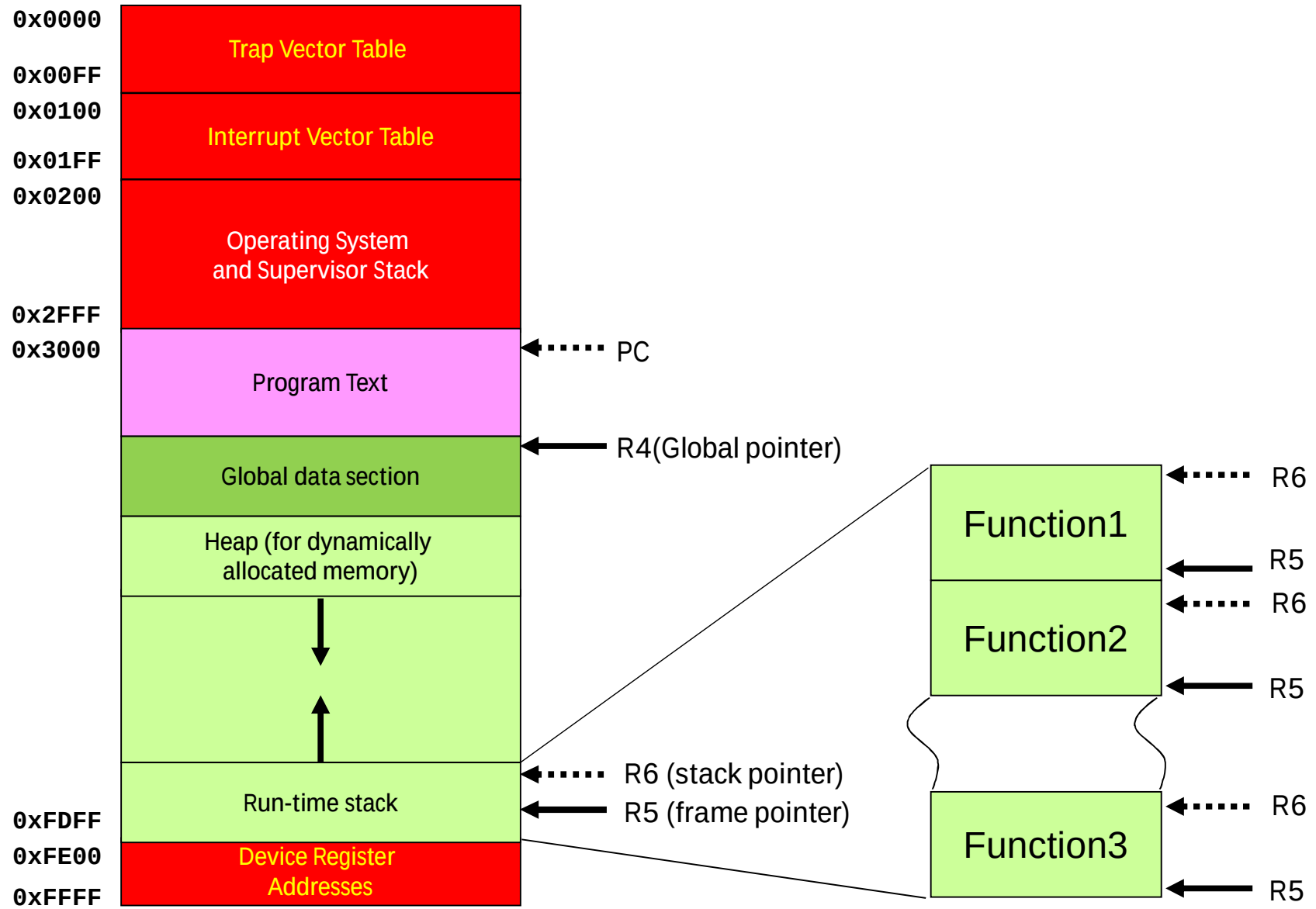
内存组织

- 寻址空间: $2^{16}$ 个存储单元 (16位地址)
- 寻址能力: 16 bits

寄存器组

- 提供一个快速的临时存储空间, 通常可在一个机器周期的时间访问到
  - 访问存储器的时间往往远大于一个机器周期的时间
- 8个通用寄存器: R0 - R7
  - 每个可存储数据宽度为16 bits
  - 寄存器编址需要多少位二进制?
- 其它寄存器
  - 程序员不能直接访问，被指令使用或影响
  - PC (program counter), condition codes， MDR， MAR

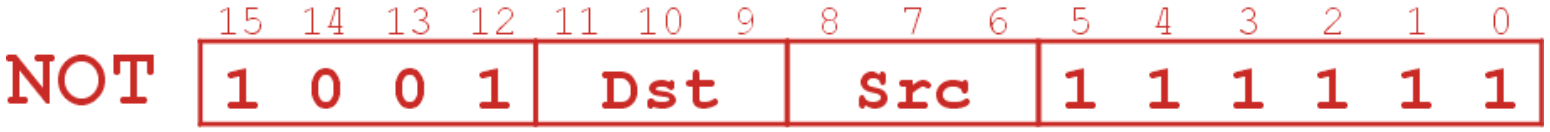
# Memory Map of the LC-3



# LC-3 Overview: 指令集

- 操作码
  - 15 个操作码(P79)
  - 逻辑和运算指令: ADD(0001), AND(0101), NOT(1001)（助记符）
  - 数据搬移指令: LD(0010), LDI(1010), LDR(0110), LEA(1110), ST(0011), STR(0111), STI(1011)
  - 控制指令: BR(0000), JSR/JSRR(0100), JMP(1100), RTI(1000), TRAP(1111)
  - 目的操作数为寄存器的指令会根据写入寄存器的值设置条件码
    - N = 写入值为负(<0), Z =写入值为零(=0), P =写入值为正(> 0)
    - 组合 NZ(<=0)/ZP(>=0)/NP(<>0)/NZP(?)
- 数据类型: 16位定点补码整数
- 寻址方式: 指令中指示参与运算操作数存储位置的方法
  - 非内存寻址（操作数不在内存中）:
    直接寻址（操作数在指令中）, 寄存器寻址（操作数在寄存器中）
  - 内存寻址（操作数在内存中）: *PC-相对*, 间接, 基址+*偏移*

# NOT （SRC/DST两个操作数必须是寄存器）



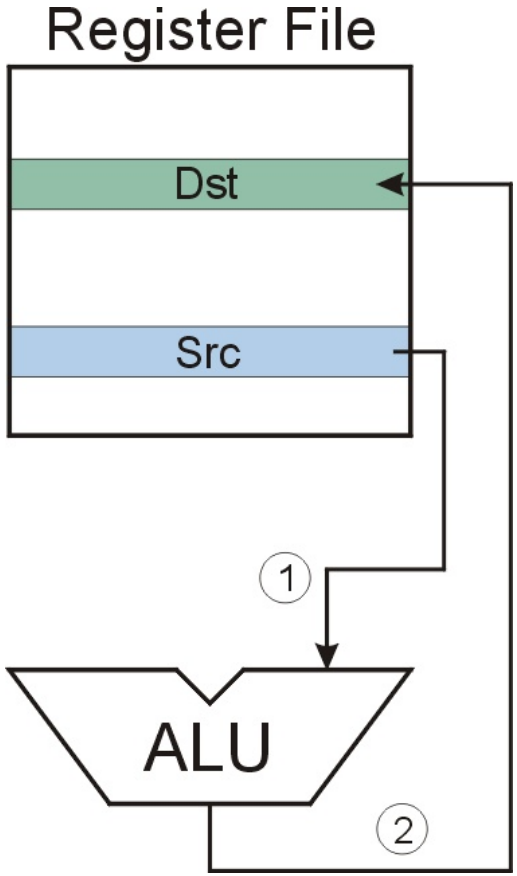

*Note: Src 和 Dst*
*可以是同一个寄存器*

*NOT R2,R3*

*1001 010 011 111111*

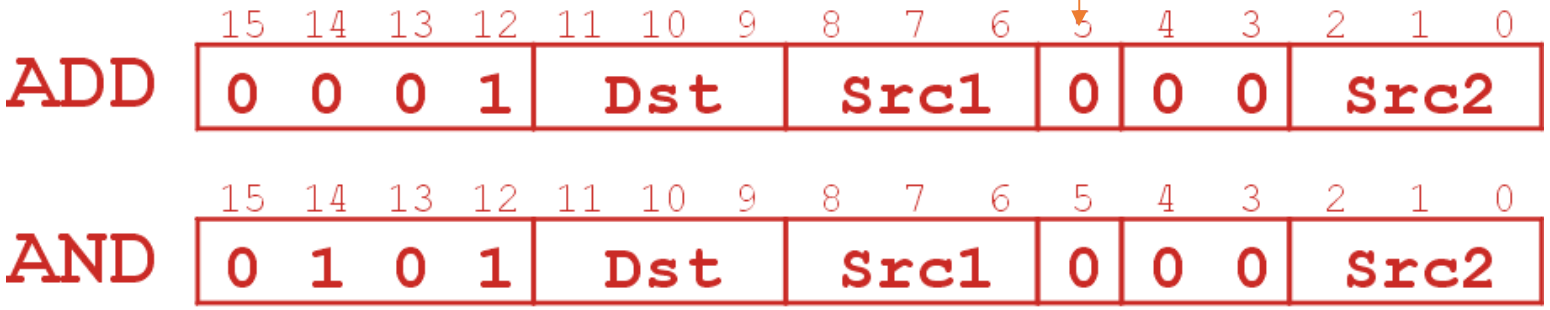
*NOT R2,R2*

*1001 010 010 111111*



# ADD/AND（寄存器模式）

为0指示“register mode”




*Note: 寄存器模式，两个源操作数和1个目的操作数都为寄存器。*

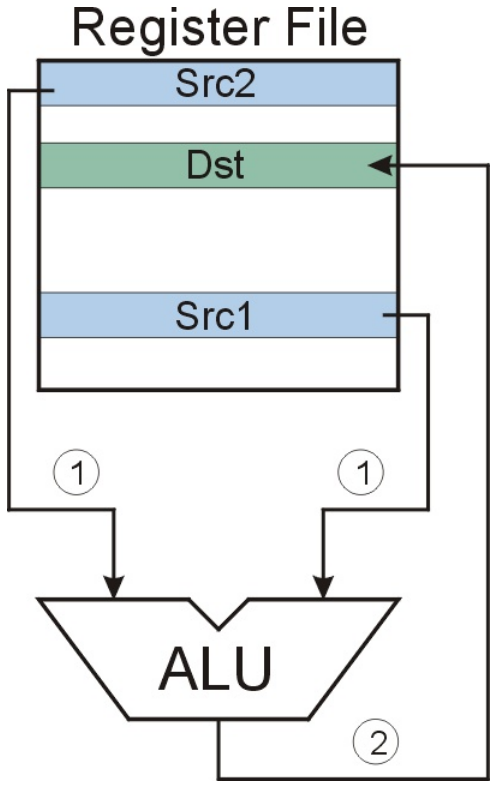
*Note: Src1/2 和 Dst 可以是同一个寄存器*

*ADD R1,R2,R3*

*0001 001 010 000 011*

*ADD R1,R1,R3*

*ADD R1,R1,R1*



# 数据搬移指令

- Load – 从内存中读数据到寄存器中
- 按内存数的寻址方式不同可分为：
  - LD: PC-相对寻址模式
  - LDR: 寄存器基址+偏移模式
  - LDI: 间接寻址模式

- Store – 写寄存器值到内存
- 按内存数的寻址方式不同可分为：
  - ST: PC-相对寻址模式
  - STR:寄存器基址+偏移模式
  - STI: 间接寻址模式

- LEA - 计算操作数的有效地址，存放到寄存器
  - LEA: 用立即数的方式给出操作数相对PC的偏移
  - *LEA指令不访存*

# PC相对寻址模式

- LC-3指令长度16位，内存地址长度16位，能在指令中直接给出内存数的有效地址吗？
  - 16位指令中操作码占用4 bits，一个目的寄存器需要占用 3 bits，只剩下9位来编码地址了
  - 不可能直接给出16位地址! 怎么解决
- 解决方法:
  - 利用PC寄存器
  - 剩下的 9 bits 用来表示数据地址和 PC的偏移量(offset).
  - 数据有效地址为PC+offset
  - 局限性：9 bits: $-256 \le \text{offset} \le +255$
- 数据的地址范围为：$PC - 256 \le X \le PC + 255$
- 注意：PC不是当前指令的地址，而是下一条指令的地址

# （基于PC的）间接寻址模式

- LC-3的PC相对寻址模式, 只能访问PC前或后256的内存单元
  - 剩下的内存怎么访问?

- 解决方案 #1:
  - 在PC相对寻址能访问到的内存单元存放一个16位地址（不是数据了）.先读取这个地址，然后以这个地址去访问内存。
  - 类似C语言的指针

# （寄存器）基址偏移寻址模式

- LC-3的PC相对寻址模式, 只能访问PC前或后256的内存单元
  - 剩下的内存怎么访问?

- 解决方法 #2:
  - 利用寄存器存放一个16位地址.偏移以立即数形式存放在指令中
  - 类似C语言的数组。int a[10]; *(a+2)
- 指令编码：
- 操作码占用4 bits, 寄存器占用3bits ， 基址寄存器占用3 bits
- 剩下6bits可用作偏移。$-32 \leq \text{offset} \leq +31$
-
- 基址寄存器中如何设置16位地址？ LD、LEA

# LEA :Load Effective Address 计算有效地址

- 计算操作数的有效地址，存放到寄存器
  - LEA: 用立即数的方式给出操作数相对PC的偏移(9bits)
  - 计算方法：PC+offset ->寄存器
  - *LEA指令不访存*

- Note: 寄存器里面存放的是地址,而不是内存单元存放的数据。

- 应用:访问连续的数据区域,用LEA 指令得到数据区域的起始地址,然后用LDR指令访问.

# 控制指令

- 通过更新PC，改变程序执行顺序

- 条件跳转
  - 跳转到分支仅当指定的条件成立
    - 更新PC到分支地址，通过在当前的PC值加一个偏移实现
  - 否则, 不跳转到分支
    - PC 不改变，顺序执行下一条指令

- 无条件跳转(直接跳转)
  - PC值肯定被改变到目标地址

- TRAP（陷入指令）
  - 改变PC 到操作系统提供的服务子程序的入口地址。 “service routine”
  - 服务子程序完成后返回到TRAP指令后一条程序代码继续执行。

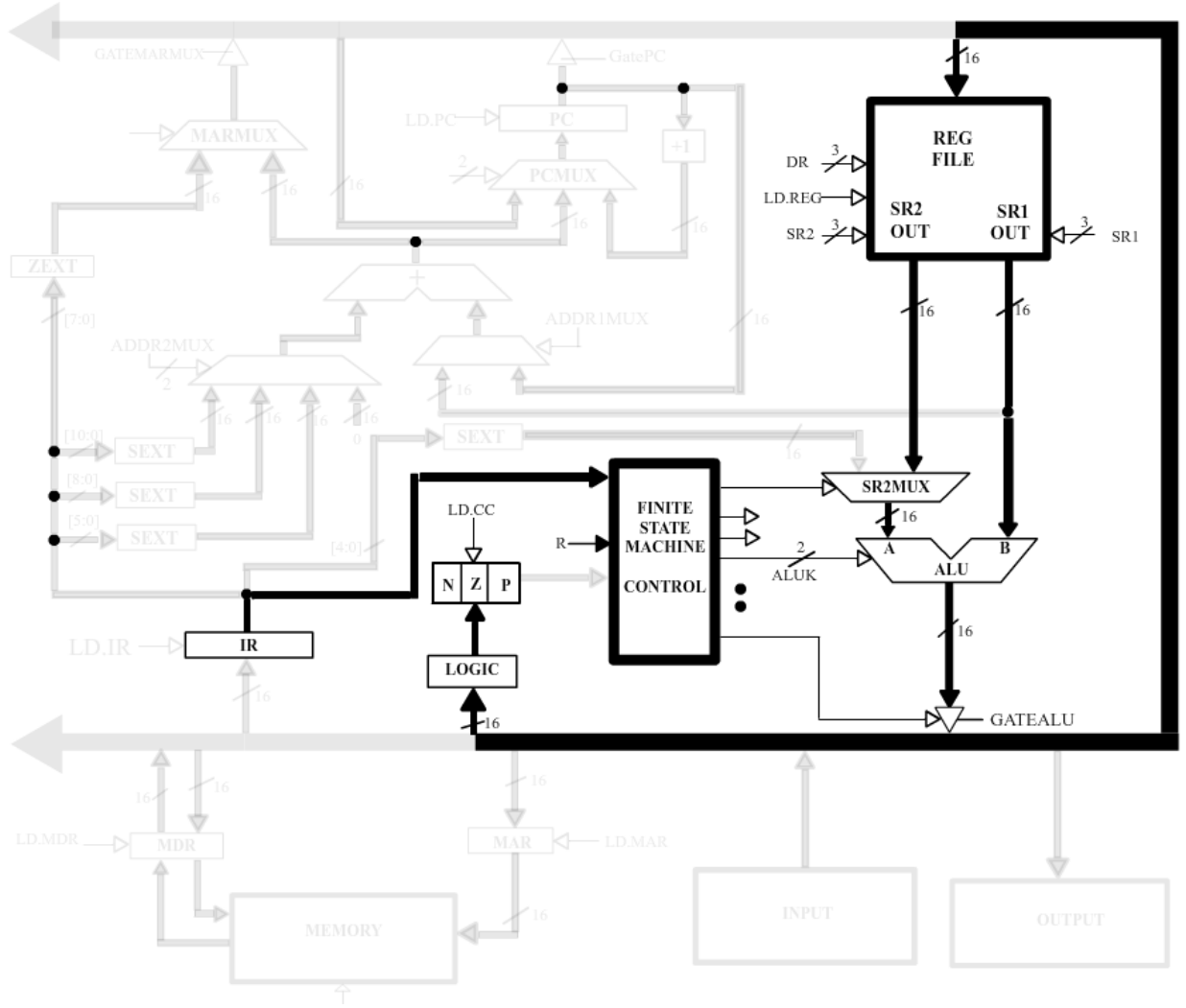
# 条件码

- LC-3 有3个1位的条件码寄存器，由最近写入的寄存器值确定
  - N – negative（〈0）
  - Z – zero （=0）
  - P -- positive (〉0)

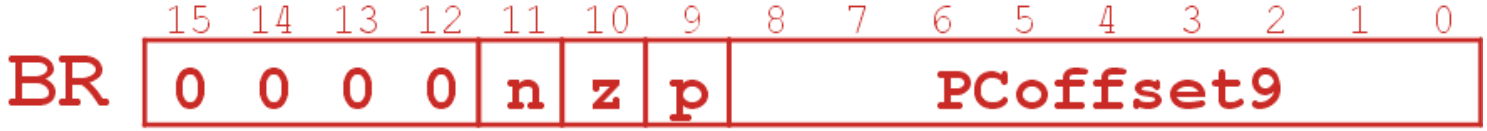
- N Z P 同一时刻只有一个标志位会改变
  - 由最近写入的寄存器值确定
  - 任何一条写寄存器的指令都会改变条件码 (ADD, AND, NOT, LD, LDR, LDI, LEA)
  - Store指令和控制指令不改变条件码

```
AND R1,R1,#0      NOT   R1,R1     NOT  R1,R1
 x3000:LEA R1,PC+25
ADD  R1,R1,#-1
ADD  R1,R1,#15
```

# 条件码(NZP 逻辑：只有写寄存器的指令才影响NZP标志)



# :件跳转指令

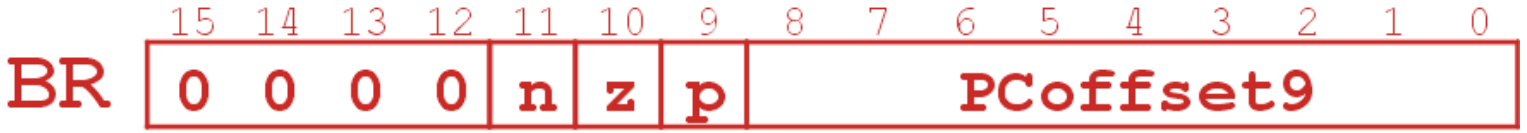


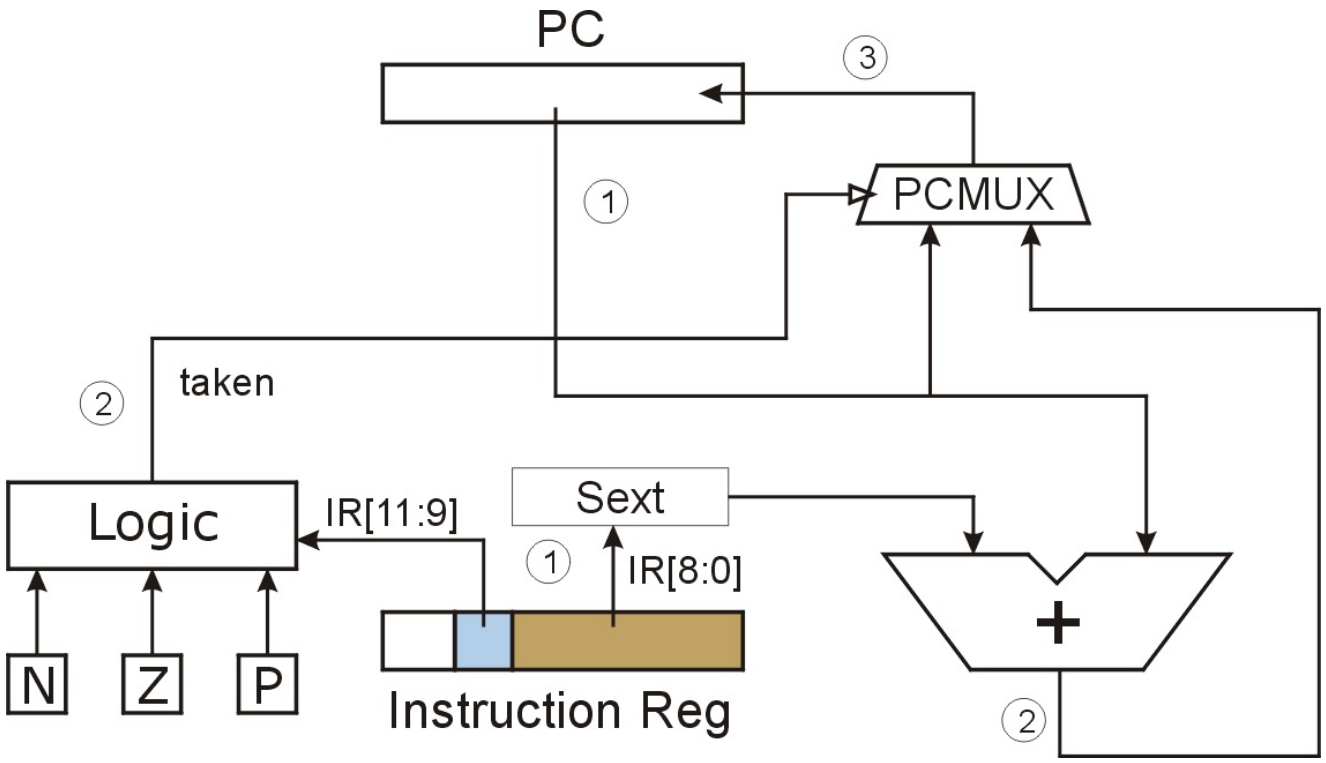

- 在跳转指令中指示需要检测哪个条件码(IR[11:9])（可以是一个或者多个），如果指定的条件码成立，则跳转，否则不跳转。
  - 目标地址采用了PC相对寻址
    target address = PC + offset (IR[8:0])
  - Note: PC不是当前指令地址，而是下一条指令的地址。
  - Note: 只能跳转到跳转指令的前255条指令或后256条指令。
  - Note: 必须和上一条会修改寄存器的指令配合使用

  Ex:

  - X3100: 0001 001 001 1 11111
  - X3101 : 0000 010 0000 00100
-

# (PC相对寻址)



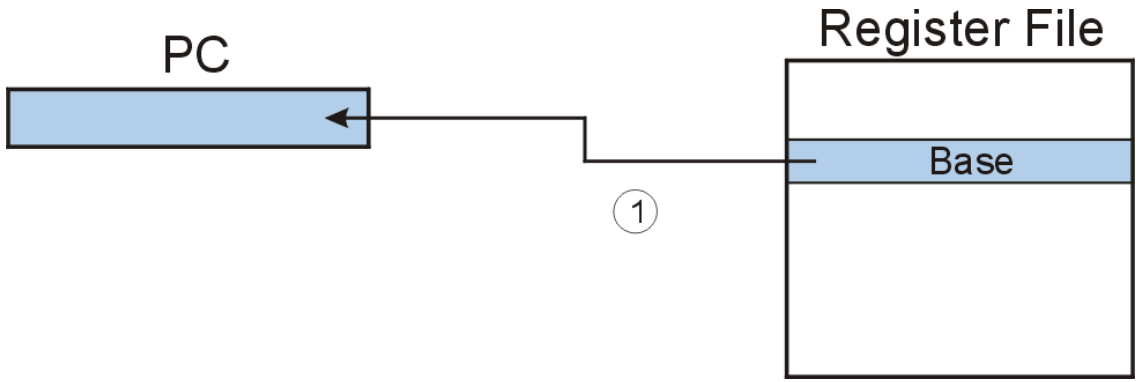

*What happens if bits [11:9] are all zero? All one?*



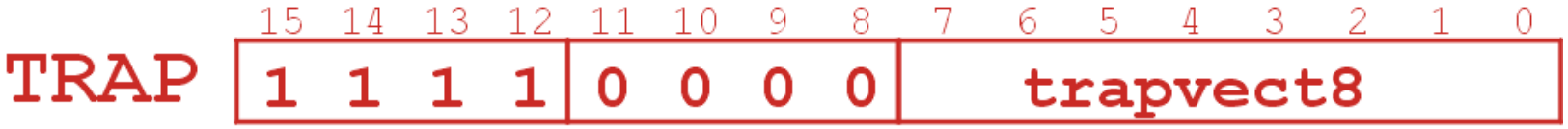
*指令助记: BRn /BRz /BRp /BRnzp /BRnz /BRnp /BRzp*

# JMP（寄存器存放跳转地址）

- JMP是一个绝对跳转指令 – 总是跳转
  - 跳转的目标地址存放在寄存器中
  - 寄存器可以存放16位地址。可以跳转到任何地方。
  - 条件跳转是有局限性的：-256到+255

# AP：调用系统服务程序




•调用系统服务程序,服务程序由 8-bit “trap vector”指定，总共支持256个服务程序。

| *vector* | *routine* |
|---|---|
| **x23** | **input a character from the keyboard to R0** |
| **x21** | **output a character in R0 to the monitor** |
| **x25** | **halt the program** |

•调用结束后, PC被设置成当前TRAP 指令的下一条。

•(具体原理后面会讨论)

- x3100: 1111 0000 0010 0101

# Trap 指令

LC-3 汇编器提供trap “伪指令”，方便编程员使用，无需记忆系统调用号

| *Code* | *Equivalent* | *Description* |
|---|---|---|
| **HALT** | **TRAP x25** | **Halt execution and print message to console.** |
| **IN** | **TRAP x23** | **Print prompt on console, read (and echo) one character from keybd. Character stored in R0[7:0].** |
| **OUT** | **TRAP x21** | **Write one character (in R0[7:0]) to console.** |
| **GETC** | **TRAP x20** | **Read one character from keyboard. Character stored in R0[7:0].** |
| **PUTS** | **TRAP x22** | **Write null-terminated string to console. Address of string is in R0.** |

# Trap 指令使用

- 1 输出一个字符
- ; the char must be in R0
- TRAP x21
- or
- OUT

- 2 To read in a character
- ; will go into R0[7:0], no echo.
- TRAP x20
- or
- GETC

**3 To end the program**

**TRAP x25**
or
**HALT**

**4显示字符串**

```
LEA R0,hello
PUTS
…
hello .STRINGZ “Hello!”
```

求：掌握根据p79页ISA表对简单程序进行翻译的能力，如下面程序

| *Address* | *Instruction* | *Comments* |
|---|---|---|
| x3000 | 0 1 0 1 0 1 0 0 1 0 1 0 0 0 0 0 | *R2 ¬ 0 (counter)* |
| x3001 | 0 0 1 0 0 1 1 0 0 0 0 1 0 0 0 0 | *R3 ¬ M[x3102] (ptr)* |
| x3002 | 1 1 1 1 0 0 0 0 0 0 1 0 0 0 1 1 | *Input to R0 (TRAP x23)* |
| x3003 | 0 1 1 0 0 0 1 0 1 1 0 0 0 0 0 0 | *R1 ¬ M[R3]* |
| x3004 | 0 0 0 1 1 0 0 0 0 1 1 1 1 1 0 0 | *R4 ¬ R1 – 4 (EOT)* |
| x3005 | 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 | *If Z, goto x300E* |
| x3006 | 1 0 0 1 0 0 1 0 0 1 1 1 1 1 1 1 | *R1 ¬ NOT R1* |
| x3007 | 0 0 0 1 0 0 1 0 0 1 1 0 0 0 0 1 | *R1 ¬ R1 + 1* |
| X3008 | 0 0 0 1 0 0 1 0 0 1 0 0 0 0 0 0 | *R1 ¬ R1 + R0* |
| x3009 | 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 1 | *If N or P, goto x300B* |

# Chapter 6-7

# 操作码、操作数

- 操作码
  - 和LC-3指令集定义的操作码对应的助记符号，不能再用作标号，系统专用并保留
  - 具体参见Appendix A
    - ex: ADD，AND，LD，LDR, ...
- 操作数
  - 寄存器数– 寄存器 Rn, n是寄存器编号
  - 立即数 – 数字用 # (十进制) or x (十六进制)表示
  - 内存数 – 标号，用符号表示的内存地址
  - 用‘，’分割操作数
  - 数字, 操作数的顺序以及类型和遵守机器码指令的定义
    - ex:

```
ADD  R1,R1,R3
ADD  R1,R1,#3
LD   R6,NUMBER
BRz  LOOP
```

# 数据类型

- LC-3 只有两种基本数据类型

- 定点整数：Integer 16位补码
- 字符： Character 16位

- 都占用 16 bits wide (a word)
- 但实际字符只占用8 bit, 怎么存储??---高位零扩展

# 标号

- 标号：Label
  - 放在每行代码的开始的地址符号，表示该行代码或数据的地址，
  - 符号化的内存地址. 一般两种类型:
    - 分支和跳转语句的目标地址
    - 数据的存放地址
    - ex:

```
LOOP  ADD  R1,R1,#-1
      BRp  LOOP
```

    - Ex:

```
      LD   R2, NUMBER
      …
      …
      NUMBER   .BLKW      1
```

# 注释

- 注释：Comment
  - ‘；’之后的所有字符都是注释
  - 汇编器将忽略所有的注释
  - 注释用于帮助程序员理解程序和存档的需求
  - 注释的技巧
    - 不要滥用注释, 比如 “R1加1”，没有提供比指令更多的信息
    - 提供更深的洞察力, 比如 “指针加1指向下一个访问的数据”
    - 分隔代码片段

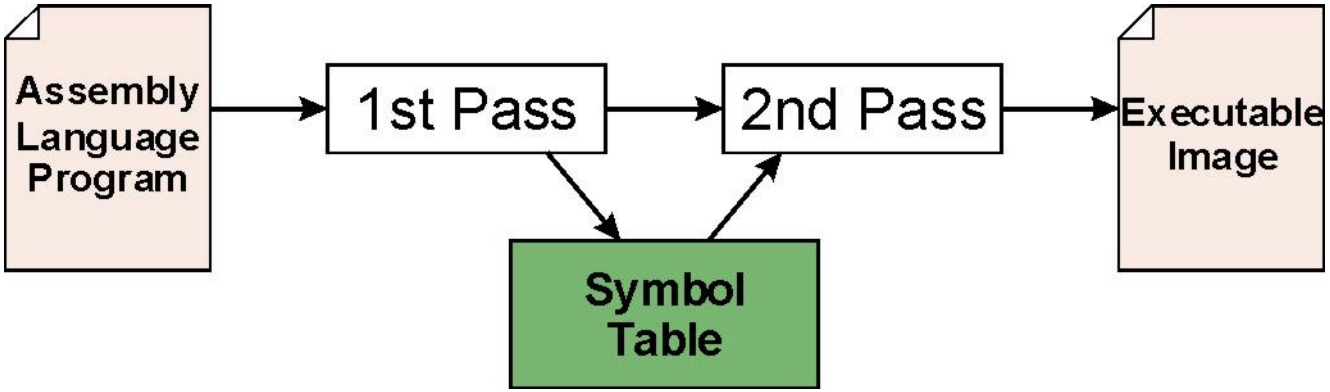
# 汇编器的伪操作

- 伪操作
  - 不对程序产生效果，不是执行指令
  - 仅供汇编器使用
  - 区别于指令，以‘.’开始

| *Opcode* | *Operand* | *Meaning* |
|---|---|---|
| .ORIG | address | 指示程序起始地址 |
| .END | | 指示程序在此结束，注意并不停止程序 |
| .BLKW | n | 分配n个字的内存单元空间 |
| .FILL | n | 分配一个字的内存单元空间并初始化为n |
| .STRINGZ | n-character string | 定义一个大小为n的字符串，占用n+1个内存单元。第n+1个字符为’\0’。 |

# LC-3汇编过程

•把汇编语言源程序(.asm)转换为可执行机器代码的过程(.obj)




•过程：两遍扫描

•第一遍（First Pass）：创建符号表

- 扫描源程序文件
- 找到所有的标号，并计算对应的地址
  产生所谓的符号表（*symbol table*）

•第二遍（Second Pass）生成机器语言程序代码

- 利用符号表信息将指令转化为机器语言代码

# 第一遍（ First Pass ）： 创建符号表

1. 找到.ORIG 伪操作
   确定第一条指令的起始汇编地址
   - 初始化地址跟踪计数器 (LC： location counter), 用于记录每条当前指令的地址

2. 依次扫描源程序的每一行代码
   a) 如果代码行存在标号, 将标号和指令对应的LC添加到符号表中.
   b) LC+1
      - 说明: 如果碰到伪指令 .BLKW 或 .STRINGZ, 则LC的增量为对应分配的字数。

      空行不处理

3. 碰到.END伪操作则停止汇编过程。



- NOTE: 空行：只有一个‘；’号起始的行

# 创建程序的符号表

```
;
; Program to multiply a number by
the constant 6
;
          .ORIG    x3050
          LD       R1, SIX
          LD       R2, NUMBER
          AND      R3, R3, #0        ;
Clear R3.  It will
                                     ;
contain the product.
; The inner loop
;
AGAIN     ADD      R3, R3, R2
          ADD      R1, R1, #-1       ;
R1 keeps track of
          BRp      AGAIN             ;
the iteration.
;
          HALT
;
NUMBER    .BLKW    1
SIX       .FILL    x0006
MESSAGE   .STRINGZ "CSI IS COOL"
          .BLKW #20
BOTTOM    .BLKW #1
;         .END
```

| Symbol | Address |
|---|---|
| | |
| | |
| | |
| | |
| | |

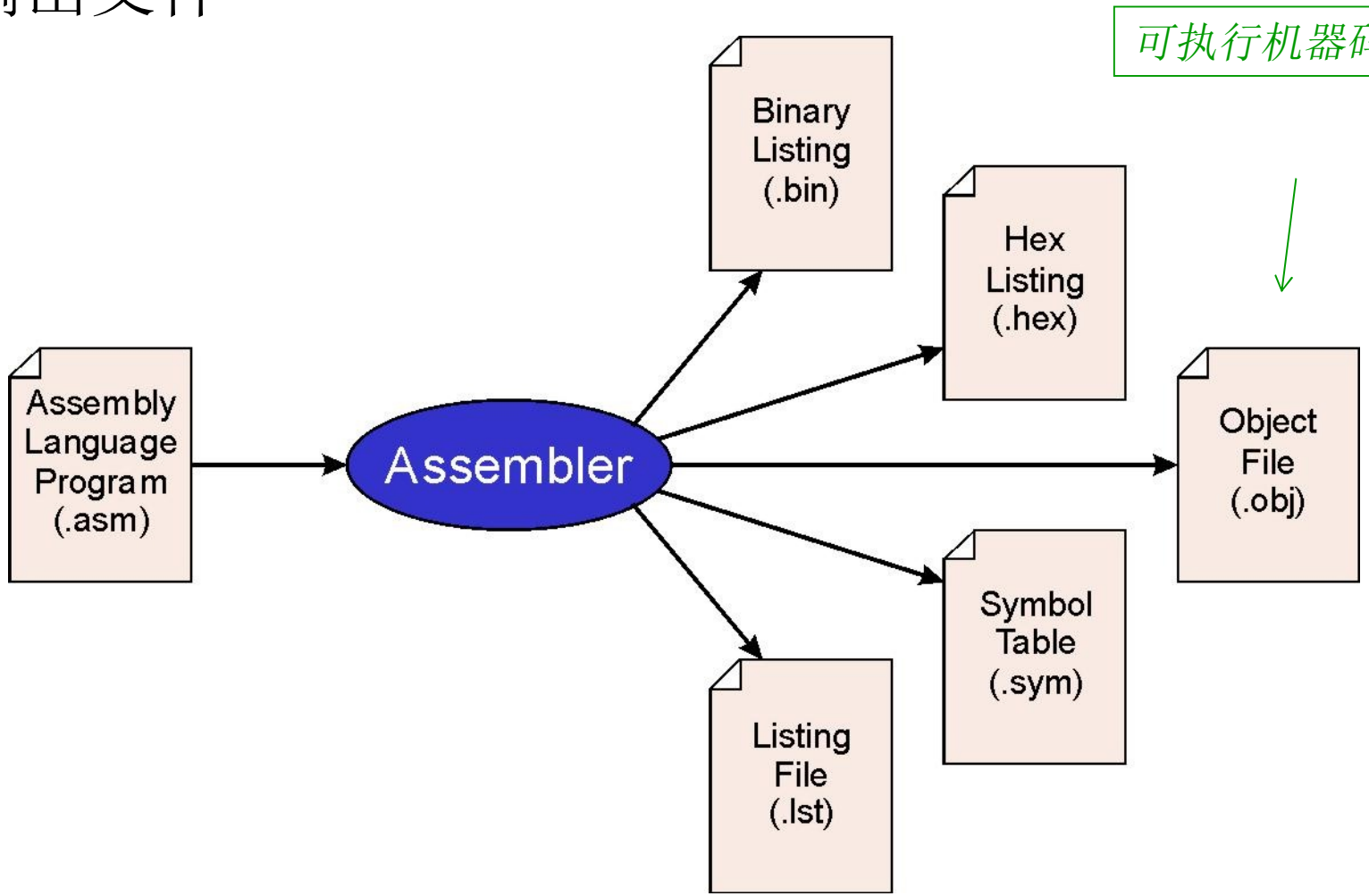
# 二遍（ Second Pass ） 生成机器语言程序

- 对每条可执行的指令，将其转换为机器语言代码
  - 如果操作数是一个标号
    从符号表中查找其地址并计算

- 可能存在的问题:
  - 操作数个数或类型不对
    - ex:
```
NOT  R1,#7
ADD  R1,R2
ADD  R3,R3,NUMBER
```
  - 立即数的范围太大
    - ex: `ADD  R1,R2,#1023`
  - 标号对应的地址相聚指令太远
    - 超过了PC相对寻址的范围（-256~+255）

# -3 汇编器

用“assemble” (Unix) or LC3Edit (Windows),
生不同的输出文件



# Chapter 8-10

# 入/输出：与外界相连接

- I/O设备种类按以下特征分为几类:
  - 功能: 输入, 输出, 存储设备
    - 输入: 键盘, 运动检测器,网络接口
    - 输出: 显示器, 打印机, 网络接口
    - 存储: 软盘, 光盘
  - 速率: 数据能传输的多快?
    - 键盘: 100 bytes/sec
    - U盘: 30-40 MB/s
    - 网络: 1 Mb/s - 1 Gb/s

# 编程接口

- 怎么访问设备内部寄存器?两种编址方式
  - 内存-映射(寄存器作为内存的一部分) vs. 专用IO指令(寄存器具有独立地址空间，可和内存重叠)

- 快速处理器和慢速的外部设备之间传输的时序是怎么控制的?
  - 异步 I/O vs. 同步 I/O

- 谁控制传输?
  - CPU (轮询) vs. 设备 (中断)
  - 举例：在家里等客人

# 传输时序控制:异步 I/O 同步 I/O

- I/O 事件一般发生的比CPU周期慢得多.
- 同步
  - 数据以一种固定的、可预测的数据供给
  - CPU 以某个固定的周期进行读/写操作
- 异步
  - 数据速率不可预测
  - CPU必须同步装置，以免遗失数据或者写入太快
  - 方法:设置状态寄存器或者标志位,在操作前检查设备是否准备好

# 主控制传输:CPU（轮询）vs. 设备（中断）

- 谁决定下一个数据传送何时发生?

•轮询

- CPU不断检查状态寄存器，直到新的数据到达或者设备已经为下一个数据做好准备
- “客人到了没?客人到了没?客人到了没?”
- 缺点: CPU和外设串行工作,利用效率低.

•中断

- 当新数据到达或者设备已经为下一个数据做好准备时，设备会发送一个特殊信号到CPU
- CPU在此期间可以执行其他任务，而不是反复轮询.
- “当客人到达时请通知我.”
- Cpu和外设可并行工作,利用效率高,但需要专用中断硬件支持.

# -3 的IO机制

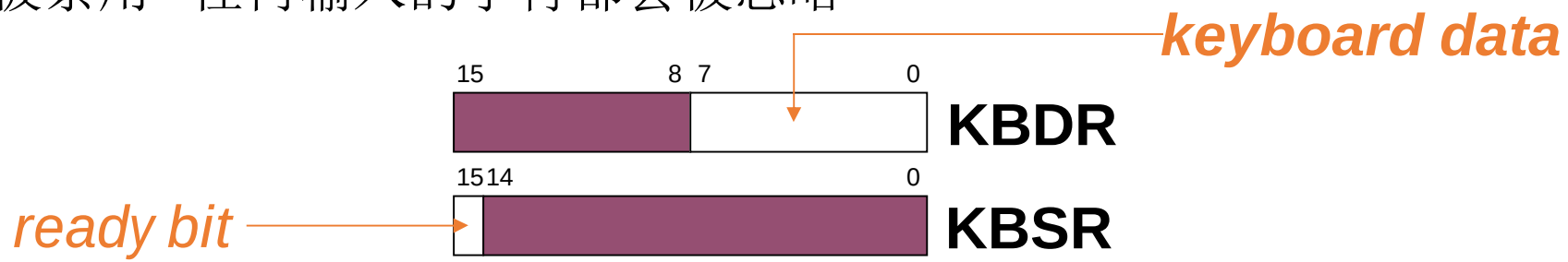
- 内存映射的输入输出 (Table A.3)

| *地址* | *输入/输出寄存器* | *作用* |
|---|---|---|
| **xFE00** | 键盘状态寄存器(KBSR) | 第十五位为1当键盘收到新的字符. |
| **xFE02** | 键盘数据寄存器(KBDR) | 第0到7位包含键盘上打出最后一个字符. |
| **xFE04** | 显示输出状态寄存器(DSR) | 第十五位为1当设备准备好,可向屏幕显示一个字符. |
| **xFE06** | 显示输出数据寄存器(DDR) | 写入到第7到0为的字符将显示在屏幕上. |

- 异步装置
  - 通过状态寄存器进行同步
- 轮询 和 中断
  - 中断的实现细节将会在第10章讨论

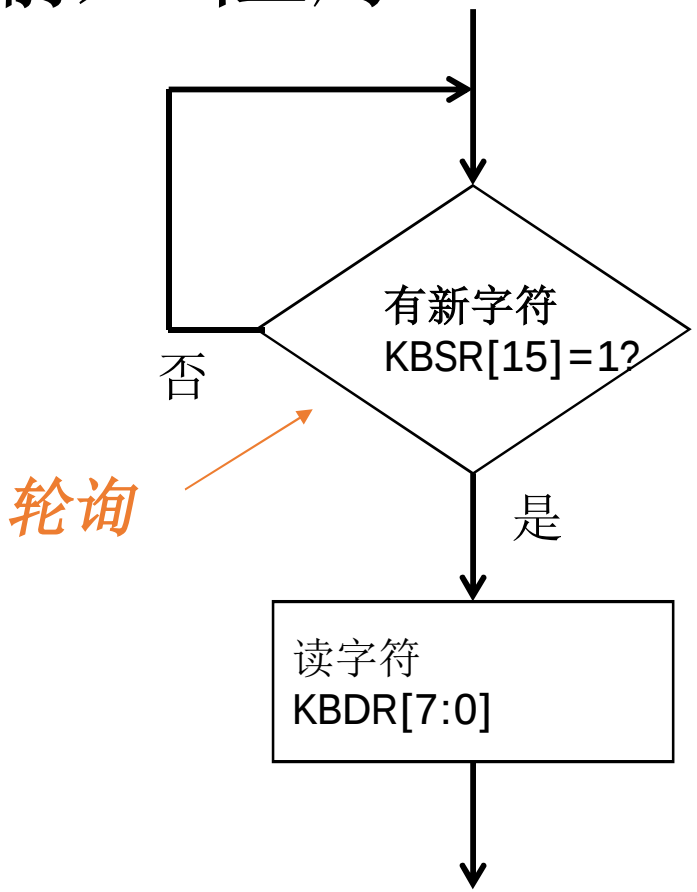
# LC-3键盘输入机制

- 当按下一个字符时:
  - 1)其ASCII码放置在KBDR的比特[7:0]位(比特[15:8]位总是0)
  - 2)“就绪位” (KBSR[15]) 被设置为1
  - 3)键盘被禁用– 任何输入的字符都会被忽略




- 读键盘数据寄存器(KBDR)数据:
  - 1）检测KBSR[15] 是否为1，为1则读取KBDR[7：0]
  - 2) KBSR[15] 设置为0
  - 3) 键盘置为使能,准备接收下一个字符

# 基本输入程序



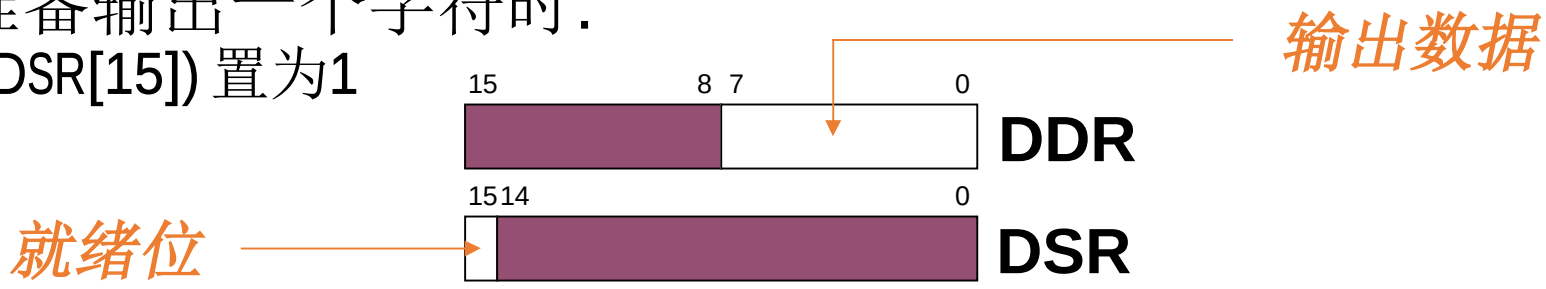

```
POLL    LDI  R0, KBSRPtr
        BRzp POLL
        LDI  R0, KBDRPtr

        ...

KBSRPtr .FILL xFE00
KBDRPtr .FILL xFE02
```

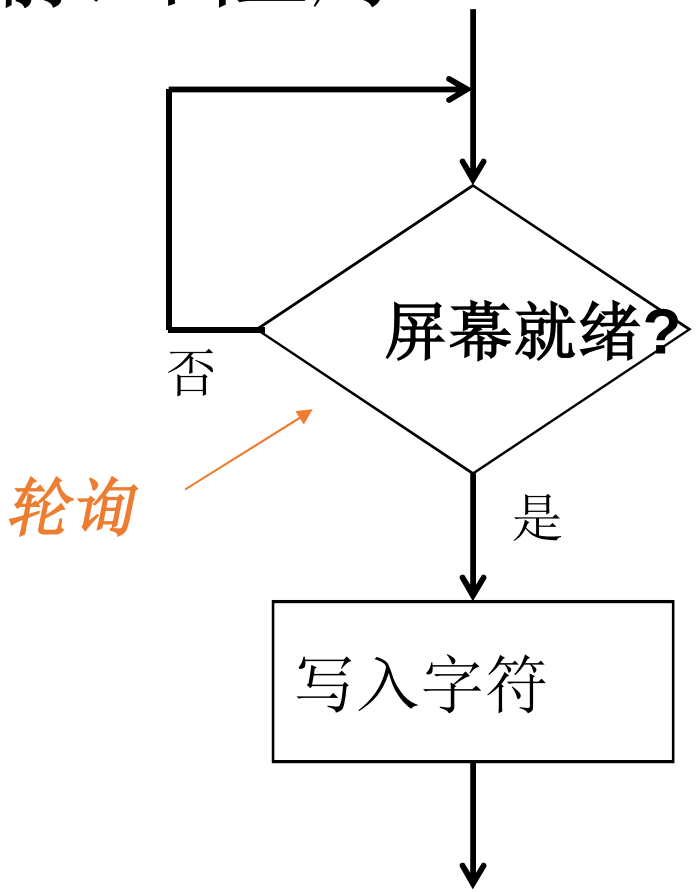
# LC-3显示器输出

- 当显示器准备输出一个字符时:
  - 就绪位(DSR[15]) 置为1




- 当就绪时,数据可写入输出数据寄存器:
  - DSR[15] 置为0
  - 写在DDR[7:0] 中的字符将被输出
  - 任何其他写入输出数据寄存器的字符将被忽略(DSR[15]=0期间)
  - 显示完成后,就绪位(DSR[15]) 置为1
  -

# 基本输出程序




```
POLL    LDI   R1, DSRPtr
        BRzp  POLL
        STI   R0, DDRPtr

        ...

DSRPtr  .FILL xFE04
DDRPtr  .FILL xFE06
```

```
01  START   ST      R1,SaveR1    ; Save registers needed
02          ST      R2,SaveR2    ; by this routine
03          ST      R3,SaveR3
04
05  ;       LD      R2,Newline
06          LDI     R3,DSR       ; Loop until monitor is ready
07  L1      BRzp    L1           ; Move cursor to new clean line
08          STI     R2,DDR
09
0A  ;       LEA     R1,Prompt    ; Starting address of prompt string
0B          LDR     R0,R1,#0     ; Write the input prompt
0C  Loop    BRz     Input        ; End of prompt string
0D          LDI     R3,DSR
0E  L2      BRzp    L2           ; Loop until monitor is ready
0F          STI     R0,DDR       ; Write next prompt character
10          ADD     R1,R1,#1     ; Increment prompt pointer
11          BRnzp   Loop         ; Get next prompt character
12
13  ;
14  Input   LDI     R3,KBSR
15          BRzp    Input        ; Poll until a character is typed
16          LDI     R0,KBDR      ; Load input character into R0
17  L3      LDI     R3,DSR
18          BRzp    L3           ; Loop until monitor is ready
19          STI     R0,DDR       ; Echo input character
1A  ;
1B  L4      LDI     R3,DSR
1C          BRzp    L4           ; Loop until monitor is ready
1D          STI     R2,DDR       ; Move cursor to new clean line
1E          LD      R1,SaveR1    ; Restore registers
1F          LD      R2,SaveR2    ; to original values
20          LD      R3,SaveR3
21          BRnzp   NEXT_TASK    ; Do the program's next task
22  ;
23  SaveR1  .BKLW   1            ; Memory for registers saved
24  SaveR2  .BKLW   1
25  SaveR3  .BKLW   1
26  DSR     .FILL   xFE04
27  DDR     .FILL   xFE06
28  KBSR    .FILL   xFE00
29  KBDR    .FILL   xFE02
2A  Newline .FILL   x000A        ; ASCII code for newline
    Prompt  .STRINGZ ''Input a character>''
```

图8-5 LC-3的键盘输入例程

# P140. LC-3完整的盘输入历程

# 基于中断驱动的输入/输出

- 为什么使用中断?
  - 轮询占用一定的处理器周期,
    尤其是对稀有事件– 这些周期可以用来处理更多计算.
  - 例子: 处理之前的输入，同时收集新的输入. (See Example 8.1 in text.)

- 基于中断,外部设备可以:

(1) 强制当前处理器执行的程序停止;

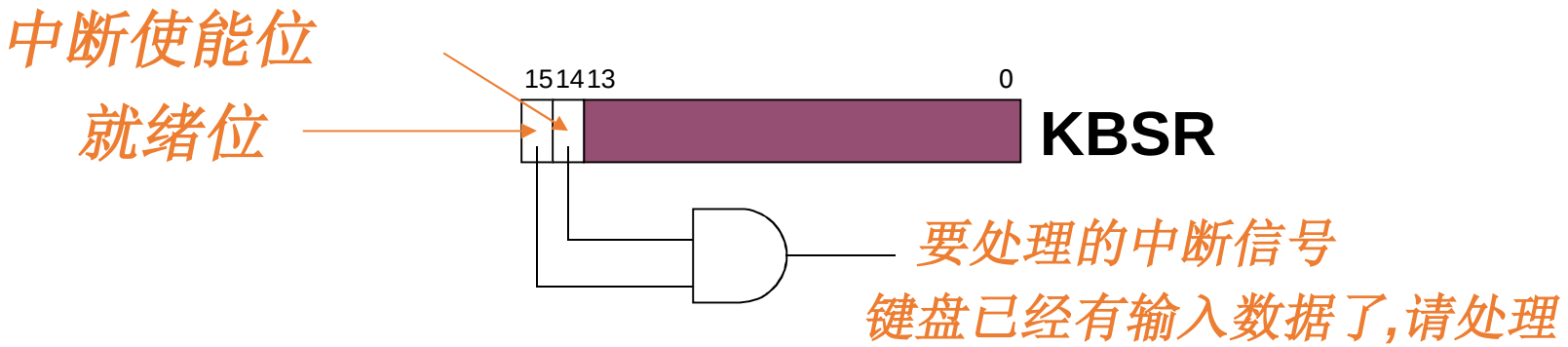
(2) 使处理器响应设备的需求;

(3) 完成后,处理器恢复停止的程序就像没发生过.

# 于中断驱动的输入/输出

- 为了实现中断机制:
  - 一种方式是I/O设备通知CPU一个感兴趣的事件发生了.
  - 一种方式是CPU去测试是否信号是否置位而且它的优先级是否比当前正在运行的程序高.

- 中断信号产生
  - 软件在设备寄存器中设置中断使能位 。中断使能位=0时设备不产生中断.中断使能位=1时设备可产生中断.
  - 当就绪位和中断使能位都置位时，产生中断信号.



# 优先级

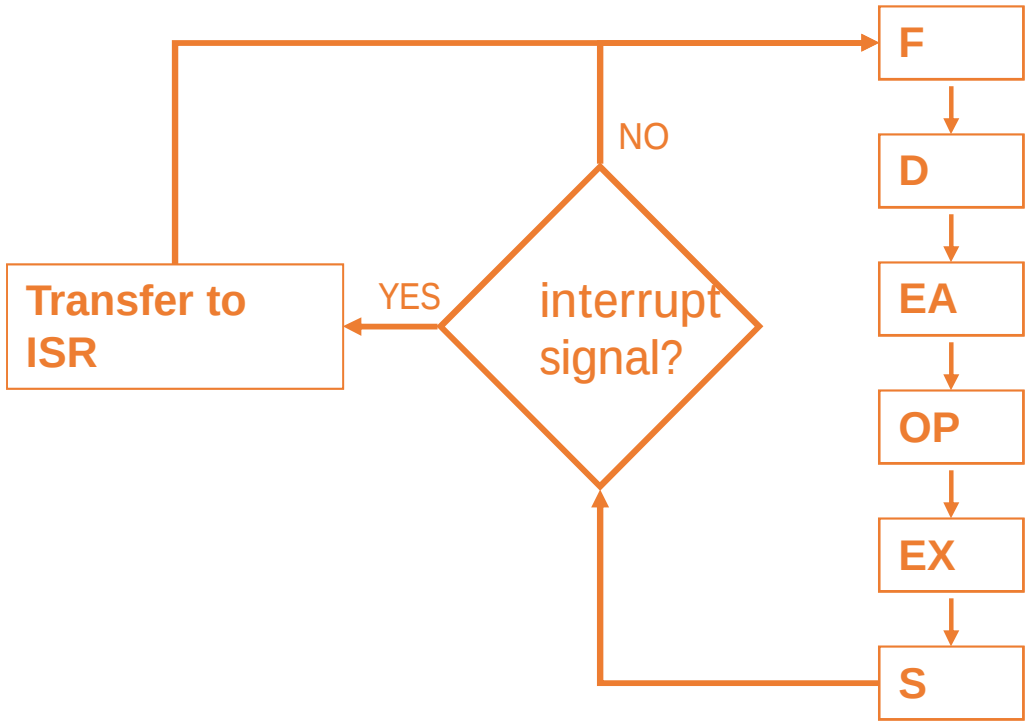
- 每一条指令在一个规定的紧急等级执行.
- LC-3: 8 优先等级 (PL0-PL7)
  - 例子:
    - 工资计算程序在PL0级执行.
    - 核能校正程序在 PL6级执行.
  - 运行在 PL6级的设备可以中断PL0级的程序 ,反过来却不行.
- 优先级编码器 优先级编码器同当前处理的程序的优先级相比较选择优先级最高的程序，如果允许的话将会产生中断信号.

# 断信号的检测

PU 在指令执行时S(写内存)阶段和F(取指)阶段之间检测是否有设中断信号 .(由硬件自动检测)

果没有中断置位, 继续执行下一条指令.

果有中断置位, 则把控制权交给中断服务程序.



# 系统调用

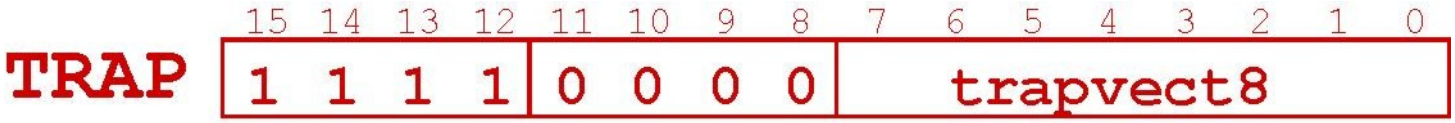
- 1. 用户程序使用系统调用
- 2. 操作系统执行调用操作
- 3. 结束后将控制权返回给用户程序

**在LC-3里，通过*TRAP* 机制来实现系统调用。**

# LC-3 TRAP 机制

- *1.包括一组服务子程序.*
  - 操作系统的一部分 – 服务程序起始固定的内存地址
    LC-3中实现的服务子程序位于系统代码区（below x3000)
  - 最多支持 256个服务子程序
- *2.起始地址表.*
  - 存放在 x0000 到 x00FF 的内存中(256x16bit) 。每16位存放一个系统服务子程序的起始地址。
  - 在别的系统中可能称为为“系统控制块”，或 “陷入矢量表”
  - *3. TRAP 指令.*
  - 用户程序通过TRAP指令来实现系统调用。操作系统将以用户程序身份执行某个特定的服务程序，并在执行结束后将控制权返回
  - LC-3: TRAP指令通过指令中8-bit的 trap vector来指示调用256个服务子程序中的哪一个。
- *4.链接回到用户程序.*
  - 提供从服务程序返回到用户程序的机制

# :AP指令




- Trap 向量
  - 指示调用哪个系统服务程序（x00-xff）
  - 通过8位trapvect8索引*起始地址表，获得对应系统调用的入口地址*
    - LC-3实现的方法：

      *起始地址表存放在内存的* 0x0000 – 0x00FF处，8-bit trap vector 通过高位0扩展成16位的内存地址，该内存地址处存放的就是相应调用的入口地址
- 如何执行
  - 从起始地址表查找服务程序地址，加载到PC中
- 如何返回
  - 将下一条指令的地址（当前的PC值）保存到R7

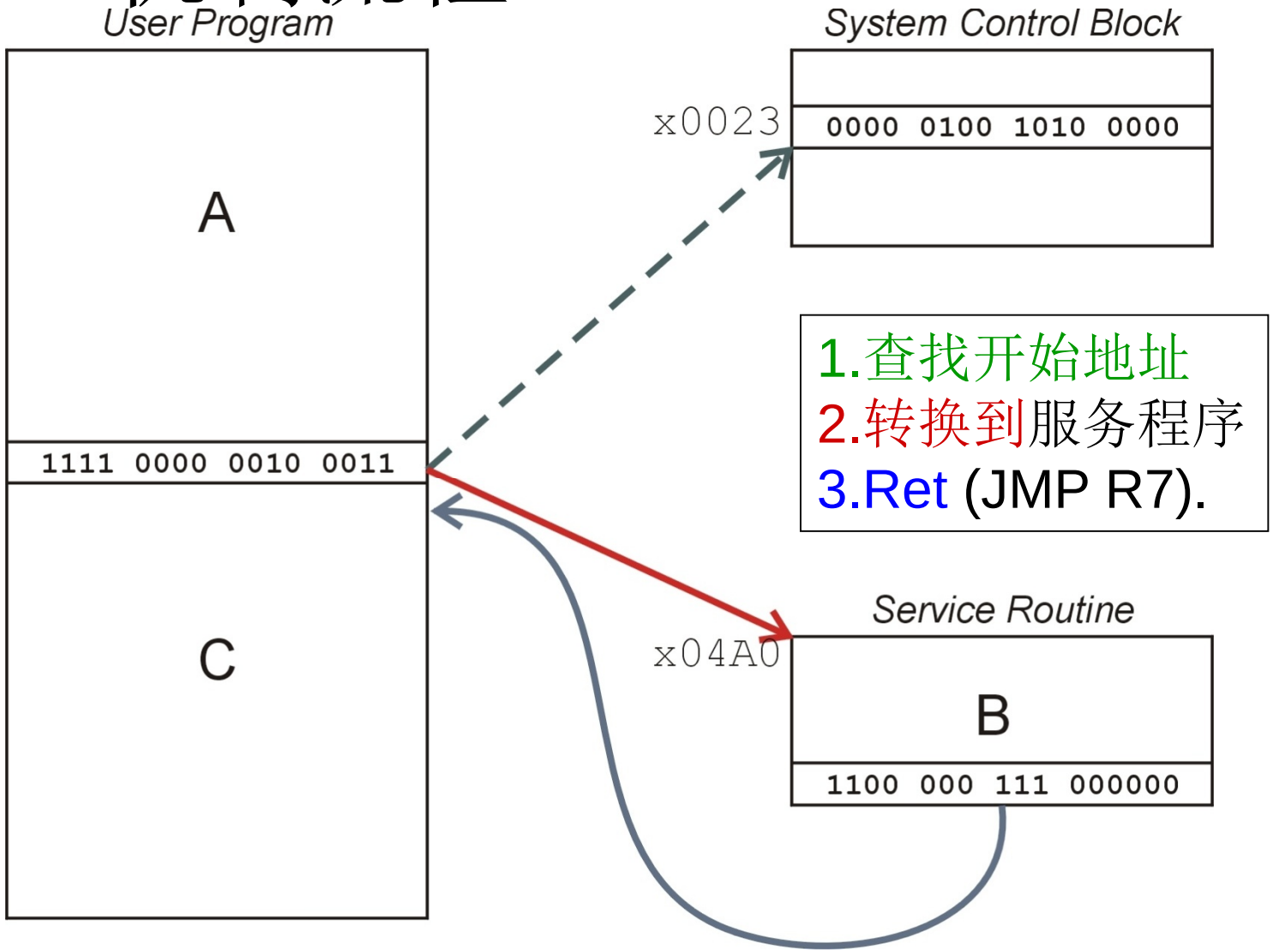
# ET （JMP R7）

•如何返回用户程序?即回到trap指令的下一条指令继续运行

•执行trap指令时将PC保存在R7
  - 服务程序使用JMP R7就可以返回到用户程序
  - LC-3 汇编语言使用 RET (return) 助记符来取代 “JMP R7”.

•因此：必须保证服务程序没有改变R7，否则无法返回.

# TRAP 机制流程


User Program
A
1111 0000 0010 0011
C
System Control Block
x0023
0000 0100 1010 0000
1.查找开始地址
2.转换到服务程序
3.Ret (JMP R7).
Service Routine
x04A0
B
1100 000 111 000000

# 存器内容的保存和恢复

- 被调用者保存-- *“callee-save”*
  - 在开始之前，保存可能被修改的寄存器内容（除非调用程序预先知道会被修改-EX. 参数）
  - 在返回前，恢复这些寄存器内容

- 调用者保存-- *“caller-save”*
  - 针对将被调用程序或被调用程序（如果知道的话）修改的寄存器，如果其内容后面需要，将其内容保存下来
    - 在TRAP之前保存R7
    - 在TRAP x23 (input character) 之前保存R0
  - 或者避免使用那些寄存器

- *保存在哪里：内存*

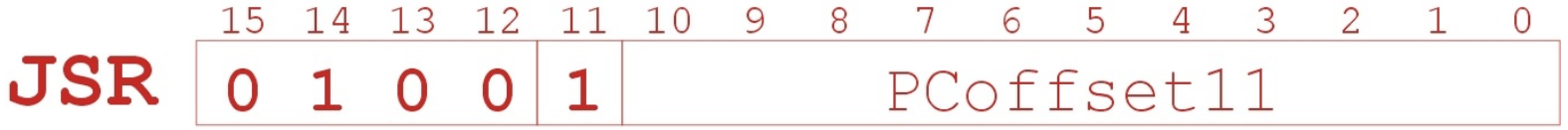
# 子程序

- 一个子程序是一个满足以下条件的程序片段：
  - 在用户空间运行 lives in user space
  - 执行一个预定义好的任务
  - 被另一个用户程序调用
  - 执行完毕后，将控制权返回给调用程序

- 与服务程序类似，但不是操作系统的一部分
  - 不与受保护的硬件资源打交道
  - 不需要特权

- 为什么需要子程序？
  - 重用有用的（调试好的！）代码
  - 在多个程序员之间划分任务
  - 使用供应商提供的代码库

# SR指令

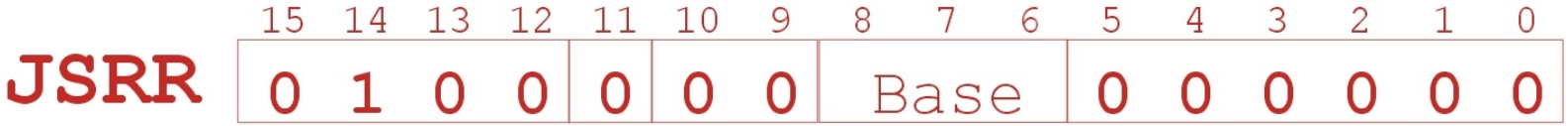



•保存当前的PC（下一条指令的地址）到R7，跳到一个指定的位置（PC相对寻址）。

- 保存返回地址叫做 “linking”
- 目标地址是 PC-relative (PC + Sext(IR[10:0]))  -1024~+1023
- 第11个bit位定义寻址模式：
  - 如果为1，PC相对寻址：目标地址 = PC + Sext(IR[10:0])
  - 如果为0，寄存器寻址：目标地址 = 寄存器内容IR[8:6]

能用BRnzp 指令取代吗？

# RR Instruction

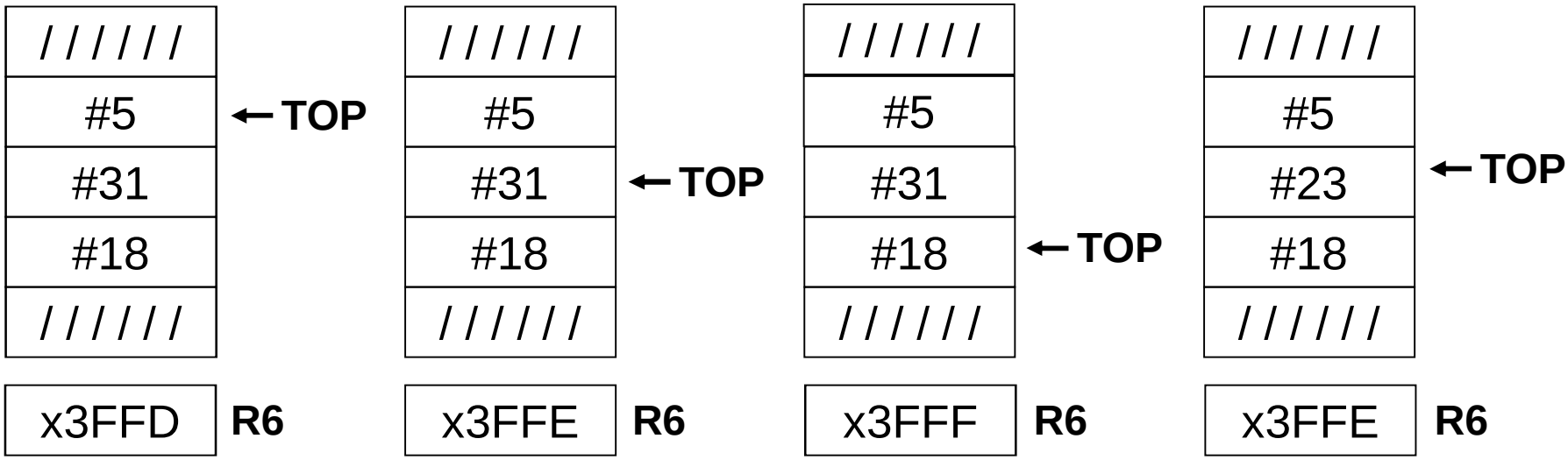



- 和JSR相似，除了寻址模式不同
  - 目标地址是寄存器基址
  - 第11个bit位定义了寻址模式

- 思考：JSSR具备的什么特征是JSR指令没有的？

# 内存中的实现：软件机制

- 弹出过程






假定用**R6**寄存器保存**TOP**指针，**R0**保存读出数据

# LC-3:基本 Push 和 Pop 操作的实现指令

- For our implementation, stack grows downward (when item added, TOS moves closer to 0)

- Push

```
ADD  R6, R6, #-1 ; decrement stack ptr
STR  R0, R6, #0  ; store data (R0)
```

- Pop

```
LDR  R0, R6, #0  ; load data from TOS
ADD  R6, R6, #1  ; decrement stack ptr
```

加入空栈和溢出检测的设计见**PP. 168-169**

# 断驱动I/O-结合堆栈的操作过程

1. 中断服务程序的启动:有来自设备的中断信号服务请求
2. 中断服务程序的执行:处理器保存相关状态信息，启动中断处理程序
3. 中断服务程序的返回:中断处理程序结束，处理器恢复相关状态信息并重启暂停的程序

# 处理器状态寄存器


- 运行模式
  - PSR[15] 指示当前程序是否运行在超级用户 Supervisor (0)状态还是普通用户 User (1)模式
  - 确保某些资源只允许被操作系统访问。

  linux: 内核空间和用户空间

  windows: 保护空间和非保护空间

- 优先级别
  - PSR[10:8] 保存当前程序的优先级别
    - 8个等级, 从 PL0 (最低)到 PL7 (最高).

- 状态码
- PSR[2:0] 保存当前的 NZP condition codes

# 在哪里保存程序状态？

- 能使用寄存器保存PC 和 PSR吗？
  - 程序员不知道什么时候会发生中断，不可能事先进行保存
- 在中断服务程序中分配内存?
  - 必须在进入中断服务程序前保存
  - 迭代和中断的嵌套：中断服务程序可能被高优先级的程序中断
  - PL0 (PL7->PL0)
- 保存时机

用户程序被中断处的指令执行完毕（存储阶段之后），在中断服务程序第一条指令的取 指令之前，必须由额外处理器硬件实现。TRAP在指令执行时候通过硬件保存PC到R7

- 解决方法：Use a stack!
  - 栈事先实现（硬件或操作系统）
  - Push state to save, pop to restore.
  - 先进后出特性方便支持中断嵌套

# Supervisor Stack

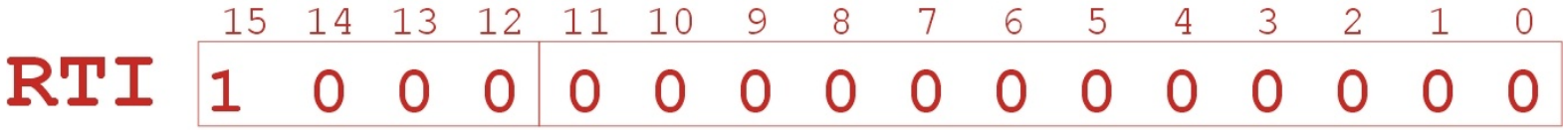
- 超级用户栈（Supervisor Stack）：内存中开辟的一个特殊的栈空间仅供特权模式下的程序使用，与用户程序栈空间隔离。
  - 指向超级用户栈的指针保存在内部寄存器Saved.SSP.
  - 指向用户栈的指针保存在内部寄存器Saved.USP.
  - Saved.SSP， Saved.USP类似MDR,MAR寄存器

- 所有程序都通过R6访问栈空间。

- 发生中断后，特权模式将从用户模式切换到超级用户模式，并将R6的内容保存到Saved.USP.

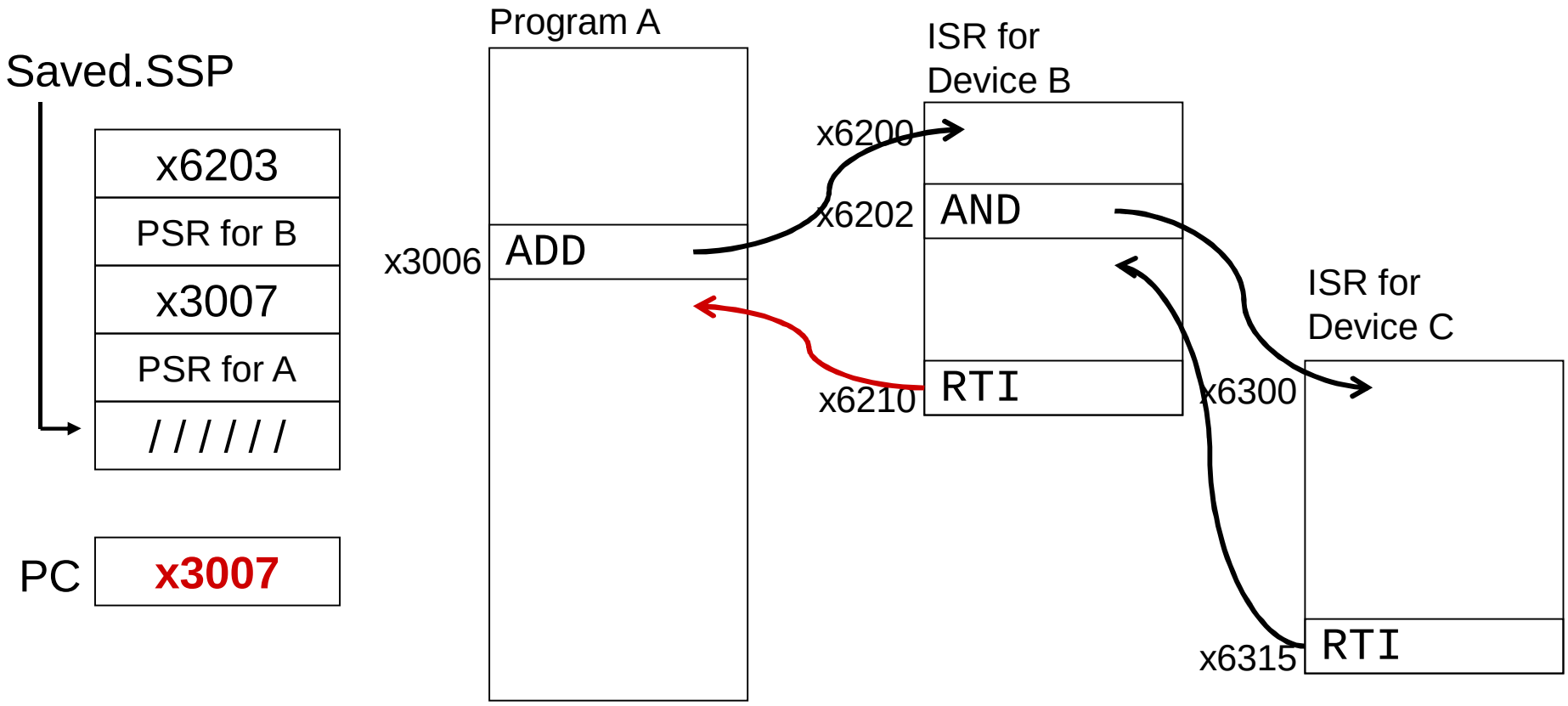
# 中断返回

- 中断返回指令 – RTI – 恢复中断前的用户状态




1. 将 PC从超级用户栈弹出 (PC = M[R6]; R6 = R6 + 1)
2. 将 PSR从超级用户栈弹出 (PSR = M[R6]; R6 = R6 + 1)
3. 如果 PSR[15] = 1, R6 = Saved.USP.
   (如果返回用户模式，需要重新加载用户栈指针；否则不改变栈指针内容)

# 了解嵌套中断的过程




执行RTI at x6210; 从栈恢复 PSR and PC
恢复 R6. 继续执行程序A,好像什么也没有发生.

# 课程考核方式

- 闭卷考试 （60%）
  - 共10道大题，每道大题或有若干小问（基本为每章1题）
  - 一道附加题（30分）
- 作业与试验（40%）
  - 作业占总成绩20分
  - 试验报告占总成绩20分（第4次试验占试验成绩的一半，即十分）

- 考试要求
  - 考试时间为2小时；
  - 考试提供p79页LC-3电脑ISA的复印件；
  - 不要轻易放弃，不留白卷。
  - 附加题为区分A与A+同学而设
    （若非附加题部分得分<85分，附加题得分毫无用处）

议：

真正花时间复习

仔细研究第3,5,7章的作业

自己找几个课本上稍复杂的例子仔细看看

祝大家期末考试取得好成绩!