Something important

- Addressability, address, address space

  Addressability是地址的*数量*，MAR的位数为n，则addressability最多为2^n;address space是地址存储的*数据的位数*（不是地址的长度！）

- **register&&memory**
  - 都需要地址来从中取信息
  - 数量不同，memory location有2^n个，register有有限个（LC-3中8个）
  - 寄存器是*CPU*的组成部份。寄存器是有限存贮容量的高速存贮部件，它们可用来暂存指令、数据和位址。

    内存是外存与CPU进行沟通的桥梁，计算机中所有程序的运行都在内存中进行, 其包含的范围非常广，一般分为只读存储器（ROM）、随机存储器（RAM）和高速缓存存储器（cache）。

    cache: 位于CPU与主内存间的一种容量较小但速度很高的存储器, 存着CPU刚用过或循环使用的一部分数据，当CPU再次使用该部分数据时可从Cache中直接调用,这样就减少了CPU的等待时间,提高了系统的效率.
- **condition codes的改变(important)**：执行任何写入或者运算操作都会改变，其他操作不会改变
- machine cycle: every stroke corresponds to a machine cycle

  clock cycle：Each of the repeated sequence of identical intervals

  有时候可以交换使用
- LC-3十进制输出
- why NZP in PSR
- J[5:0], COND[2:0], and IRD—ten bits of control signals provided by the current clock cycle.
- R to indicate the end of a memory operation.
- 注意栈帧（栈底指针）一开始应该是在底部再往下一个位置。如xFE00；系统栈帧一开始则是X3000
- state machine是一个概念还是一个实现?
- PSR:X8002 注意压栈的时候，先压PSR，再压PC; 因为RTI时系统会默认把先pop出来的值给PC
- privilege mode可以访问所有内存，user mode只能访问user space;在privilege mode下进行user的执行，不会报错
- the RUN latch is bit [15] of the Master Control Register (MCR), which is memory-mapped to location xFFFE.
- 大部分现代计算机使用中断驱动模式，因为轮询将大量降低CPU利用率
- **同步**：程序完全按照代码顺序执行

  The state transitions take place, one after the other, at identical fixed units of time.

  Controlled by a *synchronous finite state machine*.

  异步：程序的执行需要由系统事件来驱动，常见的系统事件包括中断、信号等（简单理解为：调用一个功能，不需要指代该功能最后执行的结果，该功能有结果后再通知我就行（回调通知）。也可以理解为：数据拷贝的时候，进程是否阻塞来作为同步和异步的区别）。（饮料机）

  No fixed time.There is nothing synchronizing when each state transition must occur.

在节奏不一致（异步）的时候，需要额外的synchronization mechanism来使其达成同步，如flag。如果本身就是同步，则不需要另外的同步设施，因为节奏一致，processor will exactly know what has happened.

不要把polling和interruped-driven混为一谈，两者概念并不同，polling和interrupted-driven是检测flag的方式

- interrupt: PL4

Abbreviation

- **OSH**:operating system help
- **SEXT**：Sign-Extension
- general purpose register（通用寄存器）
- CC: condition code
- BEN: indicate whether or not a BR should be taken
- **ACL**（Address Control Logic）
- **ACV**(Access Control Violation exception): An ACV exception occurs if a process attempts to access a location in privileged memory (either a location in system space or a device register having an address from xFE00 to xFFFF) while operating in User mode.
- **ADT**：Abstract Data Type
- **INTV**: interrupt vector (像trap一样，对应一个**Interrupt Vector Table** x0100-x01FF，trap则是x0000 to x00FF,)
- **Signals:**(Specially,inst[15:11], PSR[15], ACV, BEN, INT, and R)
  - R to indicate the end of a memory operation
  - BEN to indicate whether or not a BR should be taken.
  - .J[5:0], COND[2:0], and IRD—ten bits of control signals provided by the current clock cycle.
- **MOS**：metal-oxide semiconductor
- **CMOS**:complementary metal-oxide semiconductor
- **R**: indicate the end of a memory operation
- **LC**:location counter

Mistakes Collection

- 勿忘符号位
- 使用汇编时，imm不要越界，尤其是跳转时（LD...）
- SP==R6
- 默认先压PSR，再压PC；在使用RTI时，在一个指令中完成了两个元素的pop，不存在先pop了PC就直接跳转到另一指令这个说法。（猜测是gate还未开启）
- 区分NOR NAND的图示和在前面就加inverter的gate
- 画电路图时可以先用逻辑门写出来，再用晶体管实现
- **recursion**

```
FACT        ADD  R6,R6,#-1
            STR  R1,R6,#0   ; Push Caller's R1 on the stack, so we can use R1.
;
            ADD  R1,R0,#-1  ; If n=1, we are done since 1! = 1
            BRz  NO_RECURSE
;
            ADD  R6,R6,#-1
            STR  R7,R6,#0   ; Push return linkage onto stack
            ADD  R6,R6,#-1
            STR  R0,R6,#0   ; Push n on the stack
;
            ADD  R0,R0,#-1   ; Form n-1, argument of JSR
B           JSR  FACT
            LDR  R1,R6,#0   ; Pop n from the stack
            ADD  R6,R6,#1
            MUL  R0,R0,R1   ; form n*(n-1)!
;
            LDR  R7,R6,#0   ; Pop return linkage into R7
            ADD  R6,R6,#1
NO_RECURSE  LDR  R1,R6,#0   ; Pop caller's R1 back into R1
            ADD  R6,R6,#1
            RET
```

所有在subroutine中用到的非返回值都需要压栈

- MAR,MIO.EN,R.W determine the address control logic,which will affect the inmux

1. Abstraction、Hardware/Software、Basis

Abstraction

- omit the details
- 前提：细节、底层运转正常
- productive
- should be raised to high level
- "black box" model

Software/Hardware

- should not be divided
- If software developers can understand the hardware implementation, they can use its characteristics to write more efficient code.
- If hardware developers can understand the needs of the software, they can make optimizations that are more conducive to the software.

Computer System

- CPU(central processing unit)
- Memory: data, instructions(在disk与I/O设备交互)
- Disk(for limited storage of memory)
- peripheral device: I/O

Transformation

1. Problems

- natural languages
- no ambiguity

2. Algorithm

- definiteness（每一步都清晰可定义）
- effective computability（每一步都可被执行）
- finiteness

3. Language(Program)

- high-level languages: they are independent of the computer on which the programs will excute. "machine independet"
- low-level language: they are tied to the computer on which the programs will excute. There's generally one such low-level language for each computer. "assembly language for that computer"

4. ISA(instruction set architecture)

- specifies the interface between the computer program directing the computer hardware and the hardware carrying out those directions(link between software and hardware)
- h.l.l->ISA: compiler（编译器）

  assembly language->ISA: assembly(汇编器)
- specify：the number of opcodes，data type(acceptable operands), addressing mode（寻址模式，定位各种操作数的不同方法，在内存中寻找特定字节）,addressability
- eg. x86,ARM

  IBM：Power PC，Z-series

5. Micro Architecture

- The implementation of ISA. Generally, one microarch only supports *one* ISA, but one ISA can be supported by *many* microarchs. These depend on the cost/performance tradeoffs.

  Microarch has many components, in this course, we only discuss the CPU
- eg.INTELP5，AMDK5
- microprocessors: Xeon ,Pentium(奔腾)

6. Circuit

7. Device(electron)

2. Bits， Data Types, Operations

Bits

- absence & presence
- voltage level
- A.B==A AND B, A'=NOT A, A+B==A OR B

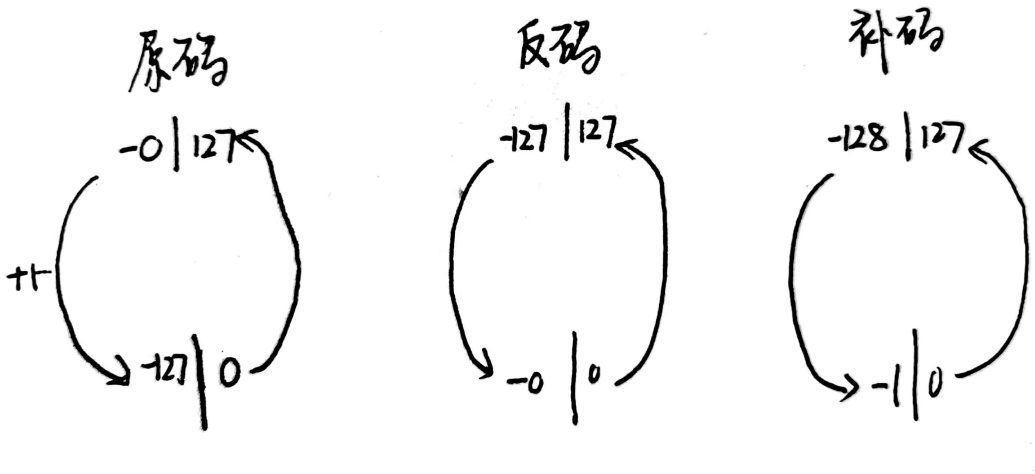
> Operations on Bits:
>
> - Arithmetic
>   - extension:
>
>   sign extention:加符号位的扩展
>
>   zero extention:加0
>   - Overflow：同号数相加溢出，结果错误；异号数相加永远不会溢出（永远在范围内）
>   - n位全为1： 2^n-1
> - Logical Operations
>   - AND, OR, XOR, NOT, NAND（至少一个为0）,NOR（全为0）, DeMorgan's Laws
>   - Bit Vector: an m-bit pattern where each bit has a logical value (0 or 1) independent of the other bits (usually used in bit mask)

Data Types

Integers

- unsigned,signed
- signed-magnitude（符号位表示法，原码）, 1' s Complement, 2 's Complement

> 2's complement:
>
> $$range : [-2^{n-1}, 2^{n-1} - 1]$$
>
> - 原码转补码：反码+1
>
> 补码转原码：取反+1（符号位不变）
>
> - 以8bits为例，-128的补码为1000 0000，-128没有原码和反码，超出了表示范围，规定补码就是如此
>
> 符号位表示法（原码）和反码都出现了-0，而补码则将-0替代为-128
>
> 反码：原码符号位不变，其他位置取反



Floating Points

- 左移右移
- fraction->binary representation

ASCII Code

Hexadecimal Notation

Floating Points

	S	exponent	fraction
32-bits	1	8	23
64-bits	1	11	52

FADD：浮点数相加

ADD：补码相加

> Decimal to floating points
>
> - 写成二进制
> - 左移或右移，使小数点左边只有一个“1”，得出指数
> - 阶码=127（1023）+指数（exponent is unsigned，以127为原点以表示比0小的指数）
> - 加符号位，尾数照写

> Floating points to decimal
>
> - 符号位
> - 计算阶码，指数=阶码-127
> - 将二进制小数形式表现出来，转化为二进制

> 特殊情况：
>
> - E 全为 0。这时，浮点数的指数 E 等于 1-127（或者 1-1023），有效数字 M 不再加上第一位的 1，而是还原为 0.xxxxxx 的小数。这样做是为了表示 ±0，以及接近于 0 的很小的数字。

> *subnormal*: 为什么要还原为0.xxx： 因为如果第一个位一定有1，那么在阶码全为0时，第127位一定是1，就被限定死了，所以还原为0.xxxx；也正是因为这个原因，当阶码和位数都为0时表示的是0
>
> - E 全为 1。这时，如果有效数字 M 全为 0，表示 ± 无穷大（正负取决于符号位 s）；如果有效数字 M 不全为 0，表示这个数不是一个数（NaN）。

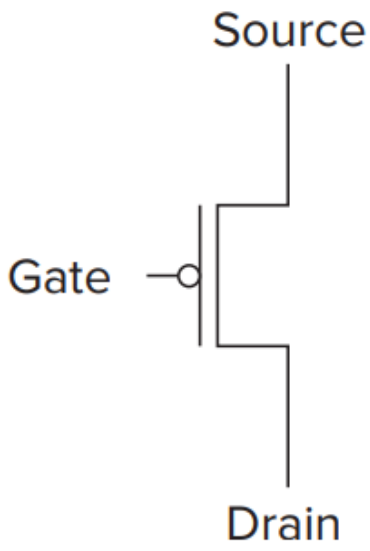
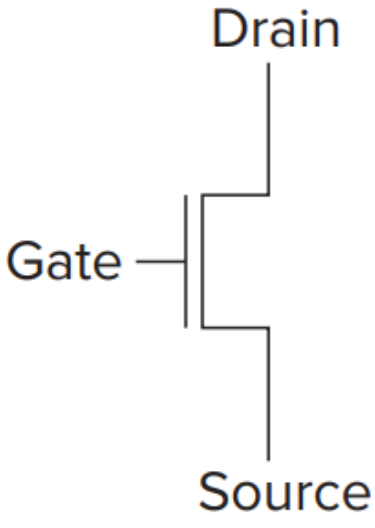
> 小数转二进制：
>
> 乘2，取整数位，顺序取，直到为0
>
> 指数：
> -126-127(全1和全0为特殊)
>
> 指数部分中间数：
>
> 2^(n-2)-1

3. Digital Logic Structures

Transistor

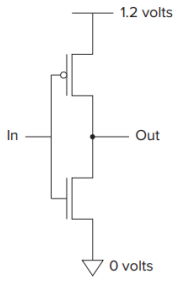
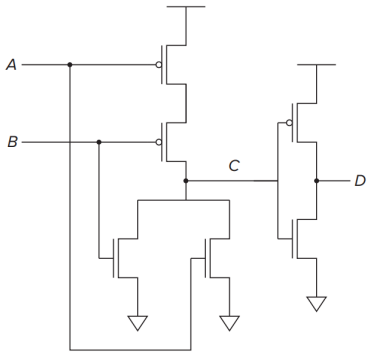
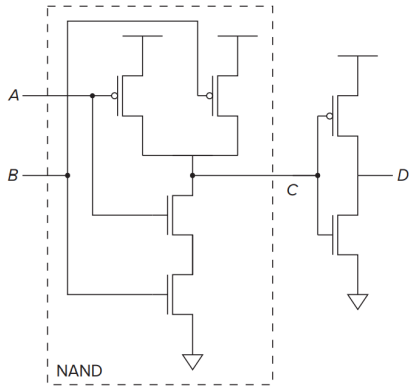
- MOS transistors(metal-oxide semiconductor)
- Gate,Drain,Source（给Gate提供电压）

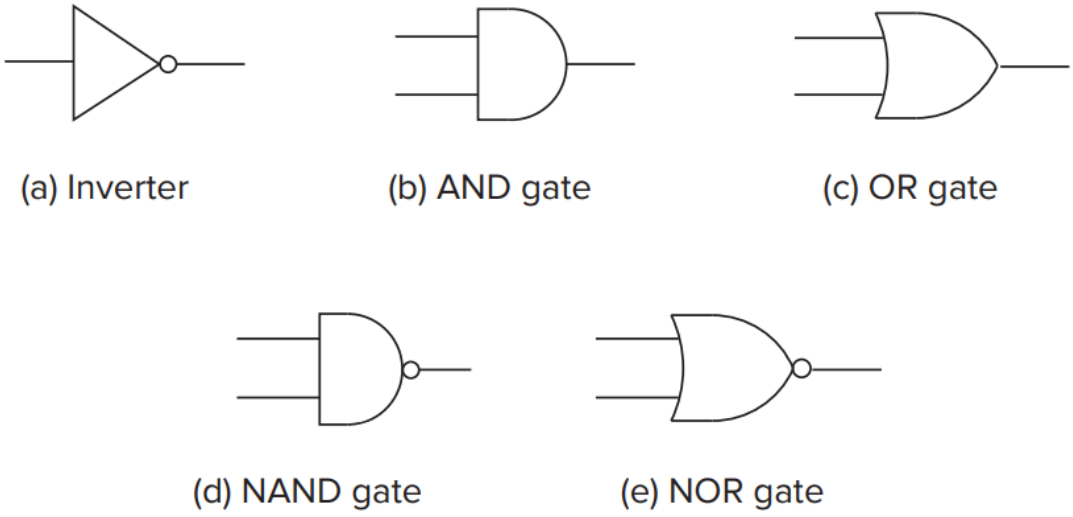
types	1.2V	0V
P-type	open circuit	close circuit
N-type	close circuit	open circuit

P-type	N-type
	

Logic Gates

要形成通路，必须把接线部分断掉

NOT Gate(Inverter)	OR/NOR Gate(P串联，N并联)	AND/NAND Gate（P并联，N串联）
		

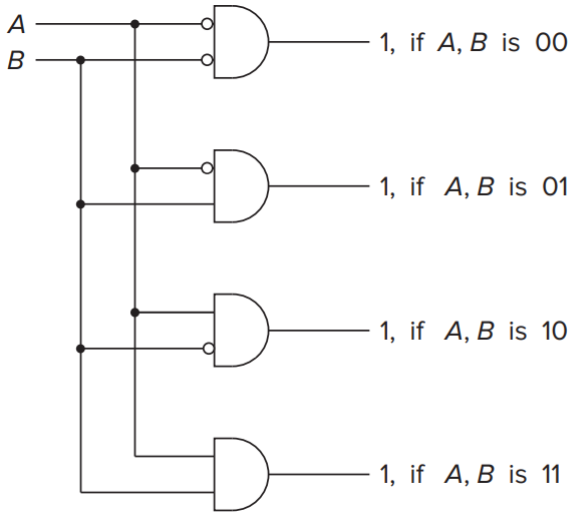


Combinatorial Logic Circuits

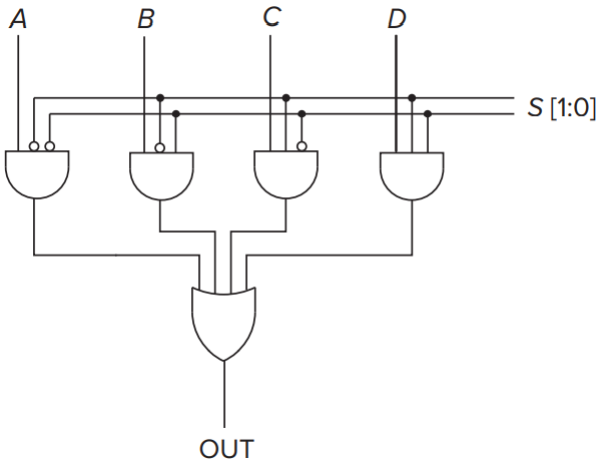
decision elements, depend on the combination of input values *right now*

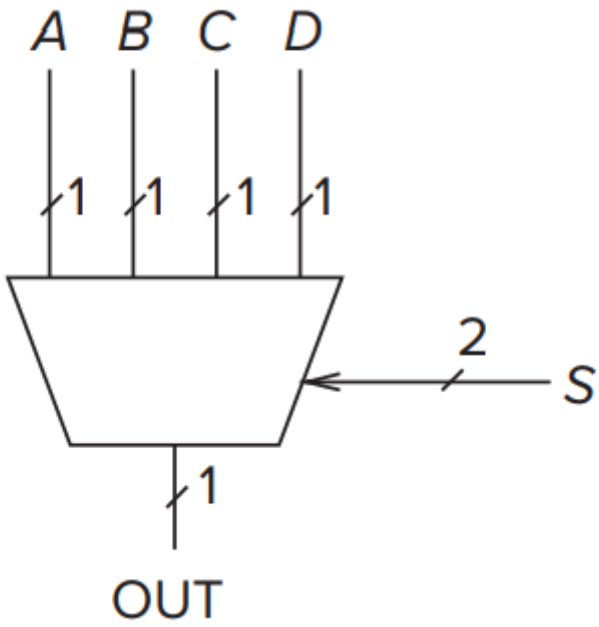
cannot store anything

- **Decoder**
  - asserted（置位）
  - exactly one output is 1;The one output that is logically 1 is the output corresponding to the input pattern that it is expected to detect.（AB输入为特定组合时，结果才为1）
  - n inputs, 2^n outputs.



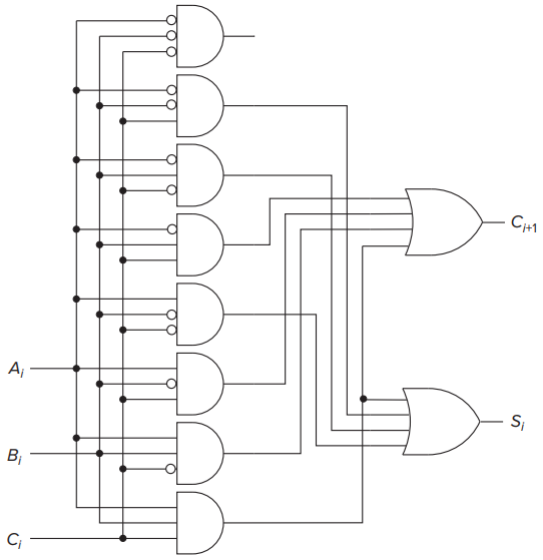
- **Mux**
  - select one of the inputs and connect it to the output.
  - The *select signal* determines which input is connected to the output
  - The number of select lines: ceiling function(log2n)
  - name: m-bits n-to-1 mux(n条线选1个，每条线有mbits)
  - 注意顺序：从左到右依次递增



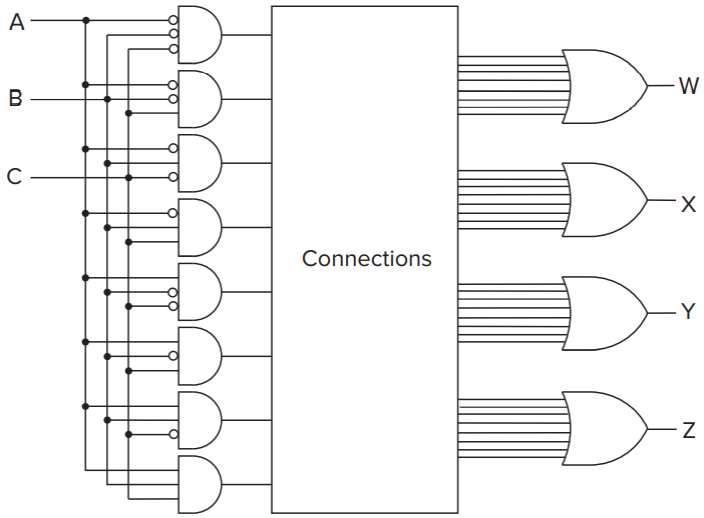


- **A One-Bit Adder**

- A，B为第i位，Ci为进位(carry)，Ci+1为产生的进位，Si为第i位的结果（看真值表，将可以产生进位的与门和Ci+1连接；将在当前位留下1的的与门和Si连接）



- **The Programmable Logic Array (PLA)** //PAL:Programmable Array Logic 由PLA、输出电路等组成
  - we program the connections from AND gate outputs to OR gate inputs to implement our desired logic functions.
  - W,X,Y,Z是自己设定的output，理论上最多有2^(2^n)个（真值表+对应值）



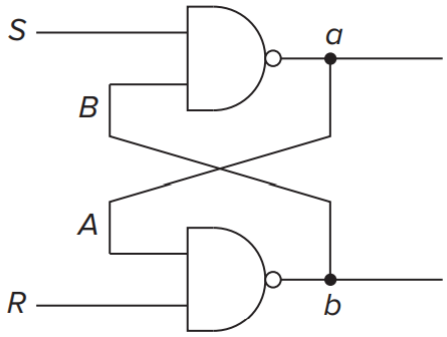
Basic Storage Elements

- **The R-S Latch**
  - store one bit of information(a端或b端是输出端，默认为a)
  - The Quiescent State: 当R、S都被置位1，状态不变
  - Setting the Latch to a 1 or a 0: 短时间内将S置为0，可以将a从0变为1；短时间内将R置为0，可以将a从1变成0

    set: set a variable to 0 or 1

    clear: set a variable to 0

    不可以将S和R同时置为0

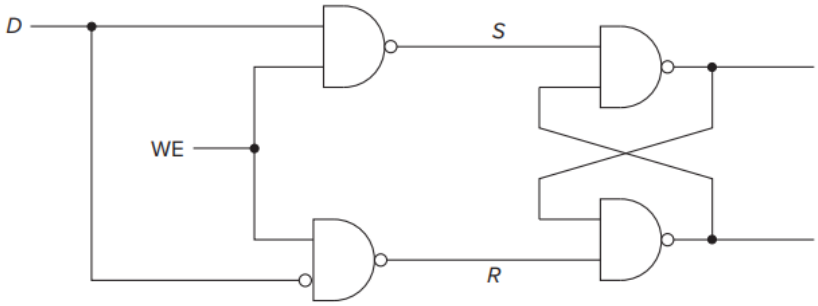


- **The Gated D Latch**
  - WE: write enable

    asserted(set to 1):allow the latch to be set to the value of D（可以被改变）

    not asserted(set to 0): S&R equal to 1
  - WE is momentarily set to 1: exactly one of S or R is set to 0

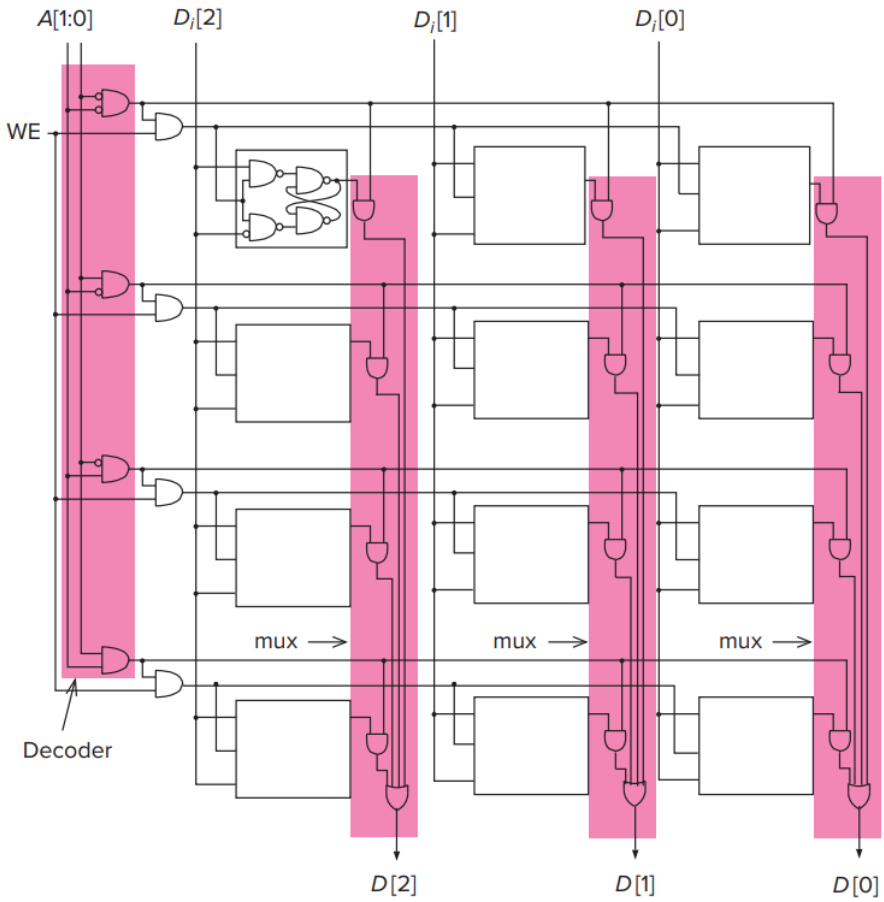
    D==1, S==0, a->1; D==0, R==0,a->0



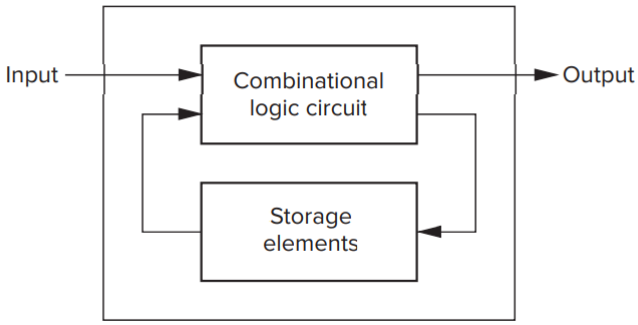
- **Logical Completeness**

Memory

- **Address Space**
  - the total number of uniquely identifiable locations （地址数量）
- **Addressability**
  - The number of bits stored in each memory location （每个地址对应的位数）
- **A 22-by-3-Bit Memory**



Sequential Logic Circuits



- can both process information and store information.
- base on the input values now and what has happened before
- The **state** of a system is a snapshot of all the relevant elements of the system at the moment the snapshot is taken.
- **The Finite State Machine（FSM） and Its State Diagram**
  - five elements:

    a finite number of states

    a finite number of external inputs

    a finite number of external outputs

    an explicit specification of all state transitions

    an explicit specification of what determines each external output value
  - **Draw FSM**
- **Types**

- asynchronous （异步）

  No fixed time.There is nothing synchronizing when each state transition must occur.

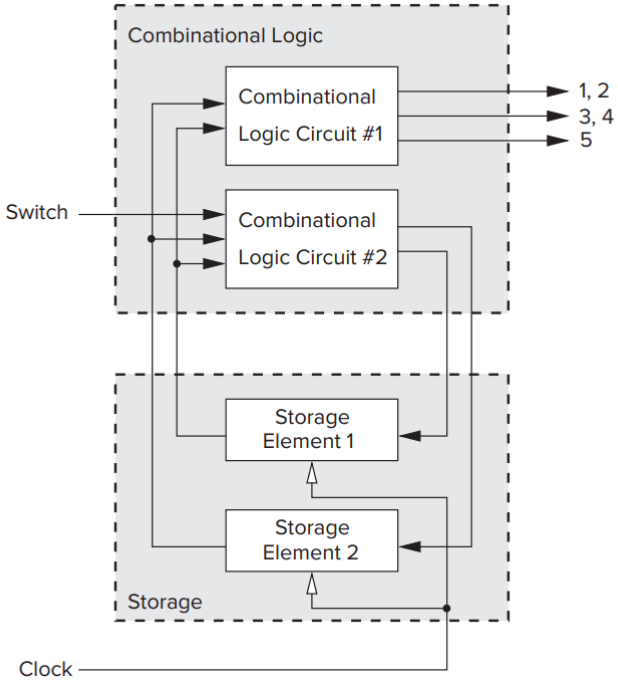
  （例：可乐贩卖机）

- synchronous（同步）

  The state transitions take place, one after the other, at identical fixed units of time.

  Controlled by a *synchronous finite state machine*.

- **The Clock**
  - between 0 volts and some specified fixed voltage(logically, o or 1)
  - Each of the repeated sequence of identical intervals is referred to as a *clock cycle*（时钟周期）
- **Samples**
  - State Variables: s1,s0



A sample: warning lights

① OFF
② 1,2 亮
③ ~~1~~1,2,3,4 亮
④ 1,2,3,4,5 亮.

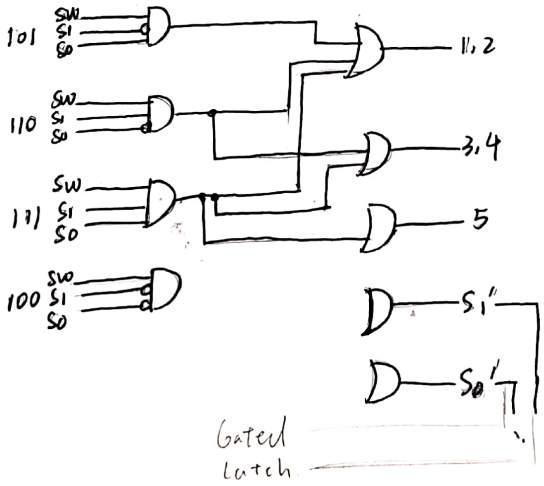
注意：switch off 时，全部不亮；switch on 时，也有灯off状态.

Combinational Logic：根据真值表设计对应电路实现状态转换.

state variables

Switch	S_1	S_0	1,2	3,4	5	STATE
0	0	0	0	0	0	00
0	0	1	0	0	0	00
0	1	0	0	0	0	00
0	1	1	0	0	0	00
1	0	0	0	0	0	00
1	0	1	1	0	0	01
1	1	0	1	1	0	11
1	1	1	1	1	1	00

只有这5个灯会亮



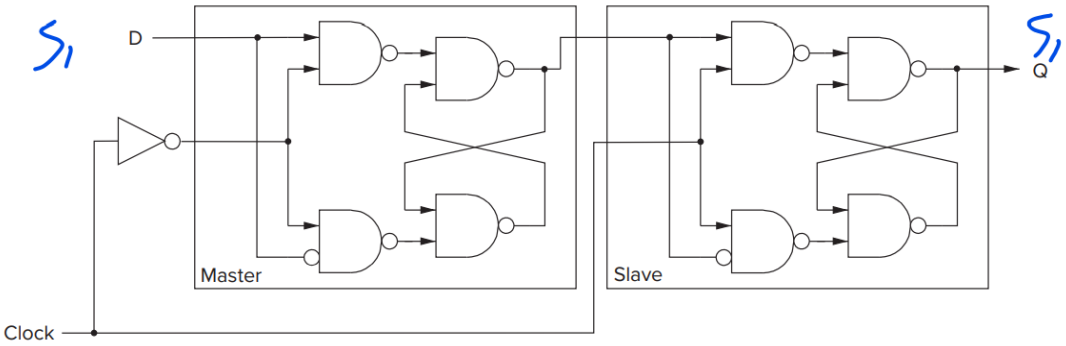
电路图解释：
1. 对应亮灯状态以或门传至灯处
2. S_1, S_0 表示storage，是上一个状态的存储图未画完，S_1' 与或门左连接使 S_1 状态为1的线路，即110和111.

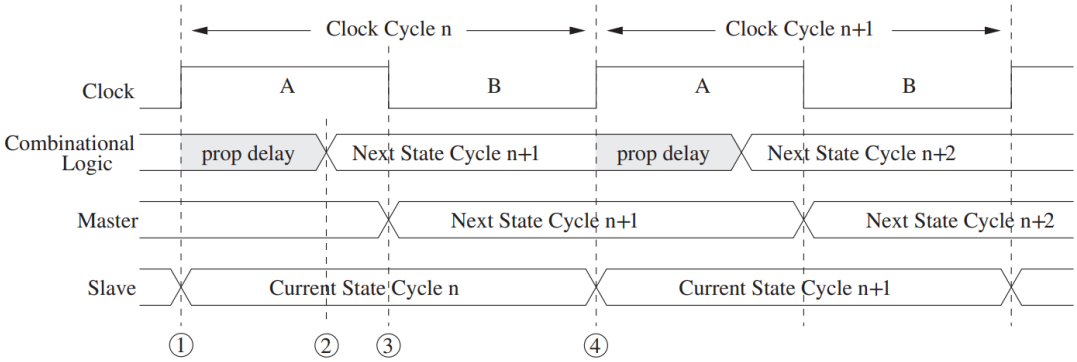
The storage element: Why not Gated Latch?

当Gated Latch is asserted, 输出随着输入改变，而输入又随着输出改变，则状态不停改变，则在当前clock style中，状态会变化得非常快；同时，我们无法得知下一个状态。

原因在于，我们希望output可以改变下一个状态，但又希望在一个周期内当前状态不要被改变；然而Gated D Latch会随着output立刻改变（不会等待到完全结束），导致状态不断改变

Solve: the master/slave Flip-Flop





- 2 latches, use clock to be WE.
- allow us to:
  - read the current state throughout the current clock cycle
  - not write the next state values into the storage elements *until* the beginning of the next clock cycle
- Process:（注意master的WE是非Clock, slave的WE是Clock）
  - S0',S1'对应或门Y,Z，产出output，作为初始与门的输入（此处只讨论其中一个）
  - 输入后，在A阶段，combinational logic进行运算产生delay，运算结束后产生新的output状态；由于时钟此刻为1，即master的WE是0，新的状态无法更改；
  - 在B阶段，时钟为0，master的WE为1，master状态被更改；slave的WE是0，无法更改
  - 在新的A阶段，slave切换到下一个状态；在每个周期内，slave的状态保持不变(即存储结构的output不变)

4. The von Neumann Model

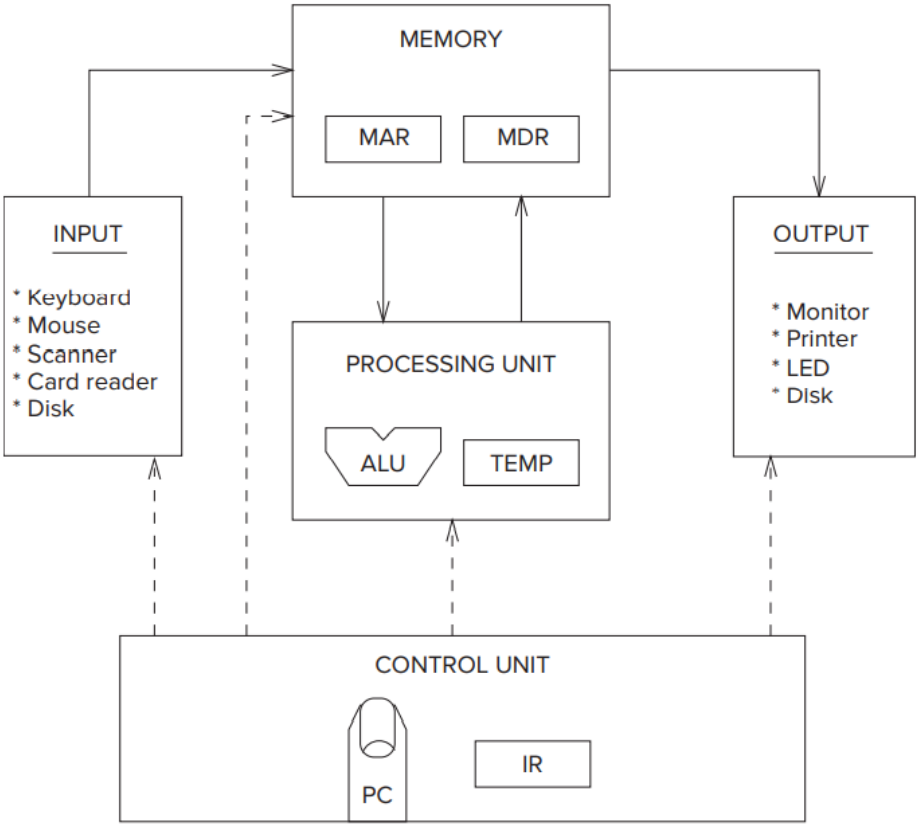
Terminology

- **words**: data elements(数据元)
- **word length**: the fixed size of data elements that the ALU normally processes（Each ISA has its own word length）
- **IR（instruction register）**：keep track of which instruction is being executed
- **PC(program counter)/IP(instruction pointer)**: contains the next instruction's address
- **register**:a set of n flip-flops that collectively are used to store one n-bit value.(in this book,PC, IR, MAR, and MDR are all 16-bit registers that store 16 bits of information each)
- **MAR(memory's address register)**:for addressing individual locations
- **MDR(memory's data register)**:) for holding the contents of a memory location on its way to/from the storage
- **machine circle**: each stroke correspond to a machine circle

Basic components

(a) computer program: instructions(smallest)

(b)computer(von Neumann model): memory, a processing unit, input, output, a control unit



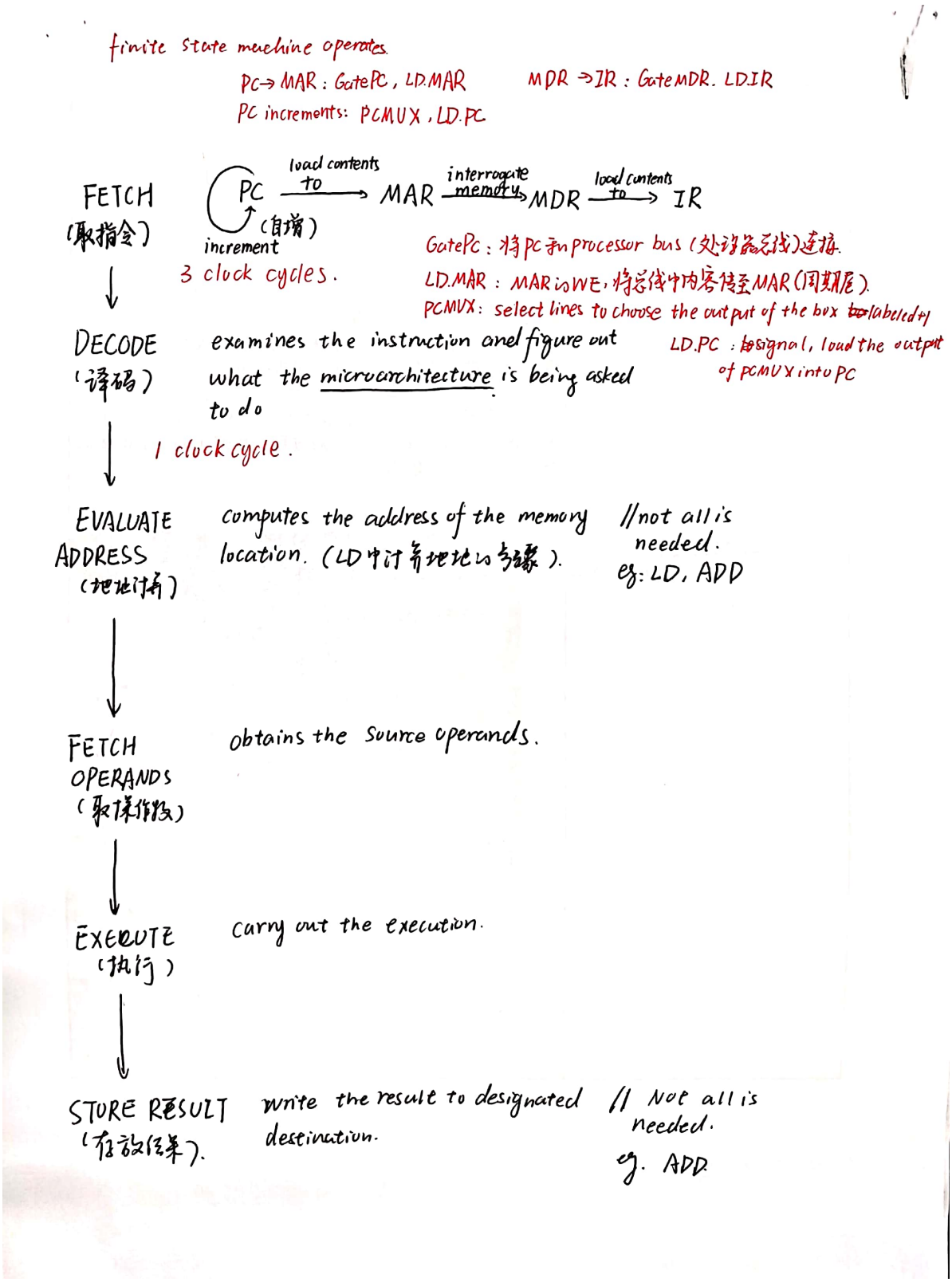
- **memory**
  - read:
    - place the address of that location in the **MAR**
    - interrogate the computer's memory(访问)
    - The information will be placed in the **MDR**
  - write:
    - write the address of the memory location in the MAR, and the value to be stored in the MDR
    - interrogate the computer's memory with the write enable signal asserted
- **Processing Unit**
- Carry out the actual processing of information in the computer
  - 最常见&&最简单：ALU（Arithmetic and Logic Unit）
- **Input and Output**
  - peripherals
- **Control Unit**
  - IR, PC(IP)

LC-3

- **memory**: MAR(16 bits), MDR(16 bits) //LC-3: address space:2^16; addressability:16
- **input/outout**: keyboard data register (KBDR) ; keyboard status register (KBSR);display data register (DDR);display status register (DSR)
- **processing unit**: ALU（1 arithmetic(ADD) and 2 logic(AND,NOT) operations);8 registers(R0-R7)
- **control unit**: finite state machine(CLK&&IR are inputs); PC

Instruction Processing

- **The Instruction Cycle**
  - 2 parts: opcode, operands
  - 3 kinds:
    - operates: operate on data
    - data movement: move information between memory and the registers and between registers/memory and input/output devices
    - control: change the sequence of instructions that will be executed
  - instruction cycle: The entire sequence of steps needed to process an instruction
  - addressing mode: Base+offset Mode; Indirect mode; PC-relatice mode
  - phase: 6 sequential phases in the instruction cycle(FETCH阶段中的MAR->MDR可能需要多个clock cycles, 由访问内存的时间决定)
  - 3 ADD instruction does not require a separate EVALUATE ADDRESS phase or a separate STORE RESULT phase. The LC-3 LD instruction does not require an EXECUTE phase.



Changing the Sequence of Execution

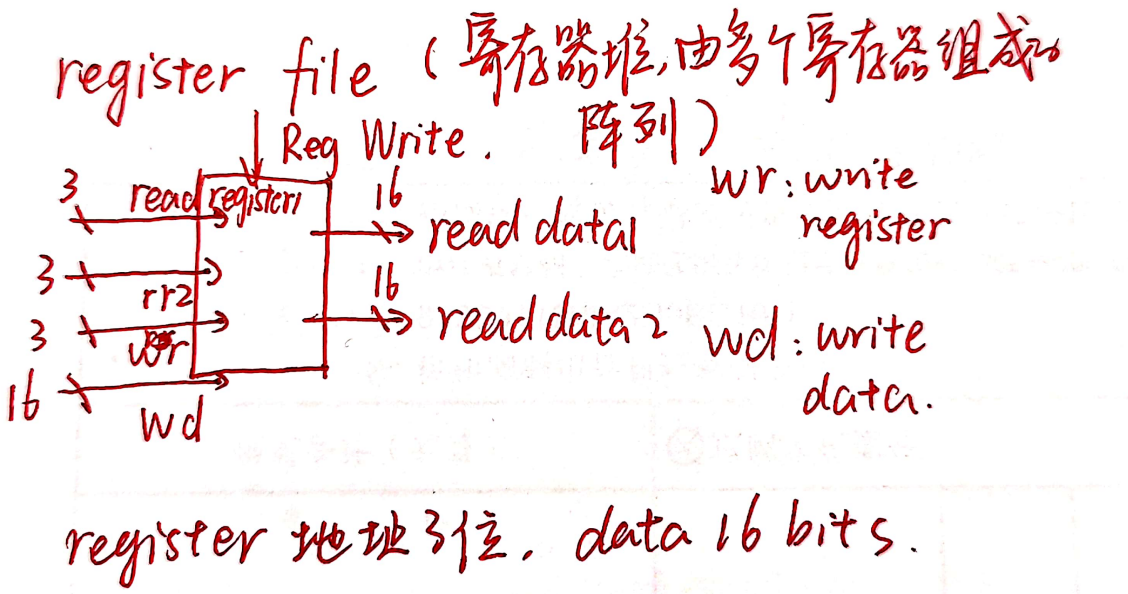
- 若没有改变顺序，程序每次执行后，PC中都会存储下一个指令的地址（PC+1）；因此要改变执行顺序，就要在FETCH阶段将PC的值改变，此处使用**control instruction**,它使得PC存储的指令地址取决于EXCUTE阶段的结果，而不是直接PC+1
- most common control instructions: **conditional branch(BR)**

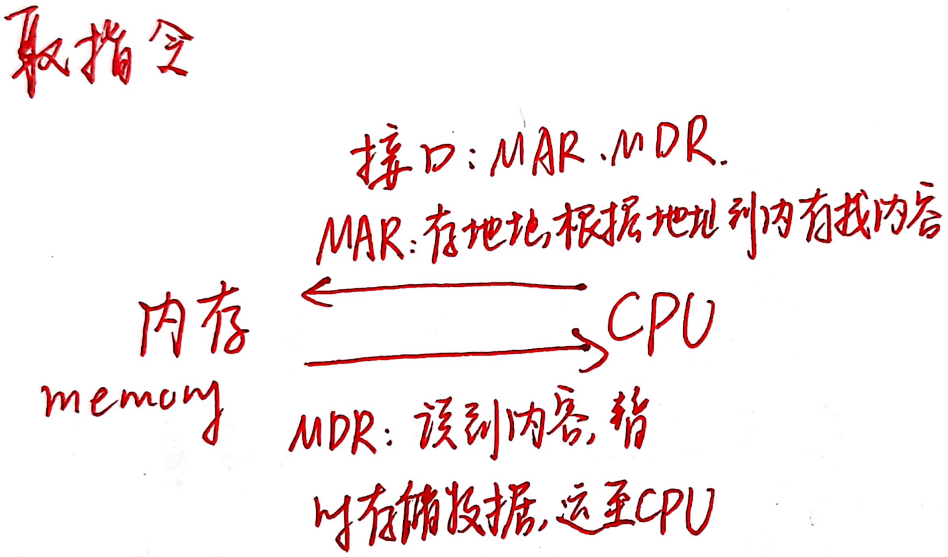
BR instruction

- **Halting the Computer (the TRAP Instruction)**
  - clock： defines the amount of time each machine cycle takes.
  - stopping the instruction cycle requires only clearing the RUN latch
  - TRAP instruction
- **example: a Multiplication Algorithm**

5. The LC-3

overview





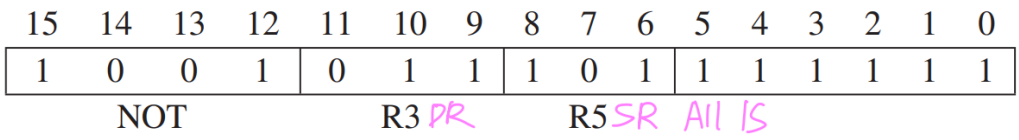
- **memory**
  - address space: 2^16
  - addressability: 16 bits

    - we refer to 16 bits as one word, and we say the LC-3 is word-addressable
- **register**
    - temporary storage locations: most used--a set of registers**(GPR, general purpose register)**
    - 8 GPRs, identified by a three-bit register number(R0-R7)
- **register file**:register set(usually shown in a table)
- **The Instruction Set**
    - defined by its set of **opcodes**, **data types**(representation of operands), **addressing modes**(determine where the operands are located)
- **opcode**
- 15 instructions, each defined by its unique opcode
- **Addressing mode**
    - operands can be found in memory, register, a part of instruction(called **literal/immediate operand**)
    - 3 modes: **PC-relative, indirect, Base+Offset**
- **Condition Codes**
    - 3 single-bit registers（**N,Z, P**）
    - 只要某位为1则会被examine, 为0不会被examine（且条件成立是or，不是and）
    - 每次执行完毕后都会被更新

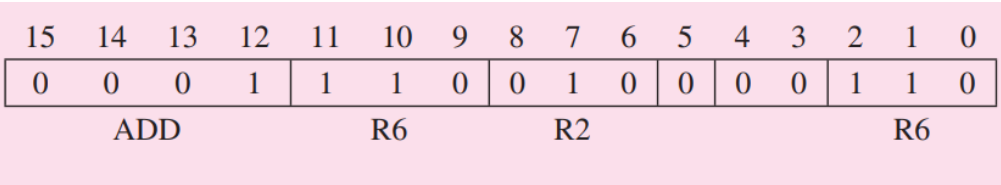
Operate Instructions

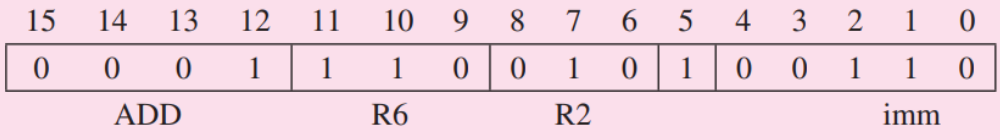
Operate

NOT



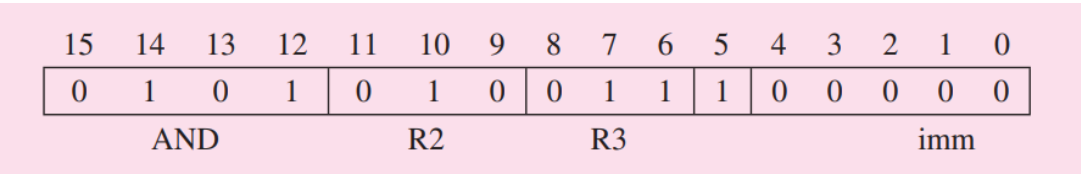
ADD





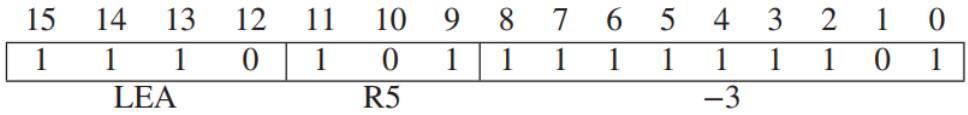
- [11:9] DR
- [8:6] SR1
- [2:0] SR2/imm
- imm: immediate value
- 第五位用来区分两种加法。当第五位为0，执行第一种加法，即两个地址对应的操作数相加并存储到对应的地址中；当第五位为1，执行第二种加法，后5位为数字（sign-extending），将该数字加到地址对应操作数中，并将结果存到对应地址中

AND



和add类似，也有两种；当第五位为0，则两个取AND并存到R2；当第五位为1，则将imm和R3取and，并存入R2（注意当imm==0，可以将R2清空）

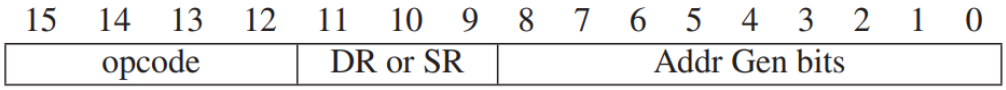
LEA(Load Effective Address)



将PC内容和offset相加得到新地址并将其存入R5

PC+1+offset(PC已经自增)

Data Movement Instructions



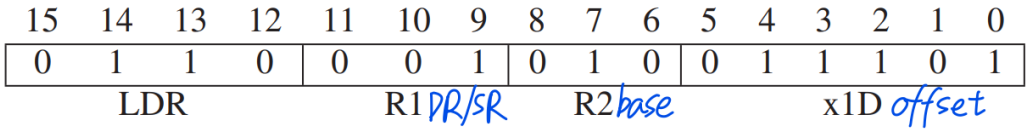
PC-Relative Mode :LD(load 0010)/ST(store 0011)

- 11:9: DR(LD) 加载-把地址中的数据传给寄存器 /SR(ST) 存储-把寄存器中数据写入地址
- 8:0：offset

Indirect Mode:LDI(load indirect 1010) /STI(store indirect 1011)

LDI：把内存中内容读到存储器中。 首先，R3是DR，先把**8:0表示的值（sign-extension)加上PC中地址**存到R3（MAR）中，MAR访问内存，将数据（实际上是地址）传给MDR，MDR又把地址传到MAR，MAR再次读取内存，MDR最终装载真正数据并传给DR。

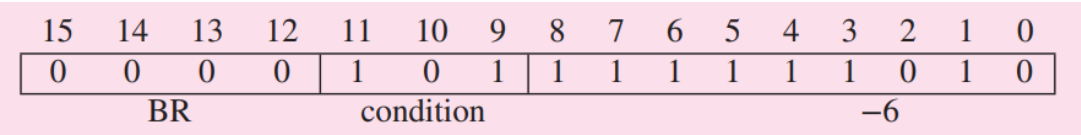
Base+offset Mode (LDR 0110/STR 0111)



LDR：R2中内容和offset相加，得到新地址，存到MAR，MAR访问内存将内容存到MDR，并最终存入R1 （即R2存放的是基地址）

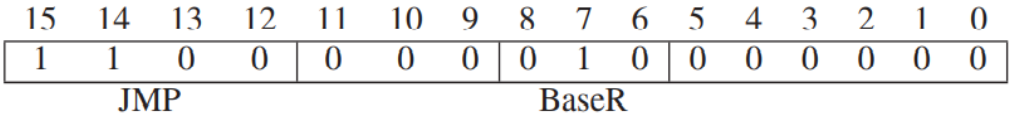
Control Instructions

Conditional Branches(BR)



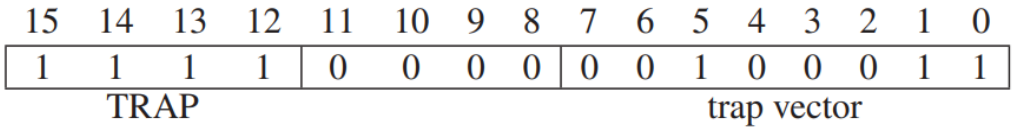
- **condition:n,z,p**（negative，zero, positive)
- 特殊：condition全为1，则无条件(unconditional)执行;全为0，PC+1，不改变时序
- opcode([5:12]), the condition to be tested([11:9]), the addressing mode bits([8:0])
- 此处condition代表EXCUTE结果是否不为0，假设初始PC存储地址为x36C9
- 若结果为0，不符合情况，指令不会对PC执行操作，PC+1，存储的地指令址变为x36CA
- 若结果不为0，符合情况，指令对PC执行操作，PC-6(with sign-extending)，存储的指令地址变为x36C3

JMP

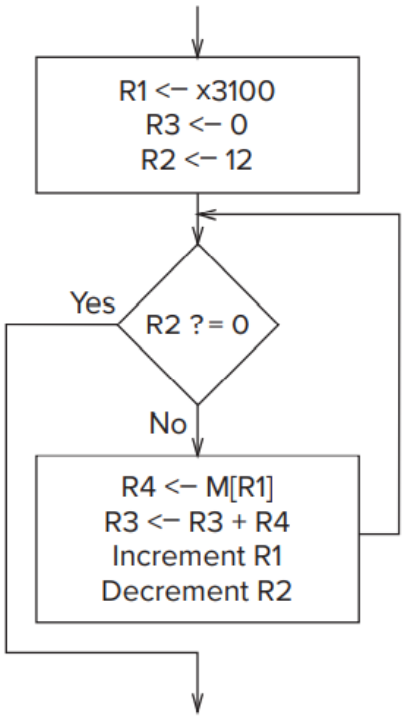
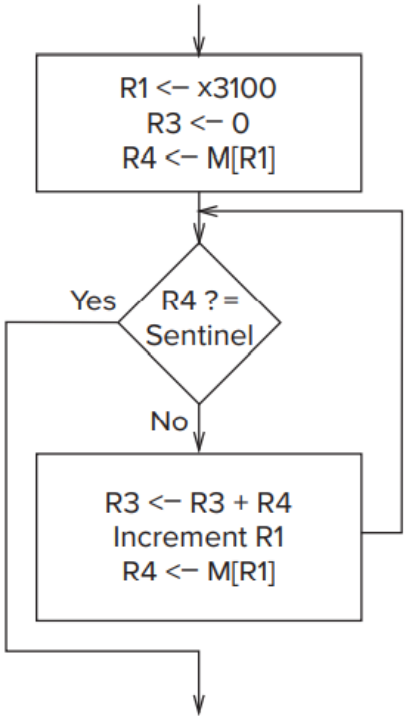


- BR指令有所限制，因为offset只能是在[-255,256]范围内，不能表示所有地址
- JMP将BaseR中的内容装入PC，执行结束后，会直接执行下一个指令，使得程序可以跳转到内存空间的任意位置，前提是BaseR的宽度是16bits (BaseR存储一个地址)

TRAP



- 改变PC内容使其指向操作系统所在空间内部

Loop Control with a Counter	Loop Control with a Sentinel（哨兵）
	
每执行一次循环，计数器减一直到为0	在要处理的数据末端放一个特殊字符，处理到它时停止

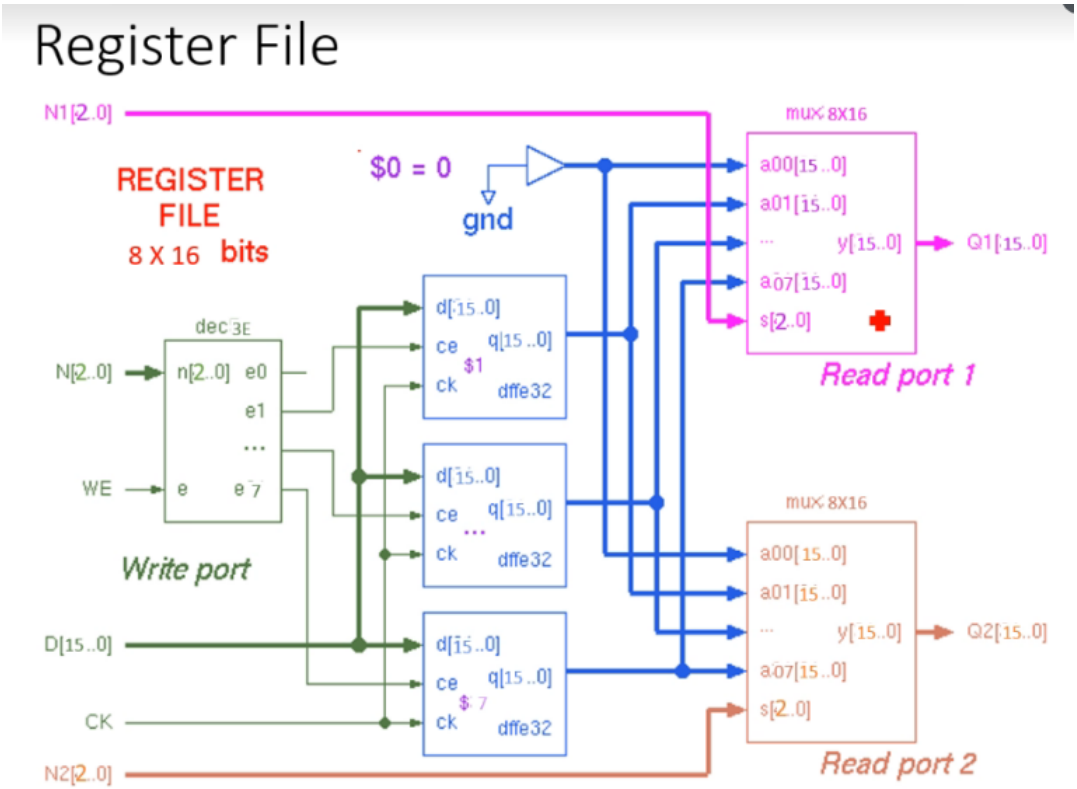
Data Movement Instructions

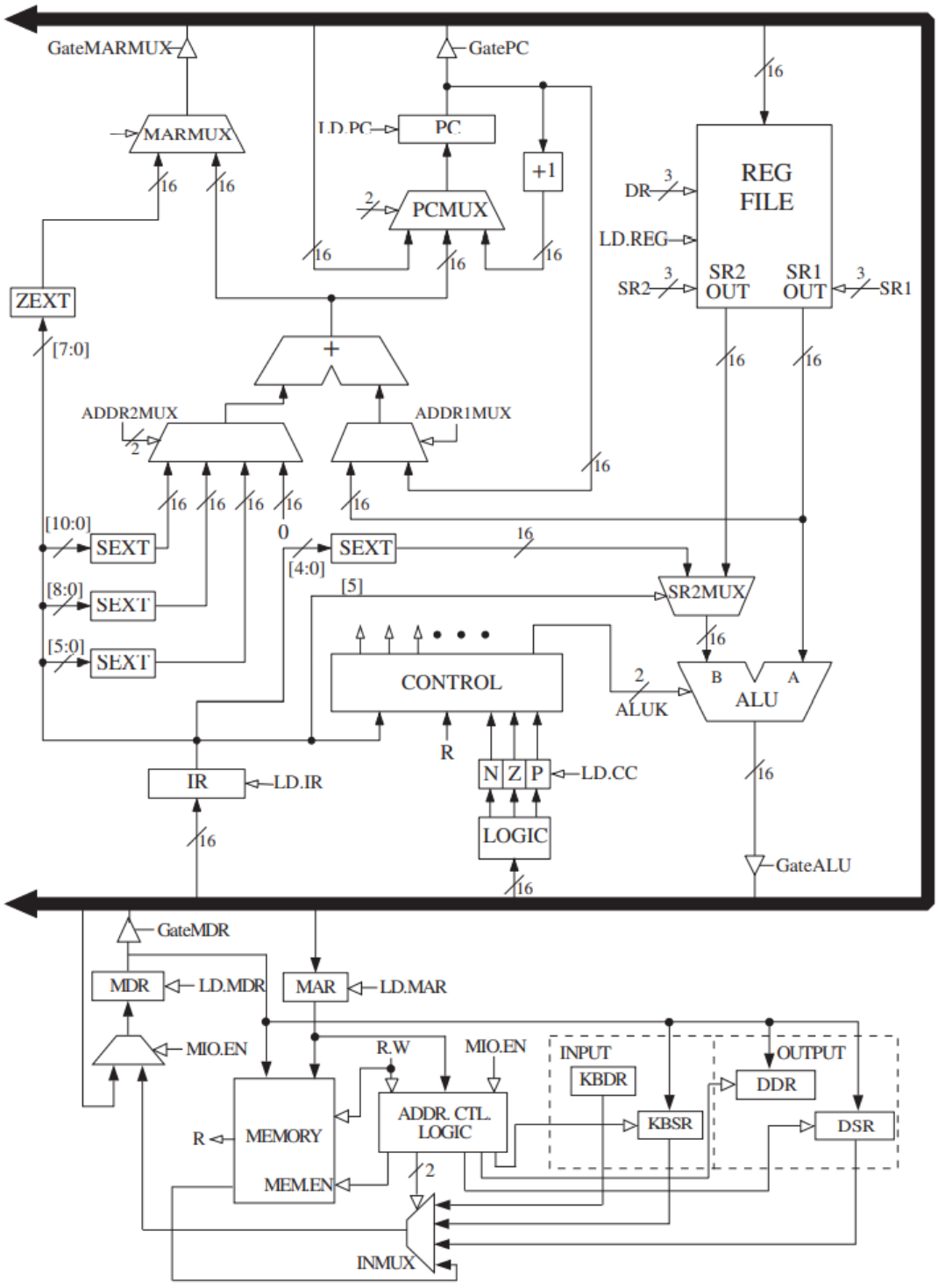
- **Overview**
  - **load**:The process of moving information from memory to a register
  - **store**: the process of moving information from a register to memory
  - **6 instructions: LD, LDR, LDI, ST, STR, STI**

Data Path(important!)

通过decoder分析信号，进行写入

用mux选择8个寄存器中的一个进行输出





- tri-state device(三态门)：防止所有部件都向bus传输信号导致混乱（信号也有此作用）
- sign-extension:
  - [0:4] AND,ADD (2 registers+signal)
  - [0:5] JMP, LDR, STR (2 register-DR+Base)
  - [0:8] BR, LEA, LD, LDI, ST, STI (1 register)
  - [0:10] JSR (0 register)

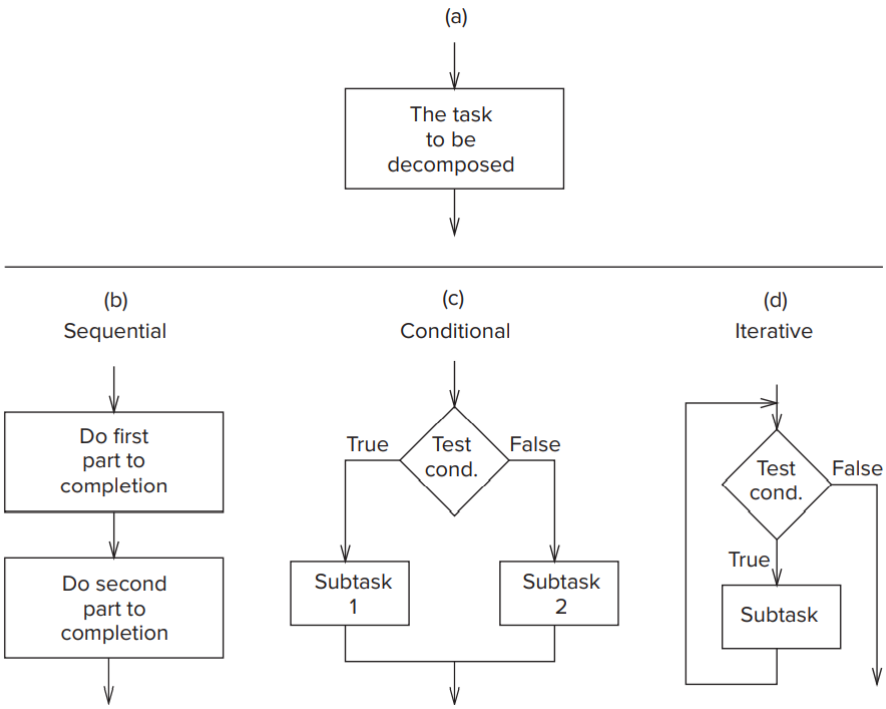
  important: the data path of different instructions

Programming

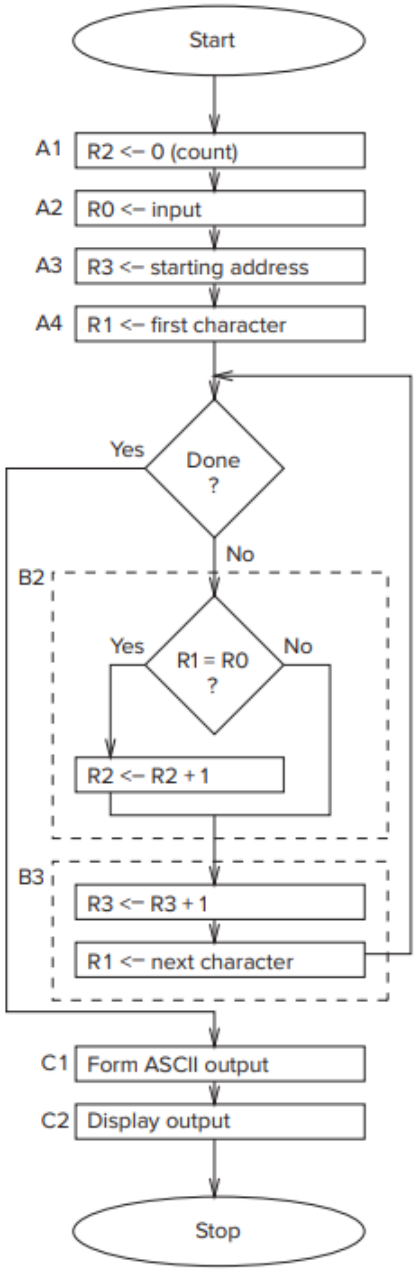
- **structured programming**

take a complex description of a problem and systematically decompose it into smaller and smaller manageable units

- **systematic decomposition**
- **stepwise refinement**
- **3 constructs to decompose**



> **A Sample**: count the number of occurrence of a specific character in a file



7. Assembly Language

- ISA dependent; usually each ISA has only one assembly language

Program

assembler: translation program

assembly: translation process

Instructions

Label Opcode Operands ; Comment

- **Opcodes and Operands**
  - mandatory(必要的)
  - #for decimal, x for hexadecimal, and b for binary.
- **Labels**
  - symbolic names that are used to identify memory locations that are referred to explicitly in the program
  - start with a letter of the alphabet
  - should not use **reserved words**
  - used in cases: explicitly referring to a memory location（显式访问地址时才需要）
    - The location is the target of a branch instruction
    - The location contains a value that is loaded or stored
- **Comments**
  - use semicolon ";"

Pseudo-Ops (Assembler Directives)

不产生操作，仅代表向汇编器传递消息用以指导汇编器工作；汇编器看到这些消息后就会将伪操作丢弃；伪操作不会占用地址

- **.ORIG**: tells the assembler where in memory to place the LC-3 program（确定程序初始位置）

  eg. .ORGI X3000
- **.FILL**:tells the assembler to set aside the next location in the program and initialize it with the value of the operand（下一个地址单元初始化为对应数值，注意单元序号要看无注释行、非伪操作行）

  eg. .FILL x0006
- **.BLKW**:tells the assembler to set aside some number of sequential memory locations (i.e., a BLocK of Words) in the program（占用一连串空间，适用于操作数值不确定的场合，可提前申请占用空间）

  eg .BLKW 1 (申请一个空间)
- **.STRINGZ**: tells the assembler to initialize a sequence of n+1 memory locations.（连续占用并初始化n+1个内存单元，前n个字的内容是字符串对应字符的ASCII码的零拓展，最后一个字初始化为0）

  eg. .STRINGZ "Hello, World!" （注意是一个单元存一个字符）
- **.END**: tells the assembler it has reached the end of the program and need not even look at anything after it.（不停止执行，只是标记程序的结束）

The Assembly Process

必须翻译为机器语言才能执行

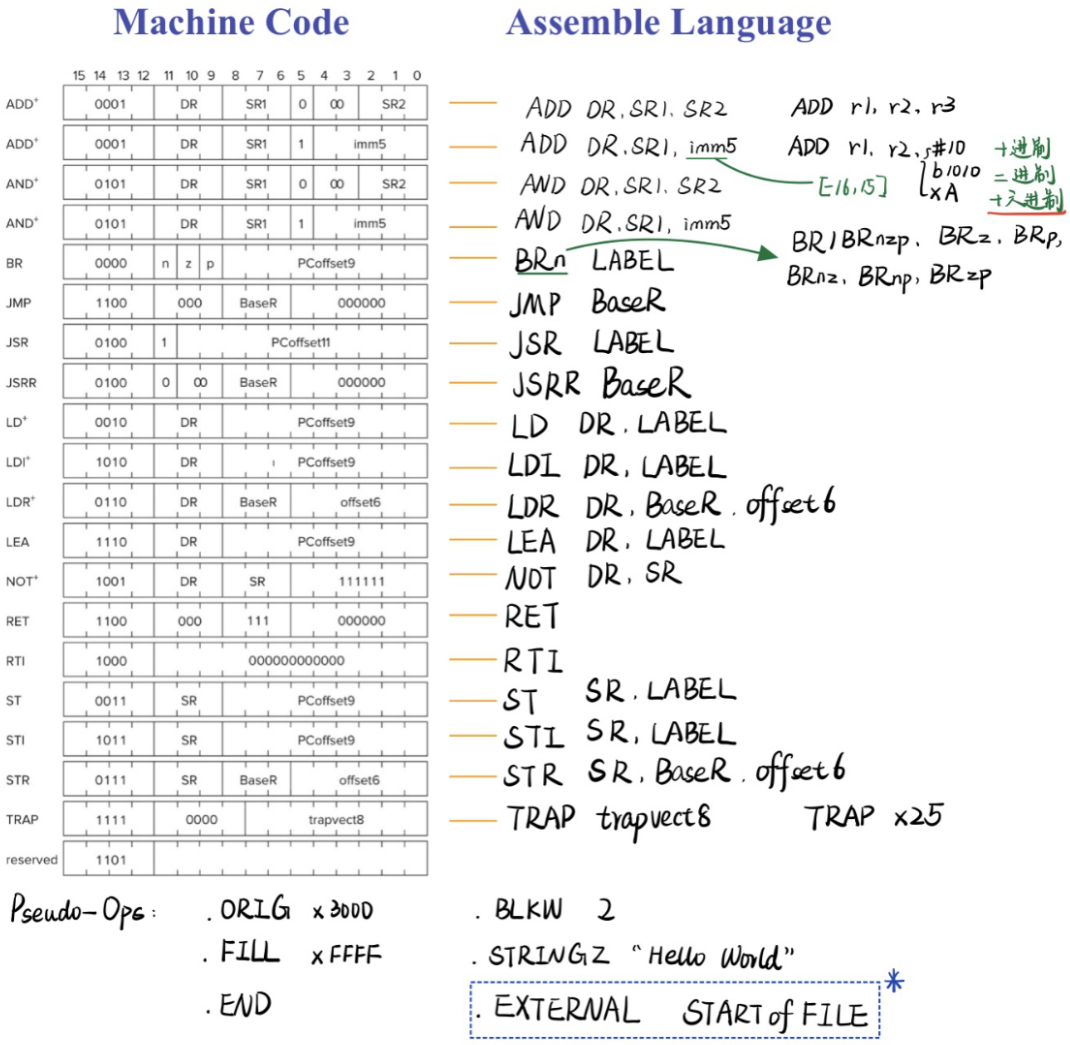
- **A Two-Pass Process**
  - *建立各符号和地址之间的映射关系——symbol table*
  - The First Pass: Creating the Symbol Table
    - 丢弃注释
    - 为每条指令和标识顺序分配地址
    - 记录标识、符号及其对应的地址（指令不会被计入二元表中）
    - 每次成功识别一次，LC++(location counter)
  - The Second Pass: Generating the Machine Language Program
    - 丢弃注释
    - 利用表生成机器指令
    - 注意在涉及偏移量的指令中，源操作数不能和当前PC移动到的位置的距离超过范围，否则不能通过汇编

Relative Info

- **executable image**
  - object files--link-->executable image
- **More than One Object File**
  - .EXTERNAL 声明某符号可能定义在其他模块中

    eg.　　.EXTERNAL ASCII

    带*指令会改变CC



- 汇编指令注意点
  - 在LD等指令中，与机器码不同，表示的不是offset而是具体的地址！会直接将地址存储的数值导入到相应的寄存器中

> 常用：
>
> LD R0 LABEL ;通常label指的是.FILL指令所在地址，存储着一个地址。此语句中R0通常为指针
>
> LDR R0 R1 #0 ;将R1中地址存储的值导入到R0中（注意R1要存储地址，通常load .FILL中内容
>
> 判断是否相同： num1取反加一再加num2
>
> 移向下一个地址：指针++（**注意区分地址和存储的数据**）

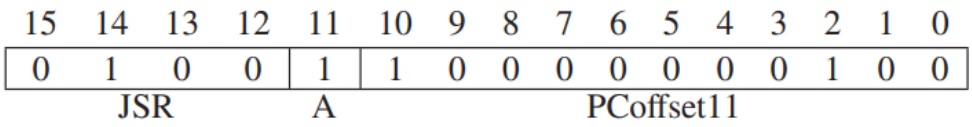
8. Data Structure

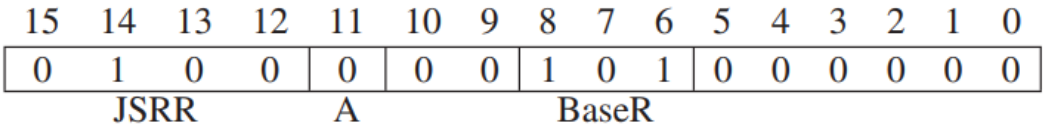
Subroutines

- call instruction(JSR(R)); return instruction(JMP)
- caller; callee(function)
- **JSR/JSRR**
  - R7<-PC+1
  - PC<-calculated address

> **Nested JSR/JSRR**

> 在使用内部SR之前，先把原来R7的值save起来，执行内部JSR后，R7被改变为内部函数JSR下一个地址，无法返回最开始的caller；此时将原来的值load入R7，使其能返回原来的地方。
>
> 对于简单函数，用临时方法保存即可；对于递归函数，则需要用栈保存





- **saving and restoring registers**
  - 在subroutine等操作中，容易将某些寄存器内的值改变，因此要提前存储好寄存器中的值
  - caller save:the caller save the needed value of some registers
  - callee save: the subroutine save the value
  - 伪指令实现：.BLKW； .FILL
- **Library routine**(库)

Stack

- LIFO: last in ,first out
- push, pop, top, stack pointer(point out th(JSRe top)
- in hardware: data entries move（top一直指向栈顶，value往下移动）

  in memory: data entries don't move（top初始即在底部，stack向上增长）

when we push or pop, the data stored on the stack does not physically move

push

```
PUSH ADD R6,R6,#-1
     STR R0,R6,#0
```

pop

```
POP LDR R0,R6,#0
    ADD R6,R6,#1
```

underflow

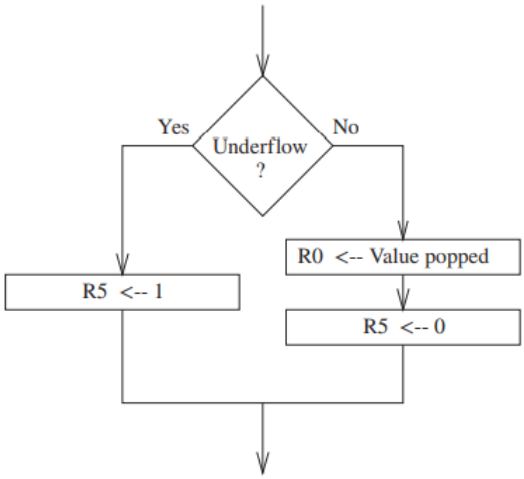
Attempting to pop items that have not been previously pushed（超出范围，如两个元素pop3个）

overflow

push过多元素

```
POP   LD R1,EMPTY
```

```
        ADD R2,R6,R1        ; Compare stack 检测是否到栈底
        BRz UNDERFLOW       ; pointer with x4000.
;
        LDR R0,R6,#0
        ADD R6,R6,#1
;
        RET
EMPTY .FILL xC000           ; EMPTY <-- negative of x4000
```



Full Picture of POP and PUSH

- R5判断操作是否成功
- 存储R1, R2初始内容
- R1导入指针（R6）负值，两者相加判断是否到栈底/栈顶，若到，则重新存储R1,R2值，操作失败，R5存储1，return
- 若未到，进行真正的PUSH/POP操作，注意指针（R6）移动（不要弄错方向！），并重新存储R1,R2值，return

```
01   ;
02   ; Subroutines for carrying out the PUSH and POP functions.  This
03   ; program works with a stack consisting of memory locations x3FFF
04   ; through x3FFB.  R6 is the stack pointer.
05   ;
06   POP             AND     R5,R5,#0        ; R5 <-- success
07                   ST      R1,Save1        ; Save registers that
08                   ST      R2,Save2        ; are needed by POP
09                   LD      R1,EMPTY        ; EMPTY contains -x4000
0B                   ADD     R2,R6,R1        ; Compare stack pointer to x4000
0C                   BRz     fail_exit       ; Branch if stack is empty
0D   ;
0E                   LDR     R0,R6,#0        ; The actual "pop"
0F                   ADD     R6,R6,#1        ; Adjust stack pointer
10                   BRnzp   success_exit
11   ;
12   PUSH            AND     R5,R5,#0
13                   ST      R1,Save1        ; Save registers that
14                   ST      R2,Save2        ; are needed by PUSH
15                   LD      R1,FULL         ; FULL contains -x3FFB
16                   ADD     R2,R6,R1        ; Compare stack pointer to x3FFB
17                   BRz     fail_exit       ; Branch if stack is full
18   ;
19                   ADD     R6,R6,#-1       ; Adjust stack pointer
1A                   STR     R0,R6,#0        ; The actual "push"
1B   success_exit    LD      R2,Save2        ; Restore original
1C                   LD      R1,Save1        ; register values
1D                   RET
1E   ;
1F   fail_exit       LD      R2,Save2        ; Restore original
20                   LD      R1,Save1        ; register values
21                   ADD     R5,R5,#1        ; R5 <-- failure
22                   RET
23   ;
24   EMPTY           .FILL   xC000           ; EMPTY contains -x4000
25   FULL            .FILL   xC005           ; FULL contains  -x3FFB
26   Save1           .FILL   x0000
27   Save2           .FILL   x0000
```

save R_1, R_2 内容

stack empty?

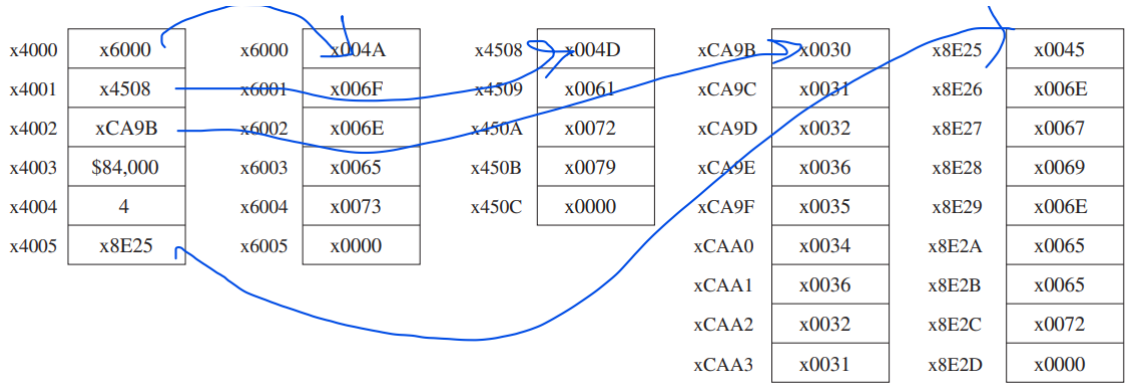
把原来的值存回 R_1, R_2

→ R_5 标志成功与否

Queue

- FIFO
- Front(指向第一个元素*之前*的位置), Rear(指向最后一个元素)
- priority queue（优先队列）: 元素被赋予优先级（但内部不一定按偏序关系排列）。当访问元素时，具有最高优先级的元素最先删除。优先队列具有最高级先出 （first in, largest out）的行为特征。通常采用堆数据结构来实现。

Character Strings



- 类似于结构体，每个位置存储一个指针，每个指针指向一串字符串。
- ASCII code: 1 byte(space： x20)
- every string must end with x0000

> representation of ordered list(有序表)
>
> sequential(array)
>
> linked list

Linked List

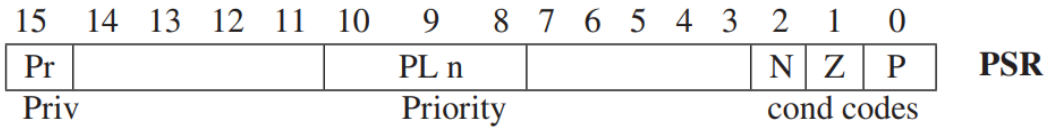
- pointer+content; memory is unnecessarily consecutive
- head,head pointer, tail,node

Array

9. I/O

previlege, priority，Memory address space

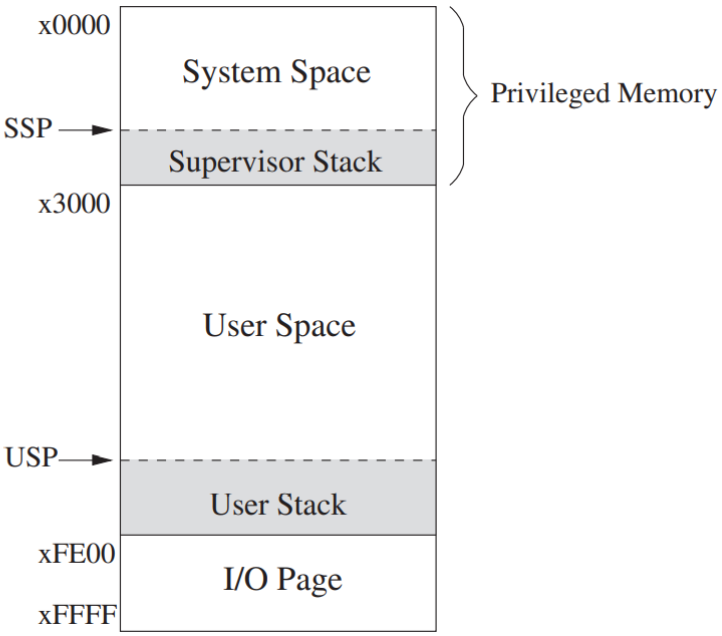
- previlege and priority: two orthogonal notions(have nothing to do with each other)
- **PSR**(processor status register)
  - priv: 1-unprivileged; 0-supervisor privilege
  - PL: 0-7
  - 注意是大写NZP



- **Organization of Memory**
  - I/O Page: 不是真正的内存.identify registers that take part in input and output functions and some special registers associated with the processors (PSR,MSR(processor's Master

Control Register))

- **SSP**: supervisor Sstack pointer; **USP**: user stack pointer



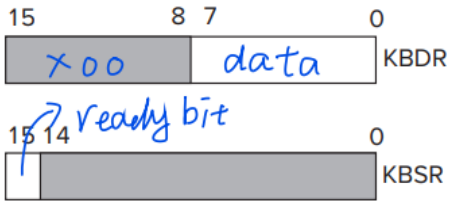
Input/Output

- **Memory-Mapped I/O vs. Special I/O Instructions**
  - Memory-Mapped I/O: 使用相同的data movement instructions，因此需要标记辨认，此处将内存地址和寄存器对应起来（mapped）
  - Special I/O Instructions: 另开一些指令执行I/O操作
- **Asynchronous vs. Synchronous**

  处理器的频率（速度）远大于人类输入输出速度，输入输出速度也不可能一直保持一致。如果同步，人类难以跟上。因此采用*异步*机制。
  - require：protocol or handshaking mechanism （in this case, one-bit status register. called *flag*)
  - a single flag: **ready bit(只要轮询查到为1，则开始写入/读取工作，输入输出设备暂时失效)**
- **Interrupt-Driven（中断驱动） vs. Polling（轮询）**
  - 交互的两种基本方式
  - 区别在于谁控制交互： I/O设备；处理器

Polling（轮询）

- **Input from the Keyboard**
  - **KBDR**(keyboard data register) xFE02
  - **KBSR**(keyboard status register) xFE00



  - **Basic Input Service Routine**

    > (when polling) The program repeatedly test KBSR[15] until the bit is set

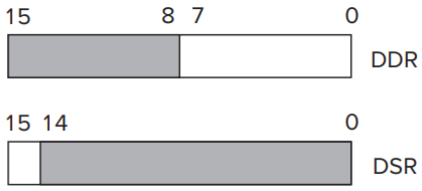
> a key is struck
>
> ASCII->KBDR[7:0], 1->KBSR[15] (cannot type; processor can load the ASCII(read))
>
> the code is read
>
> 0->KBSR[15]
>
> another key can be struck now

> **The electronic circuits will automatically set the ready bit**

```
**NOTE: 必须用LDI，LD范围太小**
```

```
01 START LDI R1, A ; Test for
02 BRzp  START ; character input        (如果ready bit不是1（负数），一直轮询)
03 LDI   R0, B
04 BRnzp NEXT_TASK ; Go to the next task
05 A .FILL xFE00 ; Address of KBSR
06 B .FILL xFE02 ; Address of KBDR
```

- **Output to the Monitor**
  - **DDR**(Display Data Register) xFE06
  - **DSR**(Display Status Register) xFE04




```
01 START LDI R1, A ; Test to see if
02 BRzp  START ; output register is ready
03 STI   R0, B
04 BRnzp NEXT_TASK
05 A .FILL xFE04 ; Address of DSR
06 B .FILL xFE06 ; Address of DDR
```

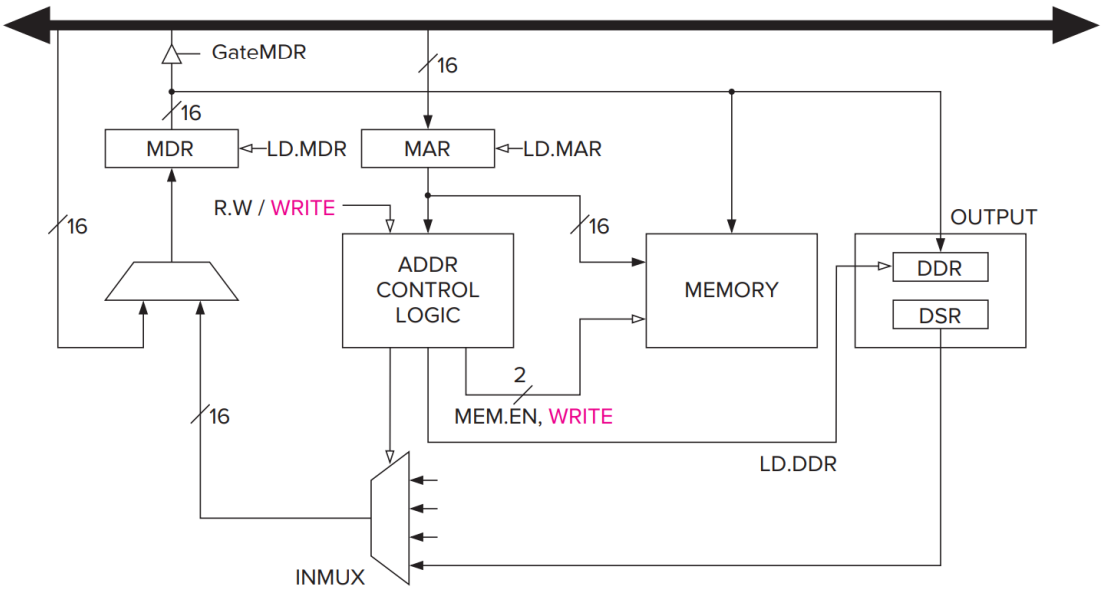


Figure 9.6 Memory-mapped output.

- **the complete picture**

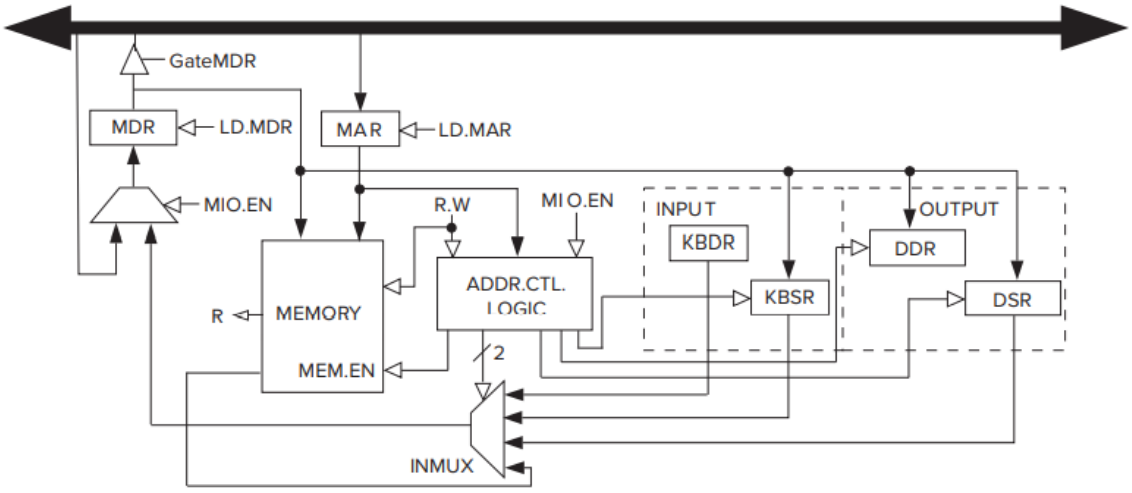


Figure 9.8 Relevant data path implementation of memory-mapped I/O.

MIO.EN: indicates whether a data movement from/to memory or I/O is to take place this clock cycle.

MDR->KBSR,DDR,DSR

DSR,KBSR,KBDR->MDR

Operating System Service Routines (LC-3 Trap Routines)

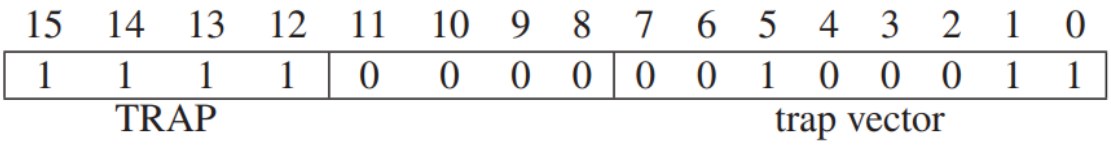
I/O activities share a lot of devices with different programs, if user can get into the details and mess up, the system may be a mess.

system call/service call: the request made by the user program(eg. TRAP)

*Operating System(OS) , I need help（**OSH**:operating system help)*

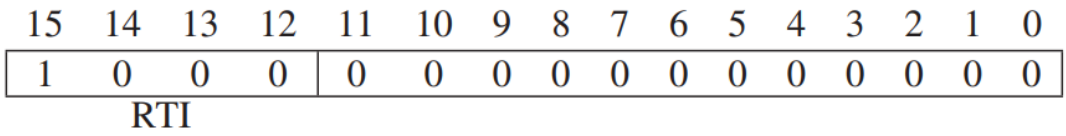
因此，我们常在用户的program里使用TRAP指令进行权限的交换，用户不需要知道实现的具体细节

- **The Trap Mechanism**
  - A set of service routines：由操作系统提供，但以用户身份执行，是操作系统的组成部分，起始于各自固定的内存地址
  - A table of the starting addresses（System Control Block/Trap Vector Table, in the supervisor memory):各服务程序的起始地址,如trap vector是x21，则在该地址中存储service routine-HALT程序的起始地址
  - The TRAP instruction
  - A linkage： 返回到用户程序
- **The TRAP Instruction**
  - changing the PC to the starting address of the relevant service routine on the basis of its trap vector
  - providing a way to get back to the program that executed the TRAP instruction. The “way back” is referred to as a linkage



> **phases**
>
> 1. user mode: R6(point to user stack)->Saved_USP
>
> Saved_SSP->R6
>
> PSR,PC->system stack;
>
> 2. 0->PSR[15] require supervisor privilege; PSR[10:8] is unchanged
> 3. trap vector->zero-extended, form an address that corresponds to a location in the Trap Vector Table; PC then is loaded the address

- **The RTI Instruction: To Return Control to the Calling Program**



  - pops the top two values on the system stack into the PC and PSR
  - PSR[15] must is examined to see whether the processor was running in **User mode** or **Supervisor mode**
    - User mode: stack pointer(R6) is adjusted. R6->Saved_SSP, Saved_USP->R6

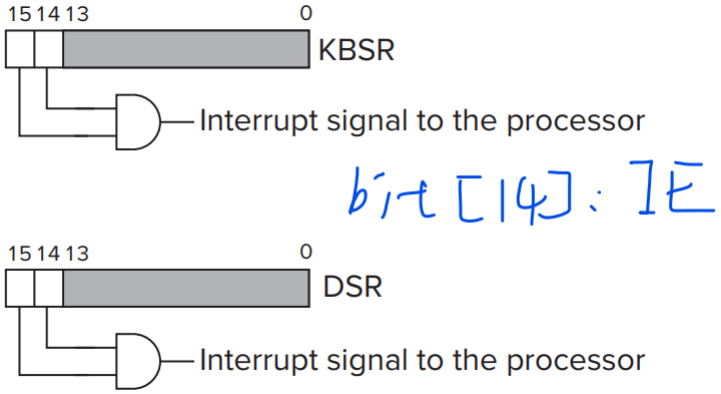
Interrupts and Interrupt-Driven I/O（中断驱动）

在polling中，CPU使用大量时间去test ready bit，Interrupt-Driven I/O可以使处理器花更多时间去处理有意义的事情（提升CPU利用率）

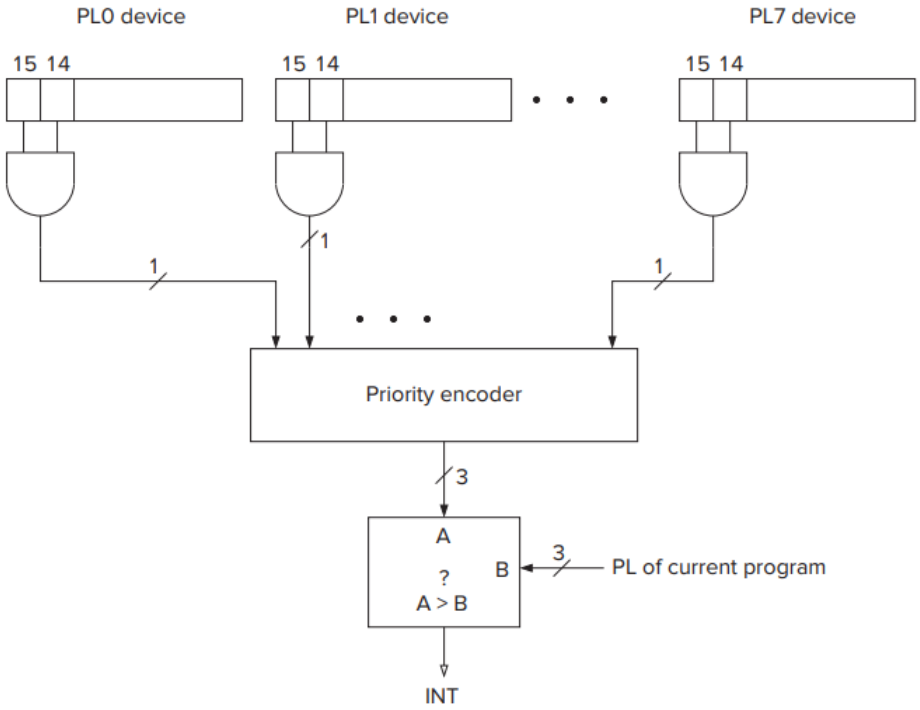
在一个指令结束，另一个指令还未开始（FETCH前）执行

- **2 parts**
  - the mechanism that enables an I/O device to interrupt the processor
  - the mechanism that handles the interrupt request
- **Part 1 Causing the Interrupt to Occur**
  - The Device Must Want Service: ready bit

- The Device Must Have the Right to Request That Service: **interrupt enable(IE)** bit
- The Urgency of the Request : 想要执行的程序priority level(PL)要更高



- **The INT Signal**: 表示停止现在执行的程序(interupt signal)
  - The Test for INT:



- **Part 2: Handling the Interrupt Request**
  - Initiate the interrupt

    1.Save the State of the Interrupted Program(PC,PSR,R6)

    2.Load the State of the Interrupt Service Routine

    INTV: interrupt vector (像trap一样，对应一个**Interrupt Vector Table** x0100-x01FF，trap则是x0000 to x00FF,)

    (要求interrupt的程序会将INTV、request signal、priority level一起送到prosessor处，一旦被接受，INTV就会扩展为16bits，PC和PSR做出相应改变)
  - Service the interrupt
  - Return from the interrupt : RTI

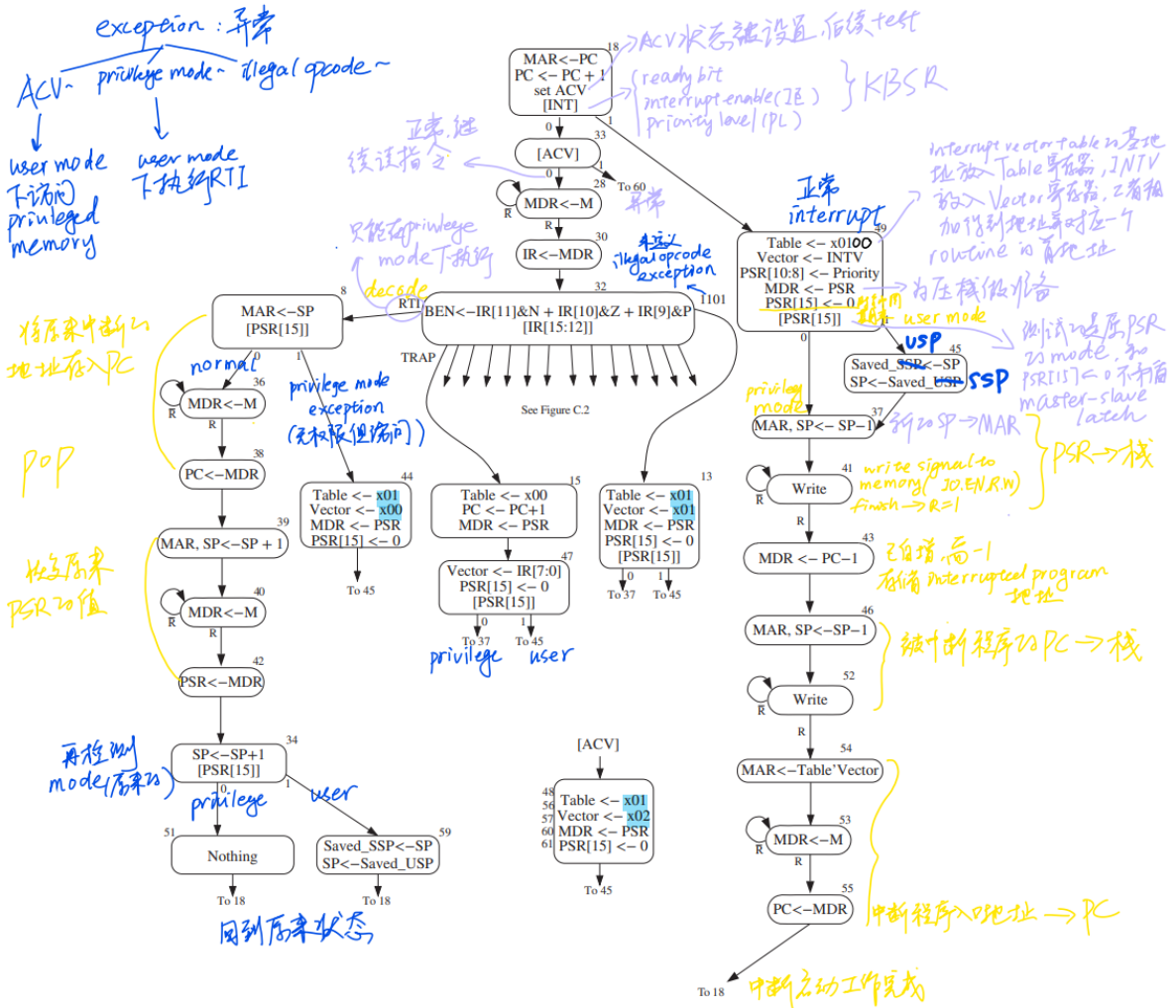


Figure C.7 LC-3 state machine showing interrupt control.

10. Calculator

- **Stack**

  区分prefix，infix等，此处和树的操作不同。

  实际上出栈并未直接remove entries，物理上依然保留着原值，但SP更改，意味着之前的栈已经被废弃，内容可被更改

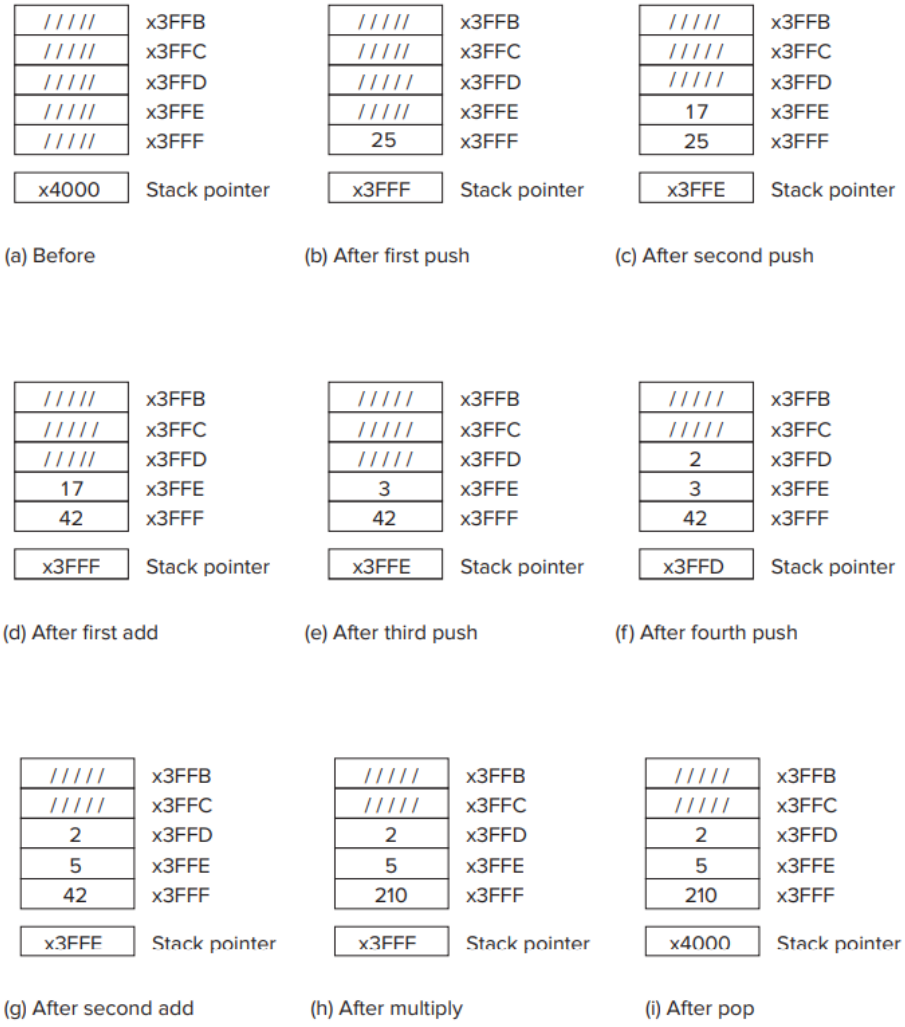
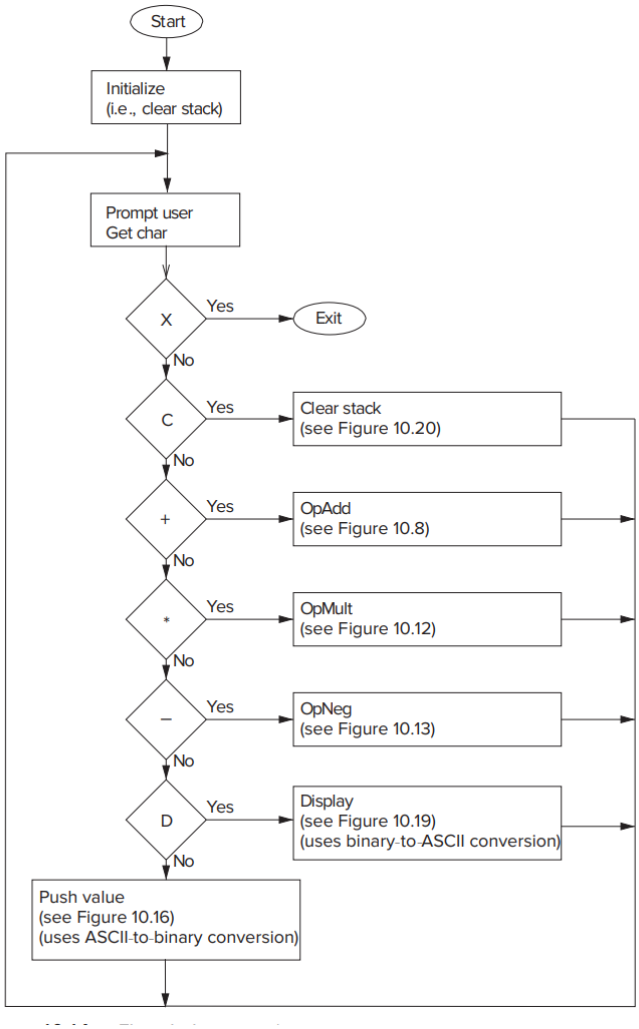
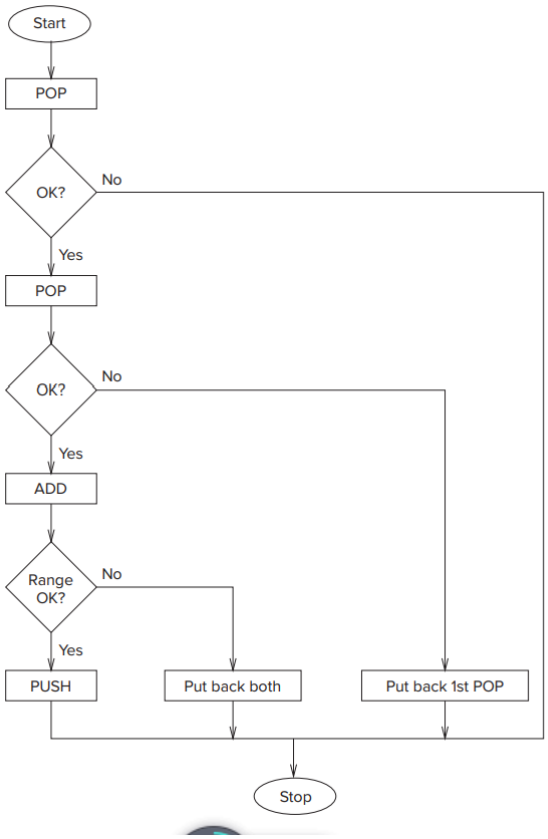


Figure 10.6 **Stack usage during the computation of (25 + 17) · (3 + 2).**

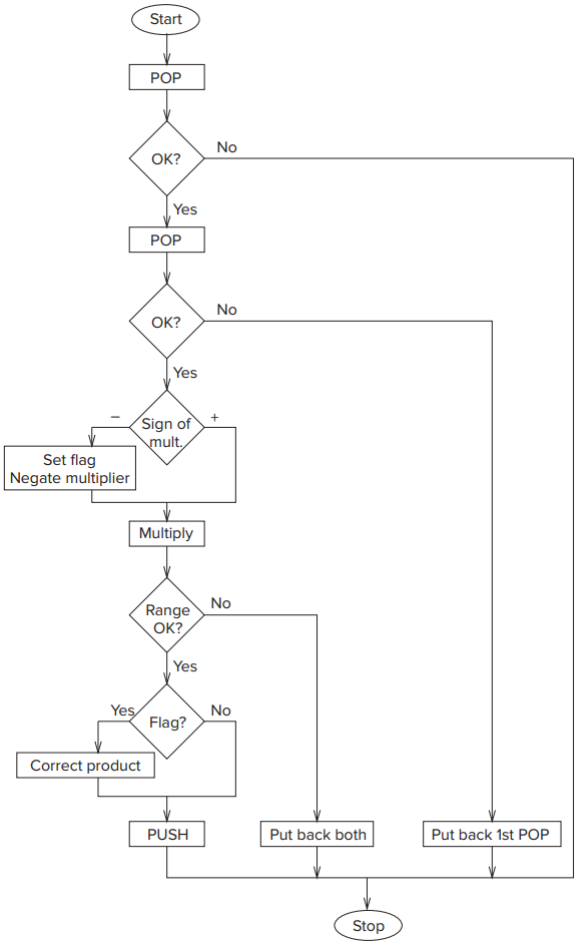
- **OpAdd,OpMult,OpNeg**



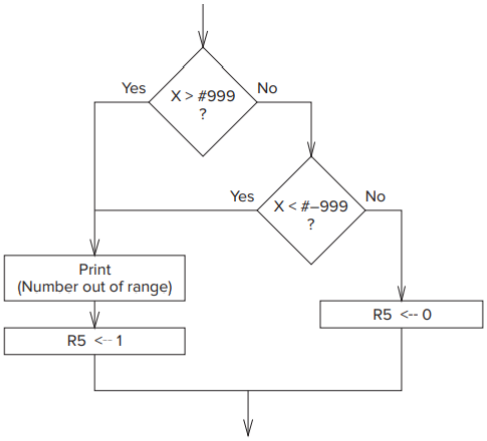
- OpAdd, which will pop two values from the stack, add them, and push the result onto the stack.



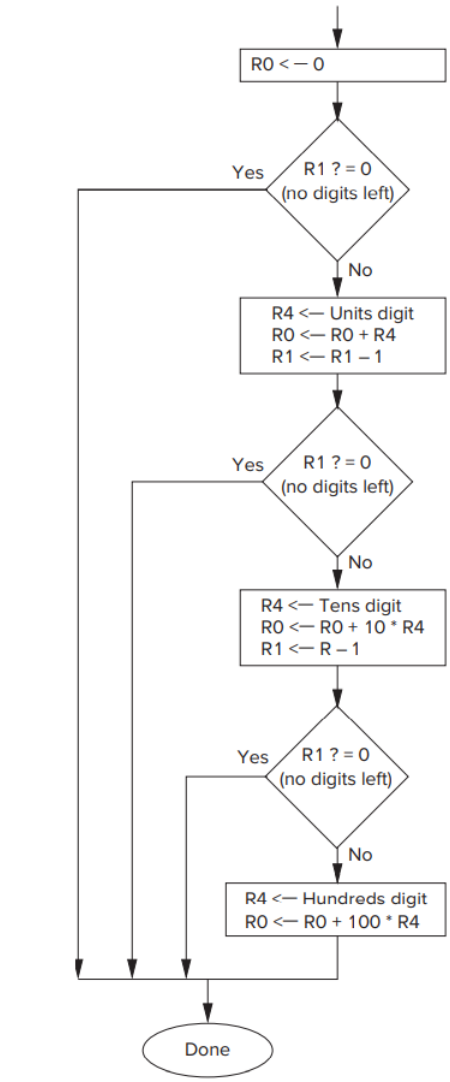
- OpMult, which will pop two values from the stack, multiply them, and push the result onto the stack.



- OpNeg, which will pop the top value, form its 2's complement negative value, and push the result onto the stack. This will allow us to subtract two numbers A minus B by first forming −B and then adding the result to A.
- **Range check**



- **ASCII->binary**(example：3 digits)



Flowchart, subroutine for ASCII-to-binary conversion.

- **Binary->ASCII**
  - 不停减100直到为负数，则知道百位的数字（减的次数-1），再加回100，得到只剩十位、个位的数字，以此类推
  - 转化成ASCII码（注意符号的处理）

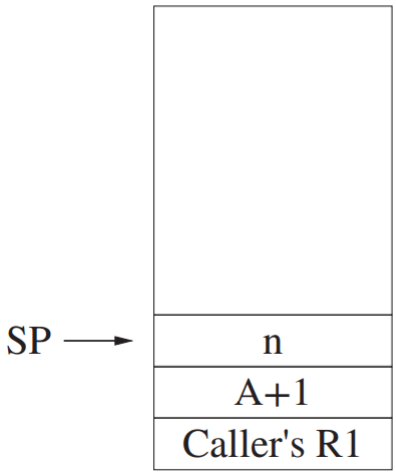
11.Re:Recursion

push: 返回地址，每一步的结果

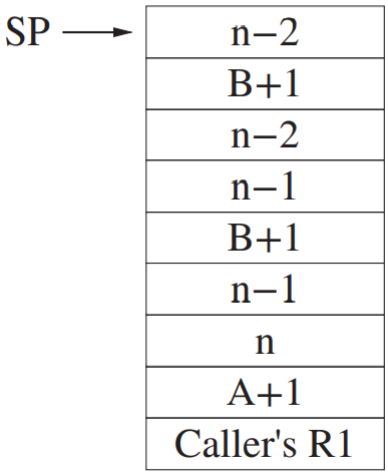
需要test->结束->pop

- **factorial**

a. Contents of stack when JSR FACT executes in #1



a. Contents of stack when JSR FACT executes in #3




```
FACT        ADD   R6,R6,#-1
            STR   R1,R6,#0    ; Push Caller's R1 on the stack, so we can use R1.
;
            ADD   R1,R0,#-1   ; If n=1, we are done since 1! = 1
            BRz   NO_RECURSE
;
            ADD   R6,R6,#-1
            STR   R7,R6,#0    ; Push return linkage onto stack
            ADD   R6,R6,#-1
            STR   R0,R6,#0    ; Push n on the stack
;
            ADD   R0,R0,#-1    ; Form n-1, argument of JSR
B           JSR   FACT
            LDR   R1,R6,#0    ; Pop n from the stack
            ADD   R6,R6,#1
            MUL   R0,R0,R1    ; form n*(n-1)!
;
            LDR   R7,R6,#0    ; Pop return linkage into R7
            ADD   R6,R6,#1
NO_RECURSE  LDR   R1,R6,#0    ; Pop caller's R1 back into R1
            ADD   R6,R6,#1
            RET
```

- comprehension

searching tree->

back tracking=pop

search&test=push

- caution
  - 使用某寄存器作为结果存储位置（返回值、判断值），不需要将其压入栈中，否则最后得到的结果并非最终结果
  - 每次进行递归，都需要将存储位置的寄存器内容进行更改（压入栈中的寄存器），如此，返回时寄存器存的是之前结点的内容