# Machine Prog &Processor Arch ——Advanced ——ISA&Logic

侯旭森 许珈铭

2023.10.18

# Advanced (CS:APP Ch. 3.10-3.11)

许珈铭

# 变长数组

- 遍历定长数组时 gcc 能利用数组长度做优化

```
/* Compute i,k of fixed matrix product */
int fix_prod_ele (fix_matrix A, fix_matrix B, long i, long k) {
    long j;
    int result = 0;

    for (j = 0; j < N; j++)
        result += A[i][j] * B[j][k];

    return result;
}
```

```
1   /* Compute i,k of fixed matrix product */
2   int fix_prod_ele_opt(fix_matrix A, fix_matrix B, long i, long k) {
3       int *Aptr = &A[i][0];    /* Points to elements in row i of A    */
4       int *Bptr = &B[0][k];    /* Points to elements in column k of B */
5       int *Bend = &B[N][k];    /* Marks stopping point for Bptr       */
6       int result = 0;
7       do {                                /* No need for initial test */
8           result += *Aptr * *Bptr;        /* Add next product to sum  */
9           Aptr ++;                        /* Move Aptr to next column */
10          Bptr += N;                      /* Move Bptr to next row    */
11      } while (Bptr != Bend);             /* Test for stopping point  */
12      return result;
13  }
```

# 变长数组

- 遍历定长数组时 gcc 能利用数组长度做优化
- 变长数组无法做上述优化，需要用 imul 等指令

```
int var_ele(long n, int A[n][n], long i, long j) {
    return A[i][j];
}
```

```
int var_ele(long n, int A[n][n], long i, long j)
n in %rdi, A in %rsi, i in %rdx, j in %rcx
var_ele:
  imulq   %rdx, %rdi              Compute n · i
  leaq    (%rsi,%rdi,4), %rax     Compute x_A + 4(n · i
  movl    (%rax,%rcx,4), %eax     Read from M[x_A + 4(n · i) + 4j]
  ret
```

# 变长栈帧

- 帧指针 %rbp
  - **enterq = (pushq %rbp) + (movq %rsp, %rbp)**
  - **leaveq = (movq %rbp, %rsp) + (popq %rbp)**

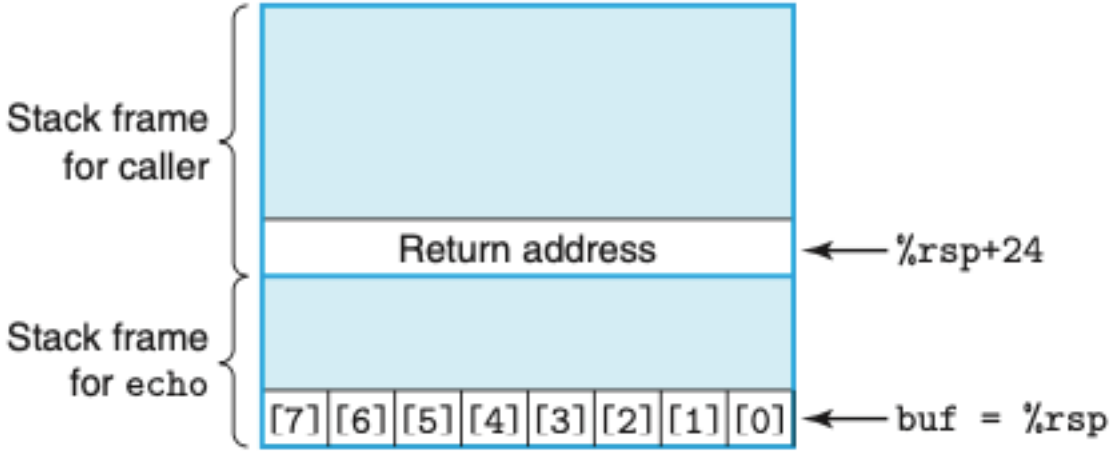
# 栈分析总结

- **call func/ret**
- **pushq/popq %reg**
- **movq %reg, -8(%rbp)**
- **enterq, leaveq**

- 栈帧分析 %rsp, %rbp
  - **movq %rbp, %rsp/movq %rsp, %rbp**
  - **Subq $16, %rsp**

# 缓冲区溢出攻击

- 最基本的形式：覆盖返回地址




| Characters typed | Additional corrupted state |
|---|---|
| 0–7 | None |
| 9–23 | Unused stack space |
| 24–31 | Return address |
| 32+ | Saved state in caller |

# 防御

- 栈随机化 / ASLR（地址空间随机化）/ 基址随机化（PIE）
  - ASLR(2001 in Linux) 加载可执行文件时提供随机化能力 stack heap libraries
  - PIE(2005 in Linux) 编译时支持可执行文件地址随机化 executable
- Canary
  - https://en.wiktionary.org/wiki/canary_in_a_coal_mine
  - %fs:0x28
- NX（No-eXecute）

# 对抗 NX: ROP

- NX 只标记了栈和堆上的代码不可执行（即不能跳转到栈上），但我们可以不执行栈和堆上的代码
- 程序本身可能含有危险代码段，只需跳转到这些位置
  - 开 PIE 时很困难
- 利用 pop 等指令可以构造恶意的寄存器值

# 对抗 canary

- 程序中有时有输出栈上字符串的指令
- 如果可以把输出字符串末尾的 ‘\0’ 改掉就可以输出任意长度内容，从而得到 canary 值
- canary 开头是 0x00，但如果能在函数返回前把这个 0x00 改掉，就可以输出 canary

# 对抗 canary

- 有些情况下程序崩溃后会自动重启，重启后 ASLR 的偏移，以及 canary 的值每次都是相同的
- 当我们能修改栈上任意字节时，我们可以先修改 canary 的某一位
- 当且仅当我们修改的值和 canary 原始值一样时，程序才不会崩溃 ⇒ 可以在 256 次尝试内得到 canary 值某一字节的值
- 逐字节暴力尝试可得到 canary 值

# 对抗 ASLR

- nop sled（课本 p322 en ver.）

# 对抗 ASLR

- 泄漏基址
- 找到某个函数的内存地址，之后计算这个函数和特定函数的相对偏移，进而得到特定函数的内存地址
- 可以用输出 canary 类似的方法，不过这里是试图让程序输出原来的返回地址

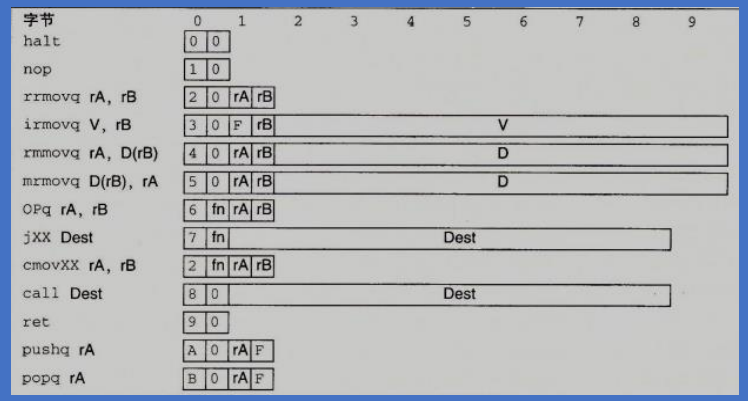
连一刻都没有为第三章的结束哀悼，立刻赶到战场的是——第四章 处理器！

# ISA&Hardware (CS:APP Ch. 4.1-4.2)

侯旭森

# ISA (Y86-64指令集体系结构)


程序员可见的状态




指令集和它们的编码

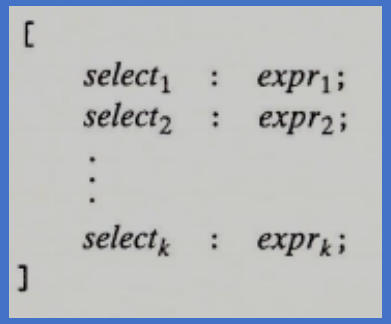
| 值 | 名字 | 含义 |
|---|---|---|
| 1 | AOK | 正常操作 |
| 2 | HLT | 遇到器执行 halt 指令 |
| 3 | ADR | 遇到非法地址 |
| 4 | INS | 遇到非法指令 |

异常处理

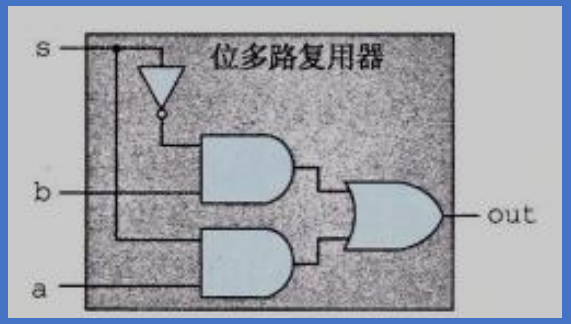
编程示例与小细节

CISC & RISC （很重要！）

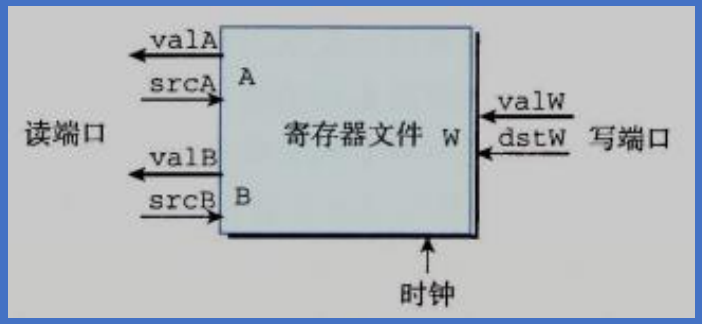
# Logic (逻辑设计和硬件控制语言HCL)




硬件控制语言HCL



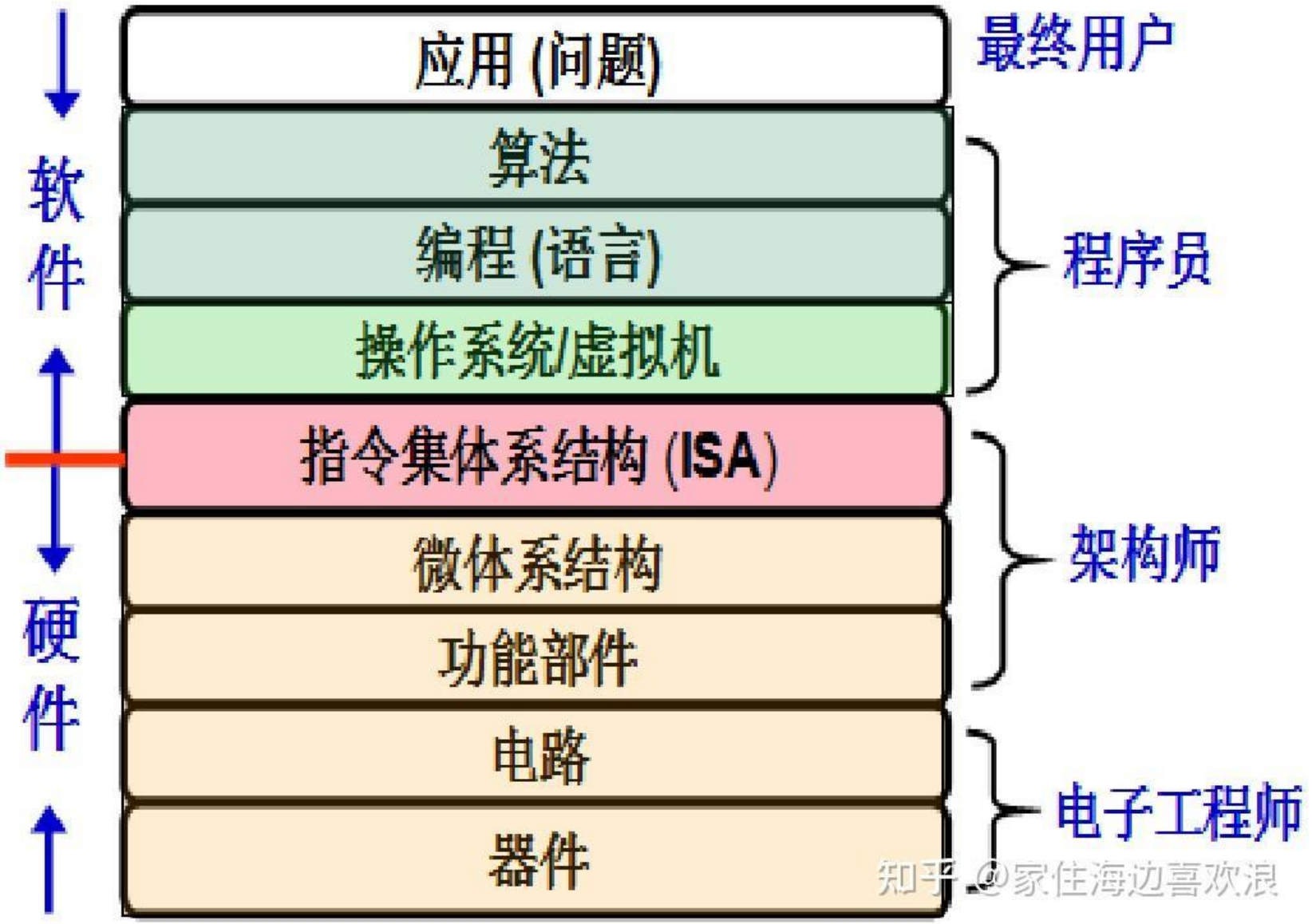

逻辑门与组合电路



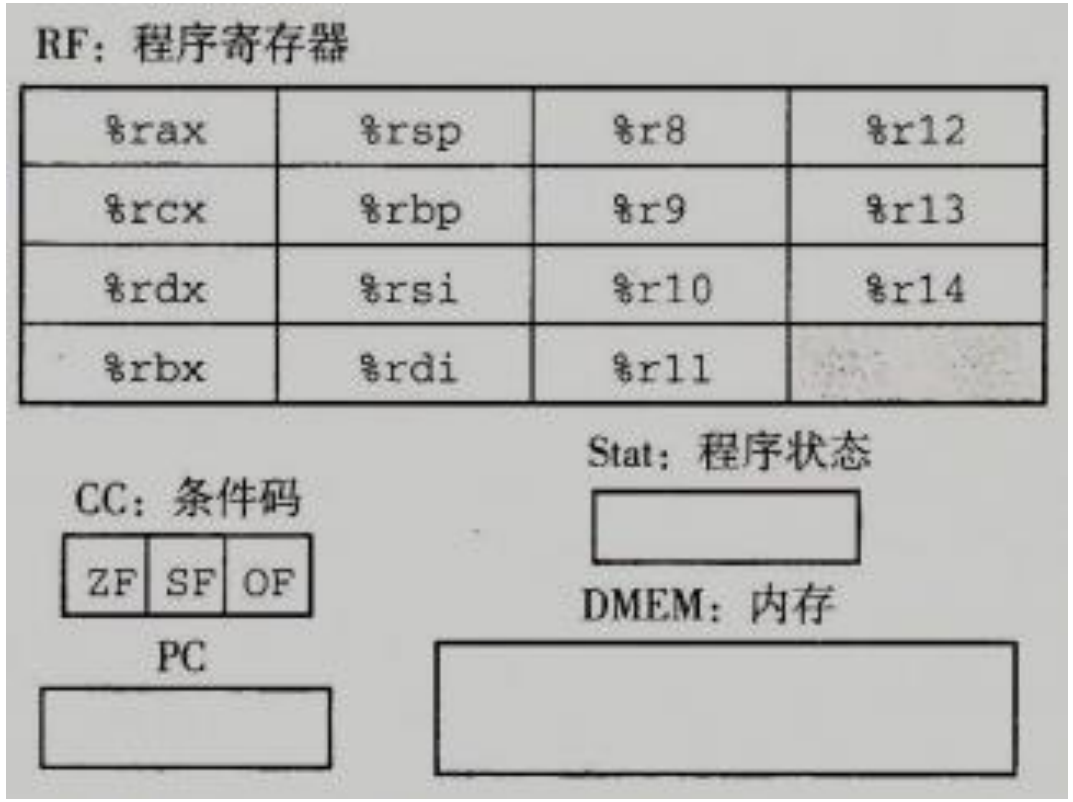

存储器和时钟

# ISA

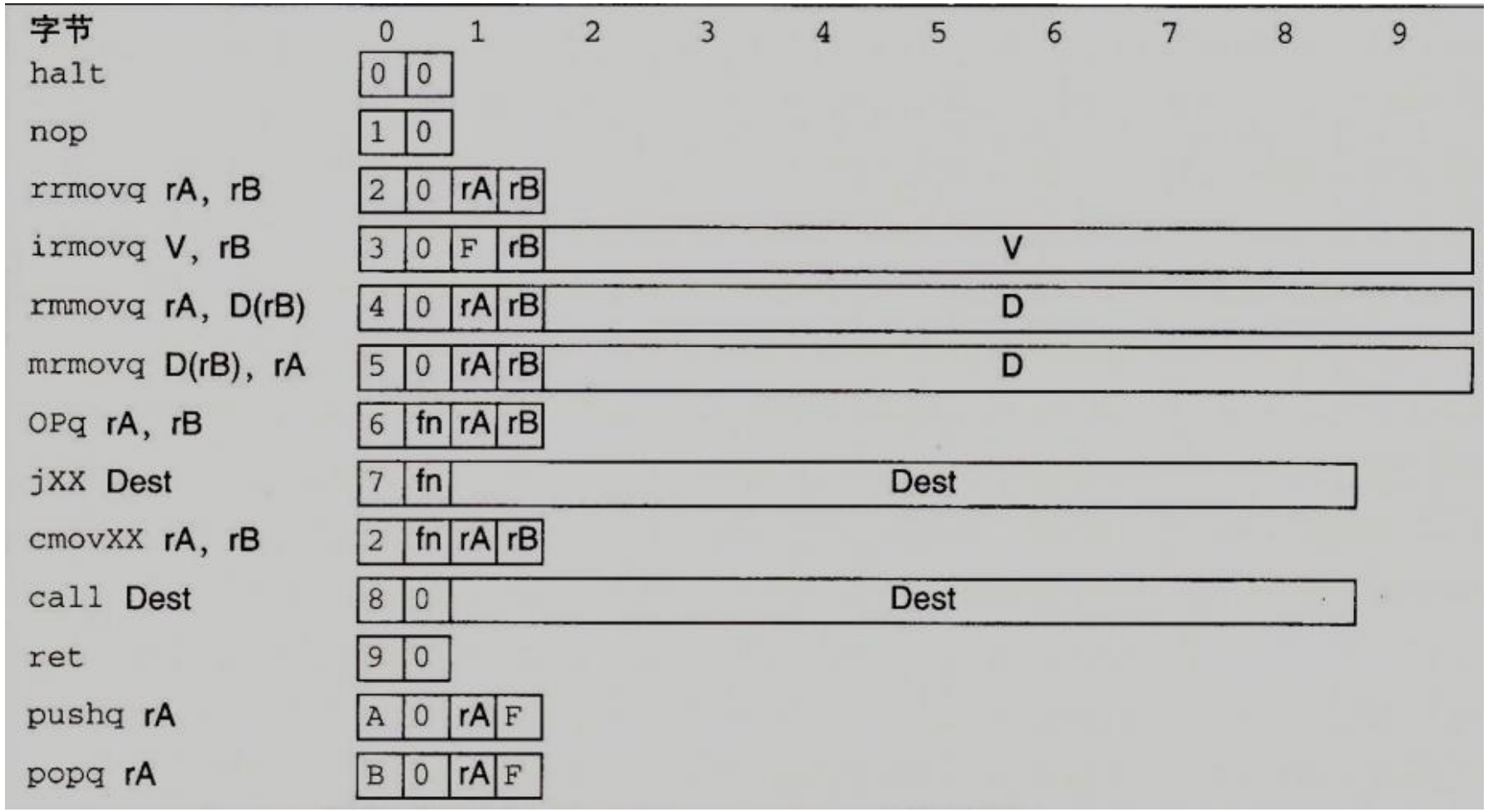
- Instruction-Set Architecture, ISA
- ISA在编译器编写者和处理器设计人员之间提供了一个抽象概念层。
- 基本上一套ISA对应一种编译语言。
- Y86-64指令集。

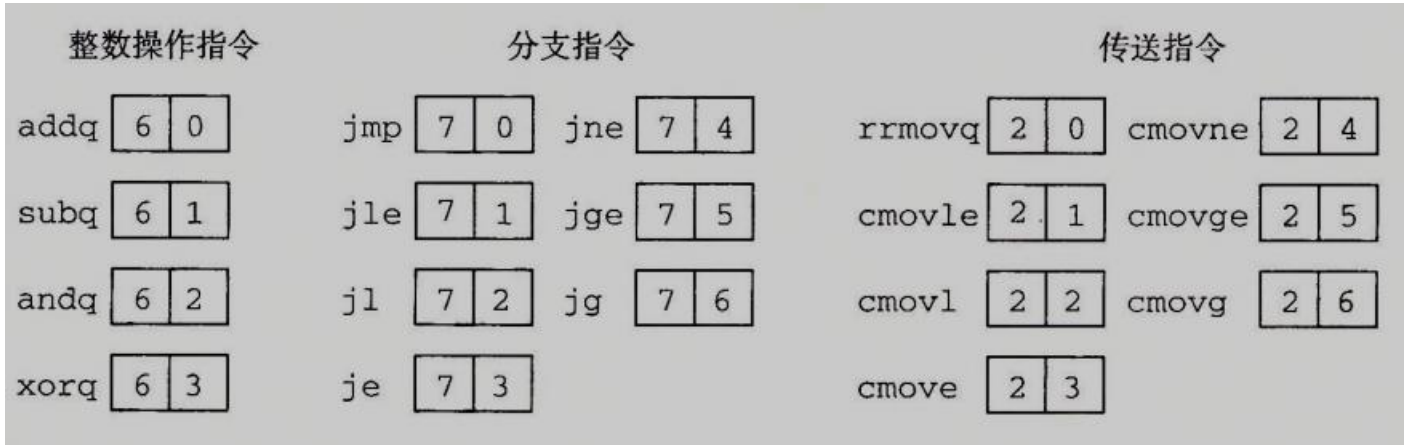


- 程序员：写汇编代码的人&编译器；
- 除%rsp外寄存器无固定含义；
- 只有15个寄存器；
- PC为程序计数器；
- Y86-64用虚拟地址来引用内存位置；
- Stat指示是正常运行还是出现了某种异常。



- 调用绝对地址;
- 使用小端法;
- halt使指令停止执行，在x86中会导致整个系统暂停运行因此没有;
- 内存引用方式为简单的基质和偏移量形式。

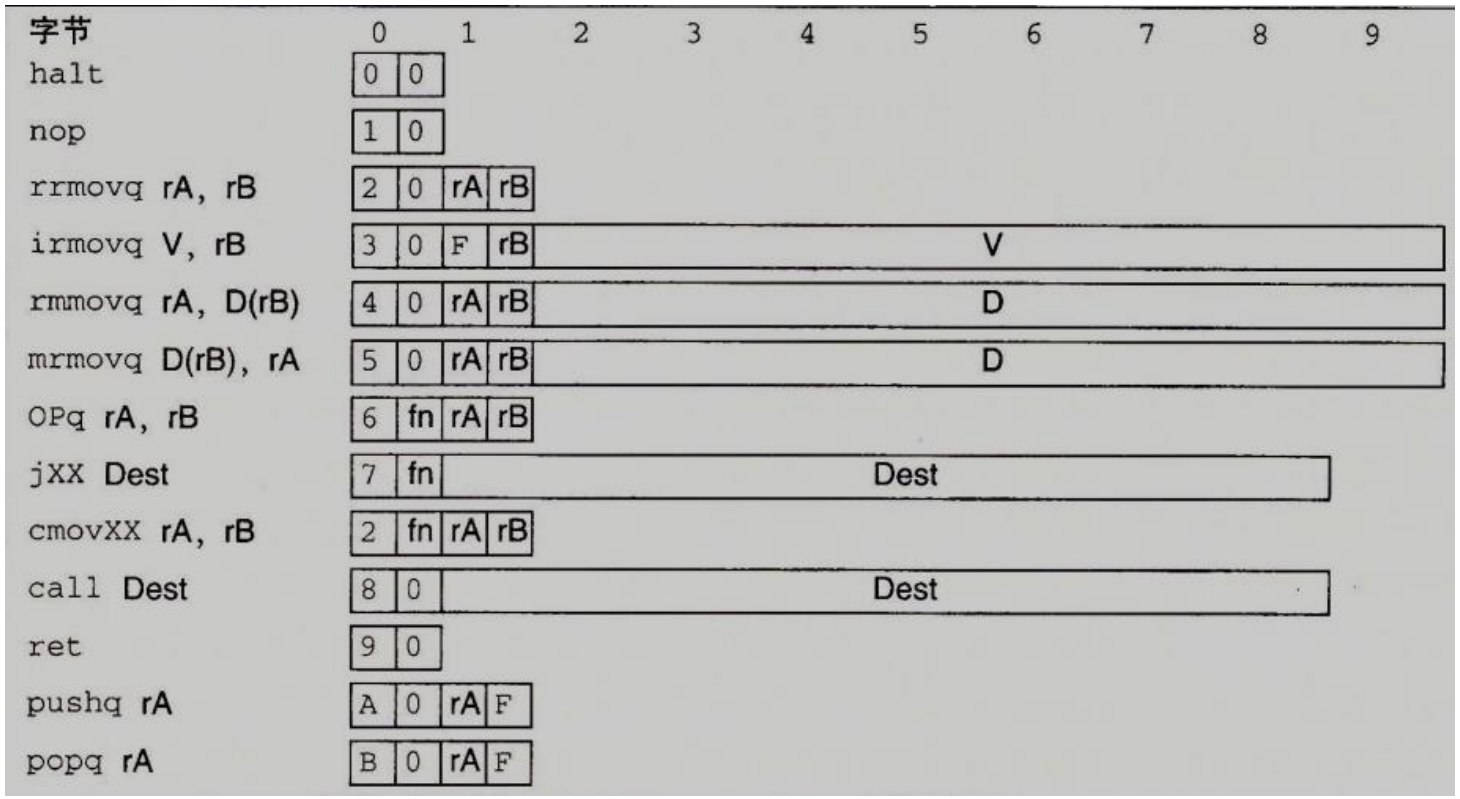





| 数字 | 寄存器名字 | 数字 | 寄存器名字 |
|---|---|---|---|
| 0 | %rax | 8 | %r8 |
| 1 | %rcx | 9 | %r9 |
| 2 | %rdx | A | %r10 |
| 3 | %rbx | B | %r11 |
| 4 | %rsp | C | %r12 |
| 5 | %rbp | D | %r13 |
| 6 | %rsi | E | %r14 |
| 7 | %rdi | F | 无寄存器 |

- 高4位是代码（code）部分，低4位是功能（function）部分；
- 程序寄存器存在于CPU的一个寄存器文件中；
- 当需要指明不应访问任何寄存器时，就用0xF表示。

- 加一个r15有什么问题吗？便于硬件设计？



- .pos 0x100 表明目标代码起始地址。

```
.pos 0x100  # Start code at address 0x100
    irmovq $15,%rbx
    rrmovq %rbx,%rcx
loop:
    rmmovq %rcx,-3(%rbx)
    addq   %rbx,%rcx
    jmp loop
```

- 遇到异常，Y86就简单地停止执行命令;
- 更完整的设计中，会调用异常处理程序。

| 值 | 名字 | 含义 |
| --- | --- | --- |
| 1 | AOK | 正常操作 |
| 2 | HLT | 遇到器执行 halt 指令 |
| 3 | ADR | 遇到非法地址 |
| 4 | INS | 遇到非法指令 |

x86-64 code

```
   long sum(long *start, long count)
   start in %rdi, count in %rsi
1  sum:
2    movl    $0, %eax        sum = 0
3    jmp     .L2             Goto test
4  .L3:                    loop:
5    addq    (%rdi), %rax    Add *start to sum
6    addq    $8, %rdi        start++
7    subq    $1, %rsi        count--
8  .L2:                    test:
9    testq   %rsi, %rsi      Test sum
10   jne     .L3             If !=0, goto loop
11   rep; ret                Return
```

Y86-64 code

```
   long sum(long *start, long count)
   start in %rdi, count in %rsi
1  sum:
2    irmovq $8,%r8           Constant 8
3    irmovq $1,%r9           Constant 1
4    xorq %rax,%rax          sum = 0
5    andq %rsi,%rsi          Set CC
6    jmp    test             Goto test
7  loop:
8    mrmovq (%rdi),%r10      Get *start
9    addq %r10,%rax          Add to sum
10   addq %r8,%rdi           start++
11   subq %r9,%rsi           count--.  Set CC
12 test:
13   jne    loop             Stop when 0
14   ret                     Return
```

图 4-6 Y86-64 汇编程序与 x86-64 汇编程序比较。Sum 函数计算一个整数数组的和。Y86-64 代码与 x86-64 代码遵循了相同的通用模式

- 算术指令中不能使用立即数；
- 需要将常数存到寄存器中；
- 需要两条指令完成寄存器与内存值的相加。

```
1  # Execution begins at address 0
2          .pos 0
3          irmovq stack, %rsp      # Set up stack pointer
4          call main               # Execute main program
5          halt                    # Terminate program
6
7  # Array of 4 elements
8          .align 8
9  array:
10         .quad 0x000d000d000d
11         .quad 0x00c000c000c0
12         .quad 0x0b000b000b00
13         .quad 0xa000a000a000
14
15 main:
16         irmovq array,%rdi
17         irmovq $4,%rsi
18         call sum                # sum(array, 4)
19         ret
20
21 # long sum(long *start, long count)
22 # start in %rdi, count in %rsi
23 sum:
24         irmovq $8,%r8           # Constant 8
25         irmovq $1,%r9           # Constant 1
26         xorq %rax,%rax          # sum = 0
27         andq %rsi,%rsi          # Set CC
28         jmp     test            # Goto test
29 loop:
30         mrmovq (%rdi),%r10      # Get *start
31         addq %r10,%rax          # Add to sum
32         addq %r8,%rdi           # start++
33         subq %r9,%rsi           # count--.  Set CC
34 test:
35         jne    loop             # Stop when 0
36         ret                     # Return
37
38 # Stack starts here and grows to lower addresses
39         .pos 0x200
40 stack:
```

```
# Execution begins at address 0
        .pos 0
        irmovq stack, %rsp      # Set up stack pointer
        call main               # Execute main program
        halt                    # Terminate program
```

从0处开始产生代码;
初始化栈指针。

```
# Stack starts here and grows to lower addresses
        .pos 0x200
stack:
```

栈从0x200开始向低地址增长。

- pushq %rsp 压入%rsp的原始值;
- popq %rsp后rsp中储存弹出的原始值;
- 与下图的原理矛盾，可能是为了保证功能的统一。

| 指令 | 效果 | 描述 |
| --- | --- | --- |
| pushq S | R[%rsp]←R[%rsp]−8;<br>M[R[%rsp]]←S | 将四字压入栈 |
| popq D | D←M[R[%rsp]];<br>R[%rsp]←R[%rsp]+8 | 将四字弹出栈 |

图 3-8 入栈和出栈指令

试着模仿写出pushq %rsp与popq %rsp时的“效果”?

- 我们的Y86-64是CISC还是RISC?
- 通常是CISC拥有更多寄存器还是RISC?

| CISC | 早期的 RISC |
| --- | --- |
| 指令数量很多。Intel 描述全套指令的文档[51]有 1200 多页。 | 指令数量少得多。通常少于 100 个。 |
| 有些指令的延迟很长。包括将一个整块从内存的一个部分复制到另一部分的指令，以及其他一些将多个寄存器的值复制到内存或从内存复制到多个寄存器的指令。 | 没有较长延迟的指令。有些早期的 RISC 机器甚至没有整数乘法指令，要求编译器通过一系列加法来实现乘法。 |
| 编码是可变长度的。x86-64 的指令长度可以是 1～15 个字节。 | 编码是固定长度的。通常所有的指令都编码为 4 个字节。 |
| 指定操作数的方式很多样。在 x86-64 中，内存操作数指示符可以有许多不同的组合，这些组合由偏移量、基址和变址寄存器以及伸缩因子组成。 | 简单寻址方式。通常只有基址和偏移量寻址。 |
| 可以对内存和寄存器操作数进行算术和逻辑运算。 | 只能对寄存器操作数进行算术和逻辑运算。允许使用内存引用的只有 load 和 store 指令，load 是从内存读到寄存器，store 是从寄存器写到内存。这种方法被称为 load/store 体系结构。 |
| 对机器级程序来说实现细节是不可见的。ISA 提供了程序和如何执行程序之间的清晰的抽象。 | 对机器级程序来说实现细节是可见的。有些 RISC 机器禁止某些特殊的指令序列，而有些跳转要到下一条指令执行完了以后才会生效。编译器必须在这些约束条件下进行性能优化。 |
| 有条件码。作为指令执行的副产品，设置了一些特殊的标志位，可以用于条件分支检测。 | 没有条件码。相反，对条件检测来说，要用明确的测试指令，这些指令会将测试结果放在一个普通的寄存器中。 |
| 栈密集的过程链接。栈被用来存取过程参数和返回地址。 | 寄存器密集的过程链接。寄存器被用来存取过程参数和返回地址。因此有些过程能完全避免内存引用。通常处理器有更多的(最多的有 32 个)寄存器。 |

- 发展趋势
- RISC引入了更多指令;
- CISC利用了高性能流水线结构;
- CISC可以保持比较好的向后兼容性，在桌面、便携计算机和基于服务器的计算领域中很有优势;
- RISC在嵌入式处理器市场（移动电话、汽车刹车、因特网电器等）上表现得非常出色，因为这些领域中降低成本和功耗比保持向后兼容性更重要。

9. 请比较 RISC 和 CISC 的特点，回答下述问题：

假设编译技术处于发展初期，程序员更愿意使用汇编语言编程来解决实际问题，那么程序员会更倾向于选用 _____ ISA。

假设你设计的处理器速度非常快，但存储系统设计使得取指令的速度非常慢（也许只是处理单元的十分之一）。这时你会更倾向于选用 _____ ISA。

A) RISC、RISC　B) CISC、CISC　C) RISC、CISC　D) CISC、RISC

9. 请比较 RISC 和 CISC 的特点，回答下述问题：

假设编译技术处于发展初期，程序员更愿意使用汇编语言编程来解决实际问题，那么程序员会更倾向于选用 _____ ISA。

假设你设计的处理器速度非常快，但存储系统设计使得取指令的速度非常慢（也许只是处理单元的十分之一）。这时你会更倾向于选用 _____ ISA。

A) RISC、RISC B) CISC、CISC C) RISC、CISC D) CISC、RISC

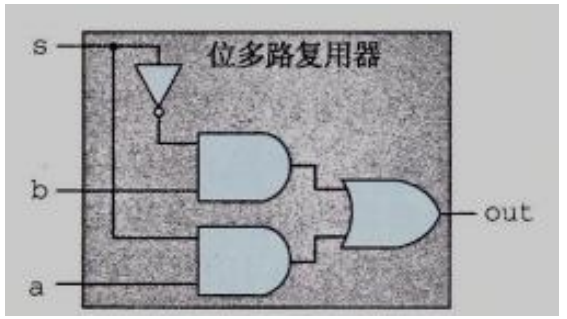
答案：B

//知识点 1：CISC 有更多的指令，有些更接近高级语言

//知识点 2：CISC 的指令功能更复杂，指令执行需要更多的周期，一定程度上可以平衡处理速度与指令访存的速度差异。不过，通常处理器设计中，主要通过多层次的存储体系结构来弥补两者之的速度差异。

# logic

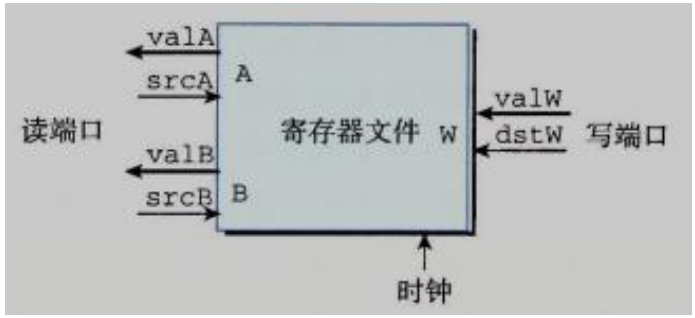
- 逻辑设计




- 硬件控制语言HCL

```
bool out = (s && a) || (!s && b);
```

- 存储器和时钟



- HCL —> HDL —> 有效的电路设计;
- 使用C语言风格的语法，但其真实含义与C不同;
- 后来的发展……

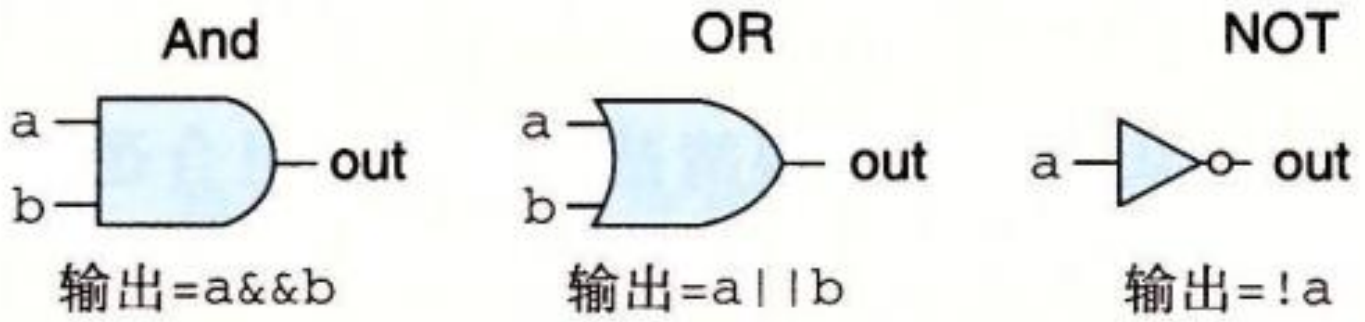
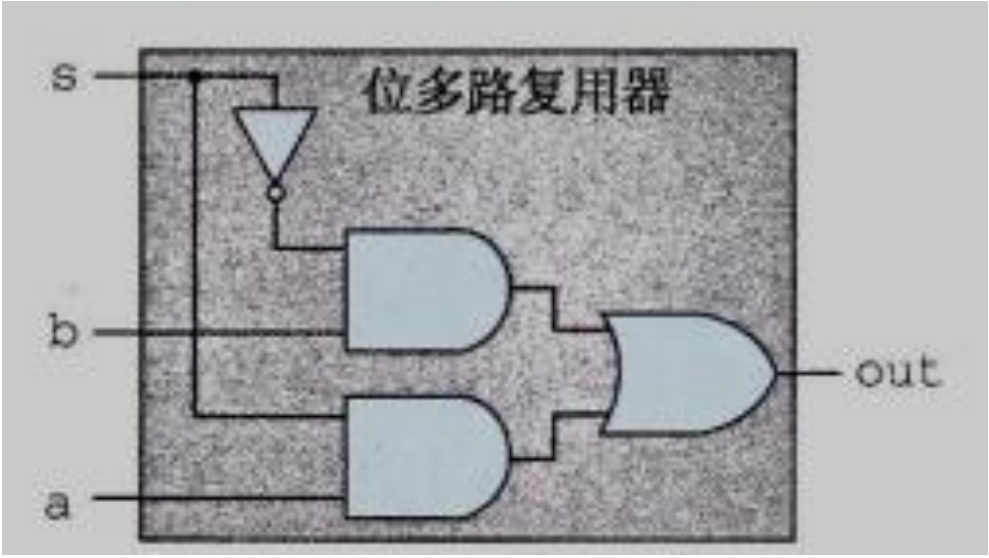



图 4-9 逻辑门类型。每个门产生的输出等于它输入的某个布尔函数

逻辑门总是活动的，一旦门的输入变化了，在很短的时间内其输出就会相应的变化。

注意当前的输入输出都是以位为单位，只有1或0两种情况。

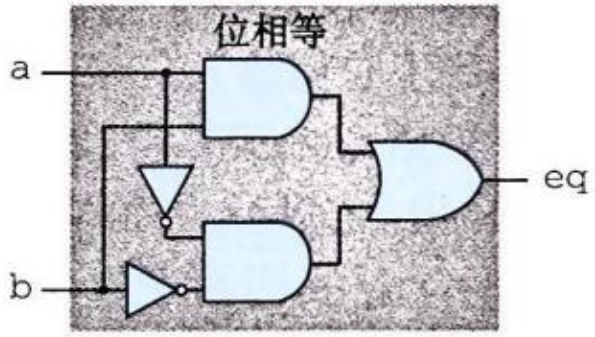
- 将多个逻辑门组合成一个网.....



# 限制:

- 每个逻辑门的输入必须是下述选项之一：1）一个系统输入（主输入），2）某个存储单元的输出，3）某个逻辑门的输出;
- 两个或多个逻辑门的输出不能连接在一起;
- 网必须是无环的。

# 检测位相等的组合电路




```
bool eq = (a && b) || (!a && !b);
```

- =相当于给结构起了一个名字;
- 输出时刻响应输入的变化，而不是像C那样执行到语句才进行求值;
- 逻辑门只对0和1进行操作;
- 组合逻辑没有部分求值的规定。

```
(a && !a) && func(b , c)
```

# 多路复用器（MUX）

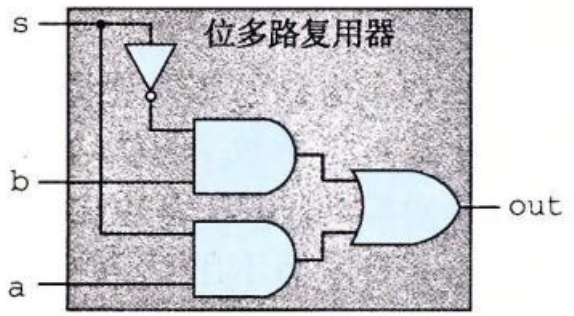



图 4-11 单个位的多路复用器电路。如果控制信号 s 为 1，则输出等于输入 a；当 s 为 0 时，输出等于输入 b

```
bool out = (s && a) || (!s && b);
```

# 字级的组合电路

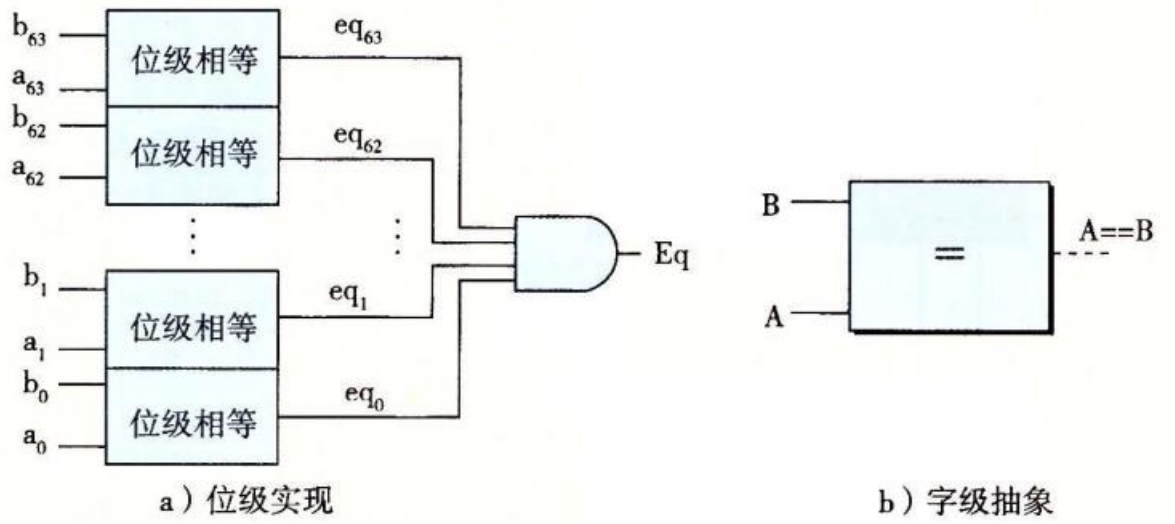
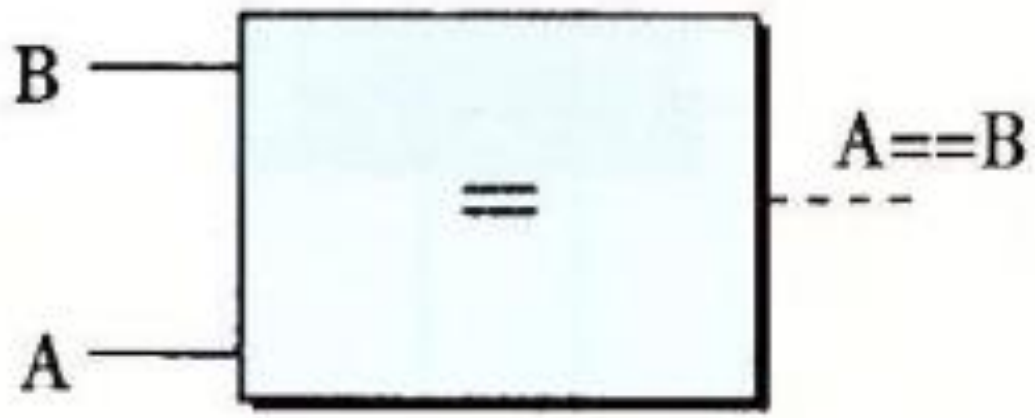



图 4-12 字级相等测试电路。当字 A 的每一位与字 B 中相应的位均相等时，输出等于 1。字级相等是 HCL 中的一个操作

```
bool Eq = (A == B);
```

# 字级的组合电路

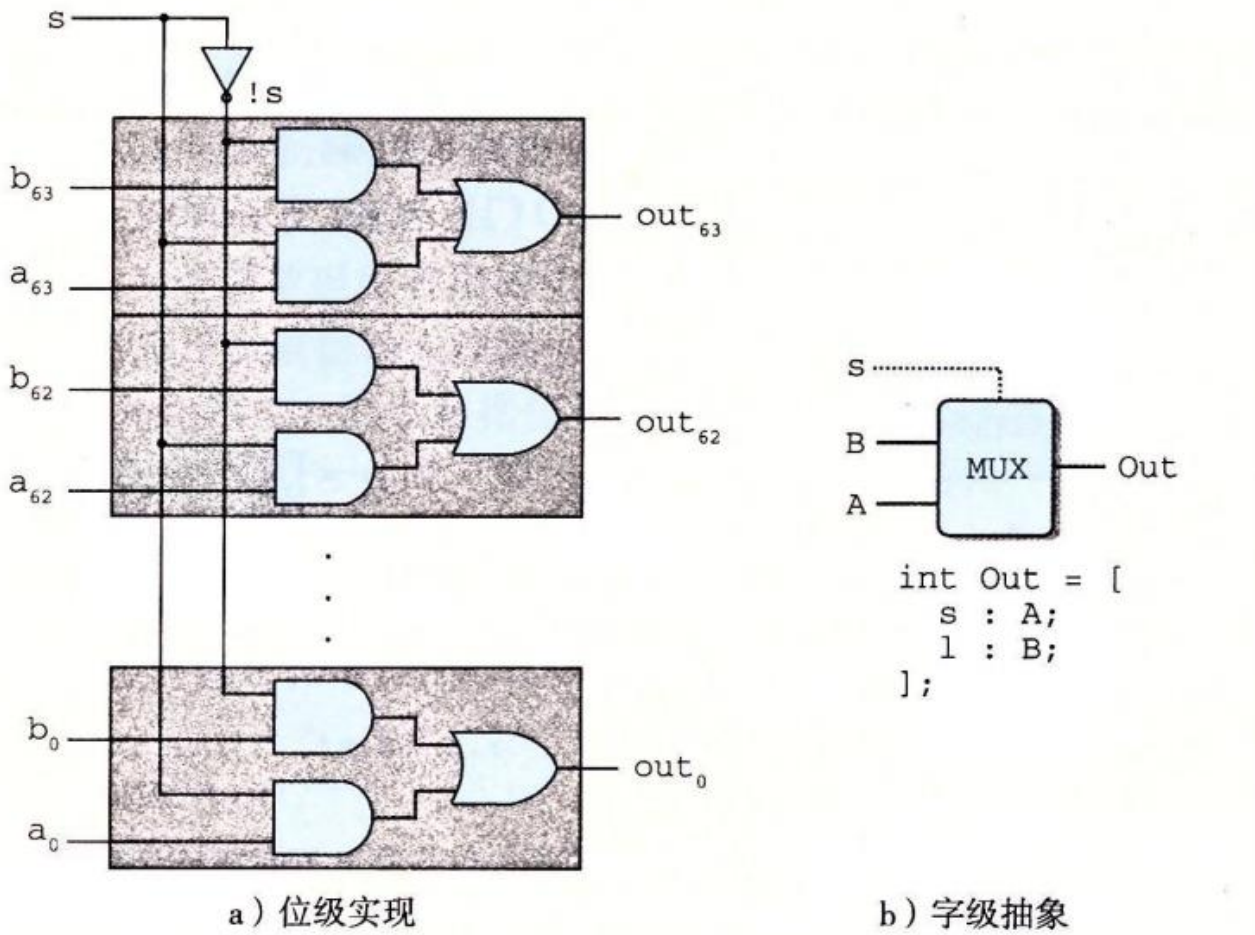



- 所有字级信号都声明为int，不指定字的大小（为了简单）；
- 中等粗度的线跑字，虚线跑位

```
bool Eq = (A == B);
```

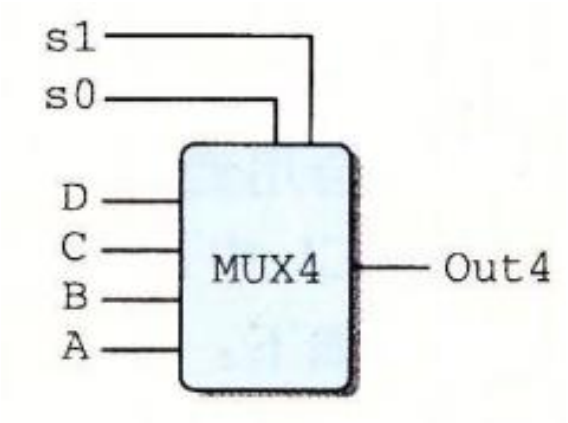
# 字级的多路复用器

```
word Out = [
    s : A;
    1 : B;
];
```




4-13 字级多路复用器电路。当控制信号 s 为 1 时，输出会等于输入字 A，否则等于 B。HCL 中用情况(case)表达式来描述多路复用器

# 情况表达式

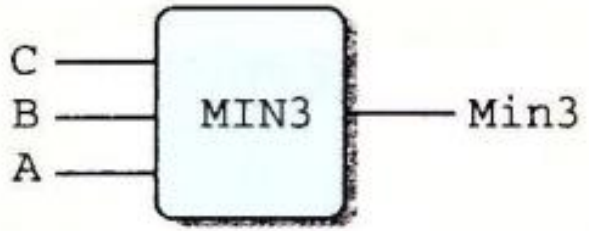



- 从上往下直到满足条件;
- 指定默认情况用1;
- 不要求不同的选择表达式之间互斥（虽然最后的硬件一定是互斥的）。

```
word Out4 = [
    !s1 && !s0 : A;
    !s1 : B;
    !s0 : C;
    1 : D;
    ];
```

```
[
    select_1  :  expr_1;
    select_2  :  expr_2;
    .
    .
    .
    select_k  :  expr_k;
]
```

# 情况表达式的小应用

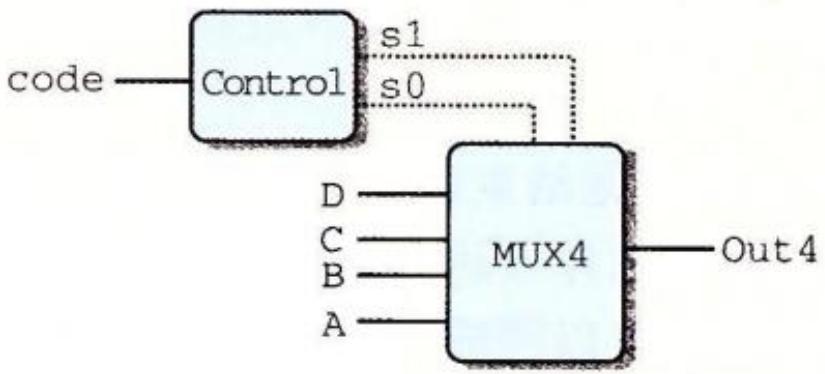



用 HCL 来表达就是：

```
word Min3 = [
        A <= B && A <= C : A;
        B <= A && B <= C : B;
        1                : C;
];
```

# 集合关系

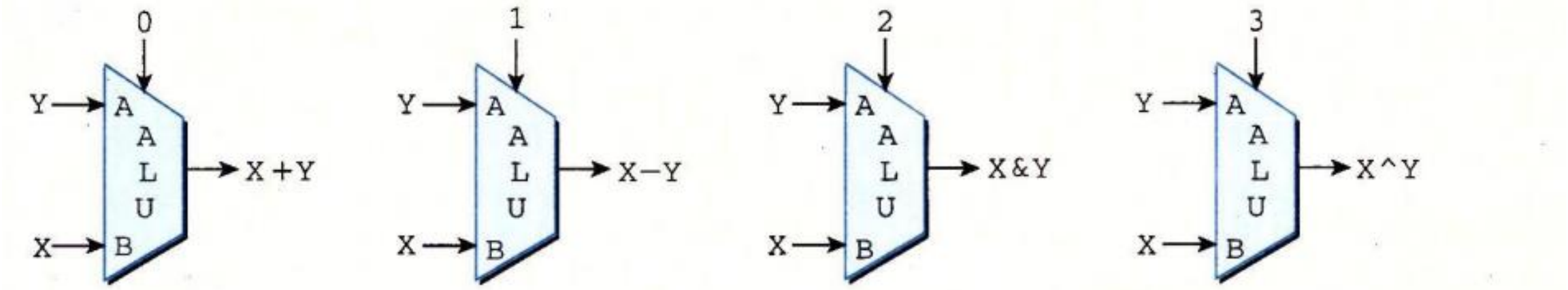
```
bool s1 = code == 2 || code == 3;
bool s0 = code == 1 || code == 3;
```




```
bool s1 = code in { 2, 3 };
bool s0 = code in { 1, 3 };
```

$$iexpr \text{ in } \{iexpr_1, iexpr_2, \cdots, iexpr_k\}$$

# 算数/逻辑单元（ALU）




5 算术/逻辑单元(ALU)。根据函数输入的设置，该电路会执行四种算术和逻辑运算中的一种

- 组合电路不能存储信息。
- 为了表达状态并在状态上进行运算，引入按位存储信息的设备；
- 需要时钟控制存储器的加载。

时钟寄存器（寄存器）

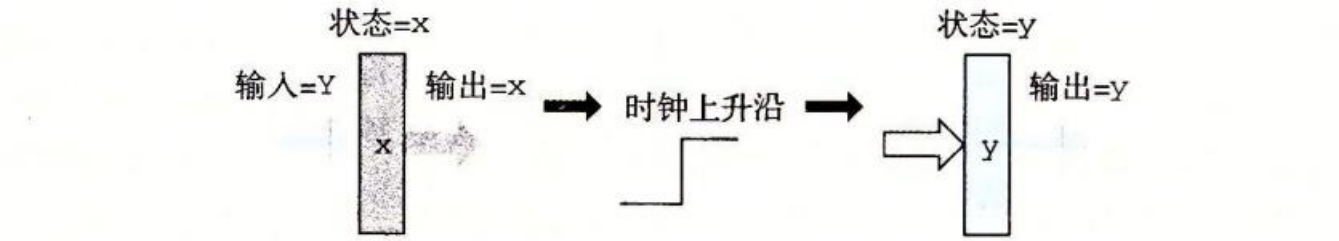
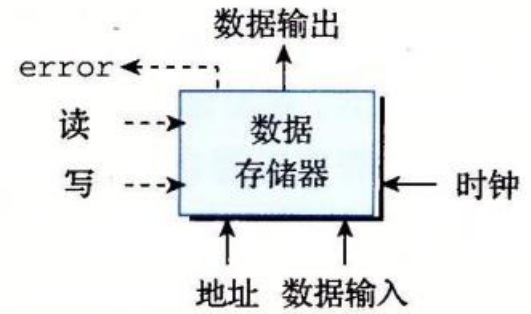


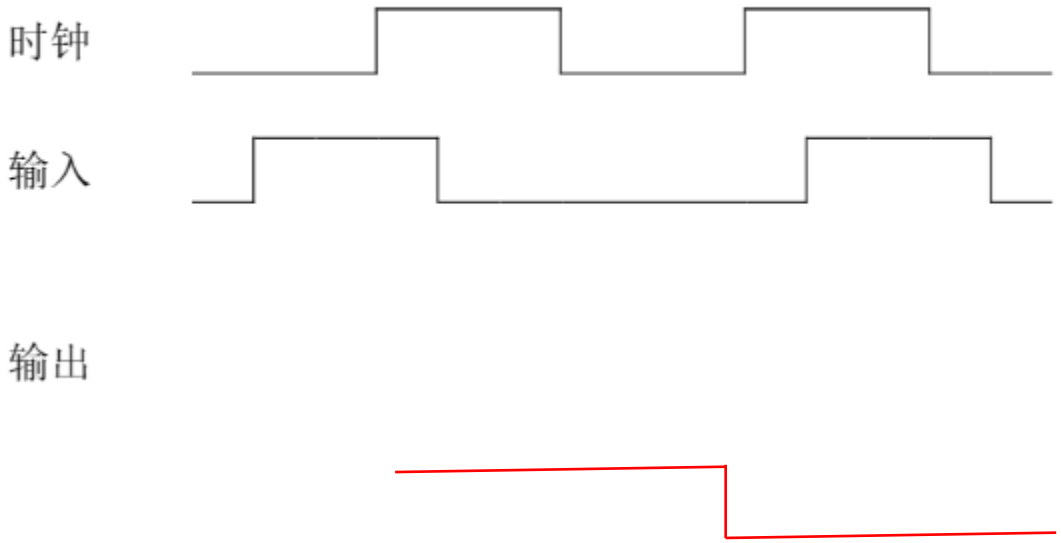

图 4-16 寄存器操作。寄存器输出会一直保持在当前寄存器状态上，直到时钟信号上升。当时钟上升时，寄存器输入上的值会成为新的寄存器状态

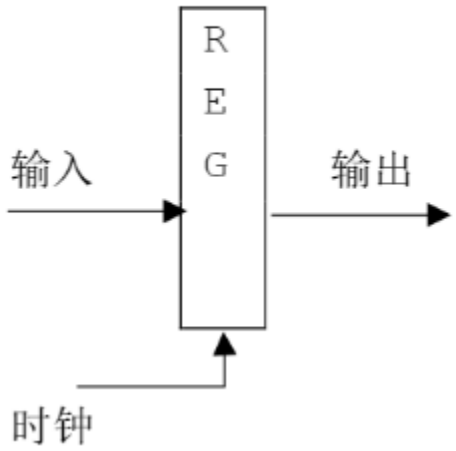
随机访问存储器（内存）



# Q12

3．下列寄存器在时钟上升沿锁存数据，画出输出的电平（忽略建立/保持时间）

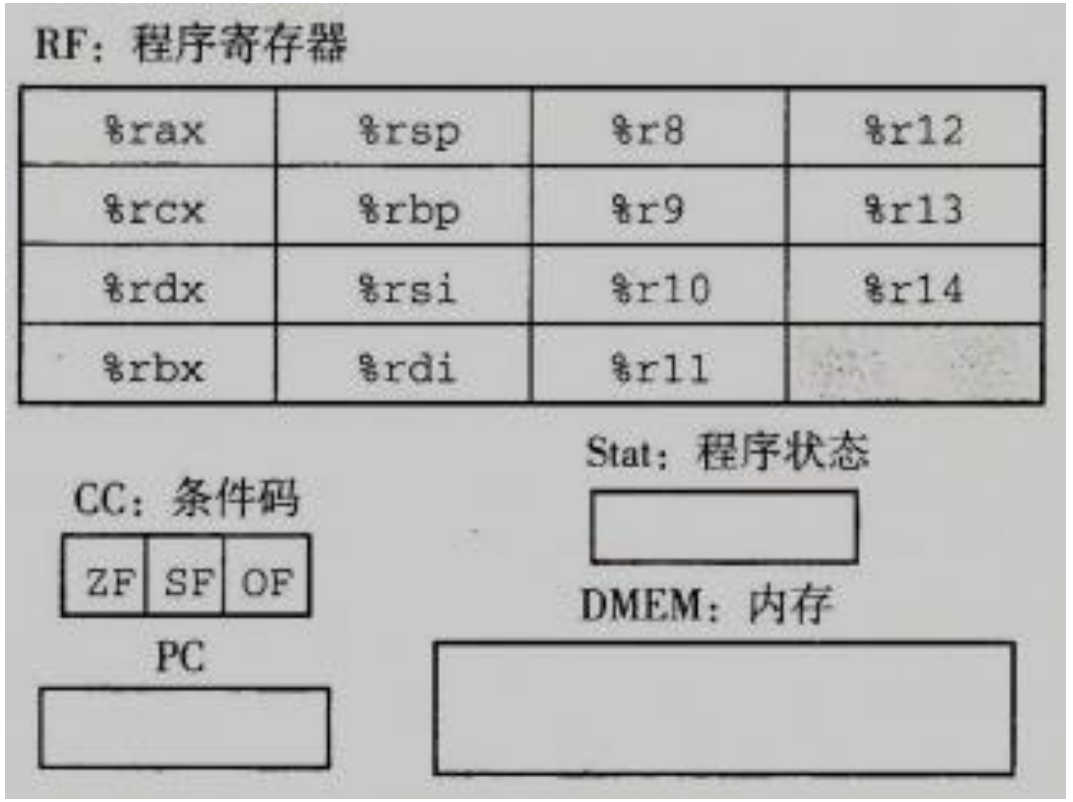


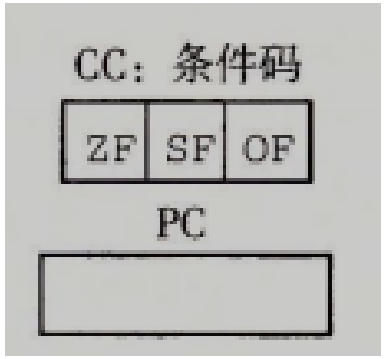

将相邻两个上升沿的输入连接起来即可

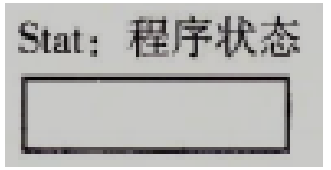
时钟寄存器（寄存器）

随机访问存储器（内存）

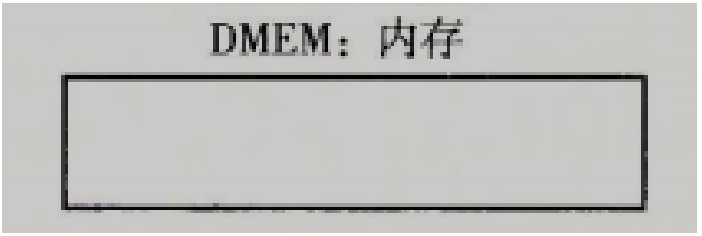


时钟寄存器（寄存器）



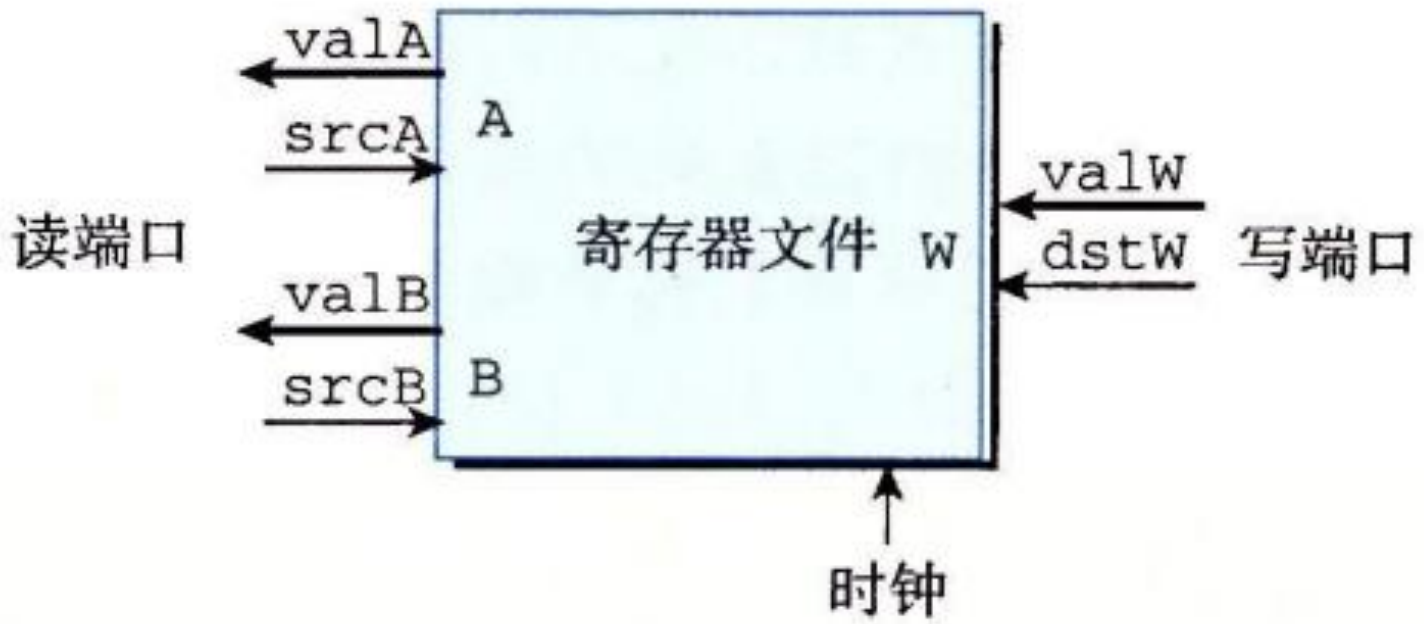




随机访问存储器（内存）



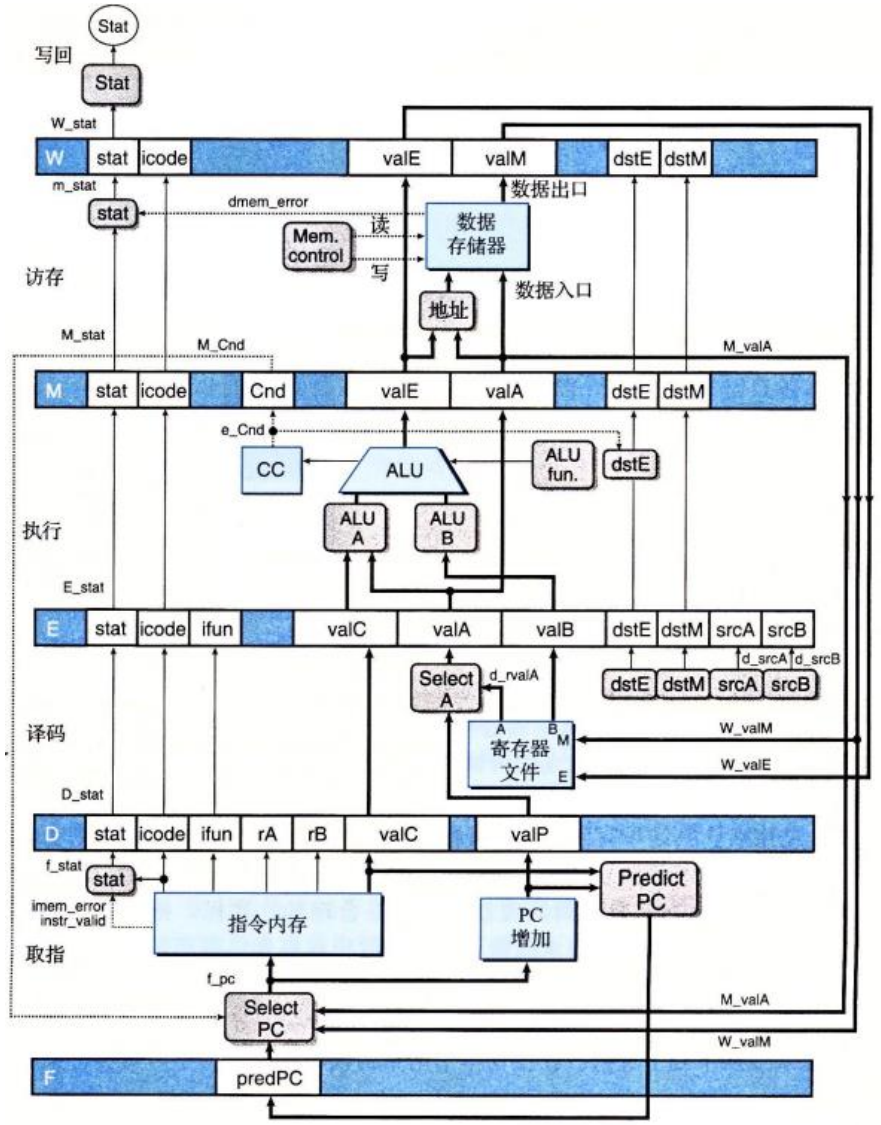

RF：程序寄存器

| | | | |
|---|---|---|---|
| %rax | %rsp | %r8 | %r12 |
| %rcx | %rbp | %r9 | %r13 |
| %rdx | %rsi | %r10 | %r14 |
| %rbx | %rdi | %r11 | |




当dstW为0xF时不写

# logic 硬件控制语言HCL　　组合电路　　存储器和时钟



- ISA与操统，到底是谁更靠近硬件？

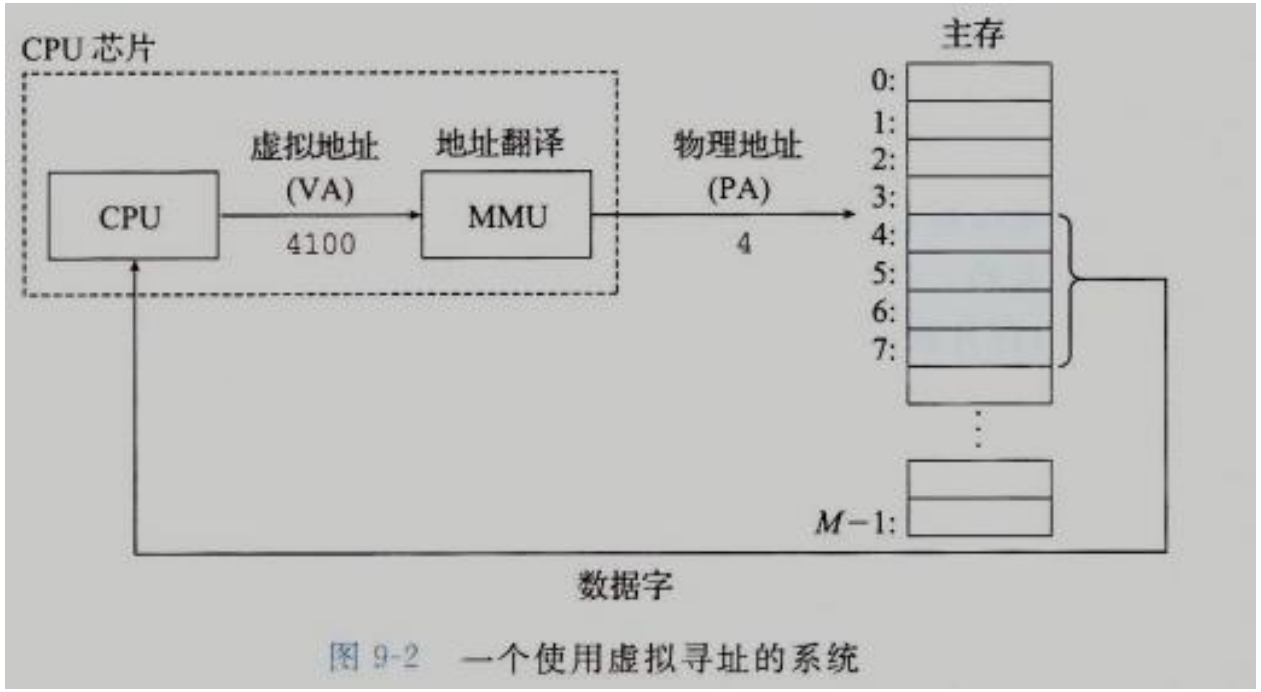
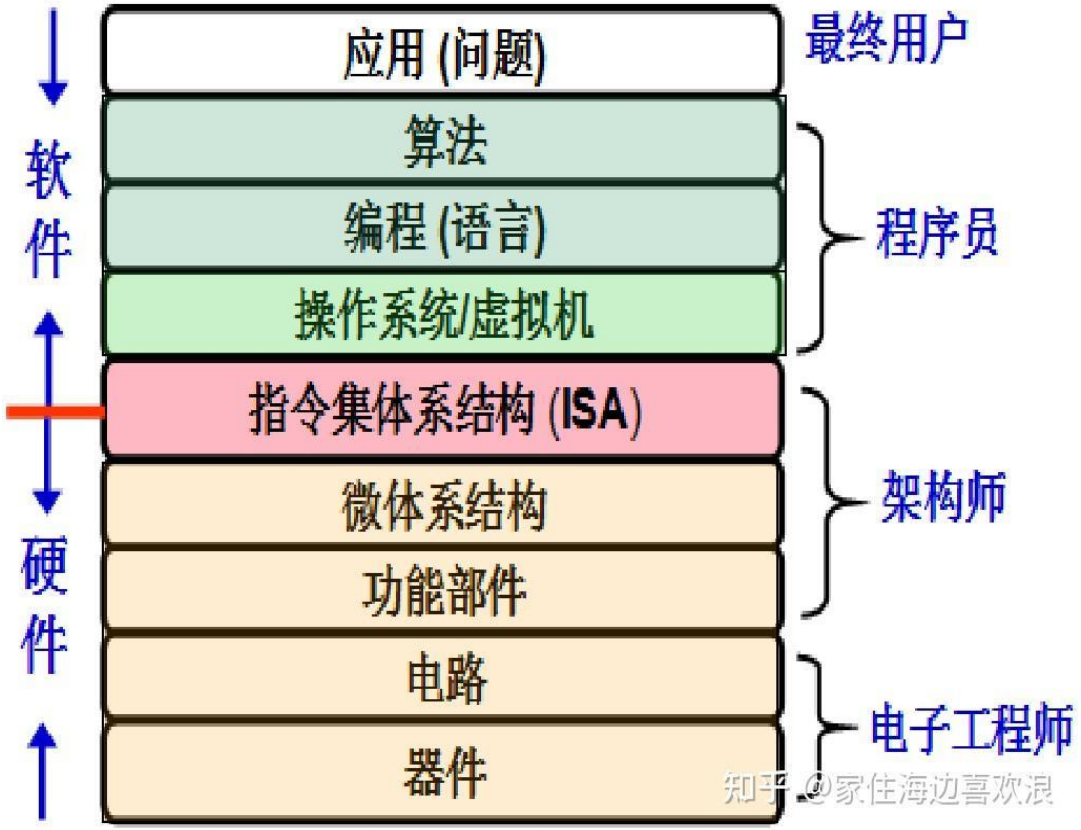


图 9-2 一个使用虚拟寻址的系统



# Practice

侯旭森

# The End