**4 Processor Architecture**

# 英语

|  |  |
| --- | --- |
| 1. dwarf使……相形见绌 2. manufacturer制造商 3. retrieval检索 4. embody体现 5. concise简洁的 6. assemble装配 7. mutually相互地 8. criteria标准 9. precede前面的 10. indefinitely无限期地 11. redundant多余的 12. preceding先前的 13. diminish减少 | 1. envision设想 2. while同时 3. intrinsic固有的 4. monolithic整体式的 5. contemporary当代的 6. acyclic无环的 7. exclusive专有的 8. periodic周期性的 9. diagramm图解 10. idiosyncrasy特质 11. spray喷涂 12. reciprocal倒数、互惠的 13. appreciate欣赏、感激、了解 |

**|** *The instructions supported by a particular processor and their byte-level encodings are known as its instruction set architecture (ISA)*

*Each manufacturer produces processors of ever-growing performance and complexity, but the different models remain compatible at the ISA level.*

*The ISA provides a conceptual layer of abstraction between compiler writers, who need only know what instructions are permitted and how they are encoded, and processor designers, who must build machines that execute those instructions.*

指令集架构在编译器编写者和处理器设计者之间提供一层抽象概念层

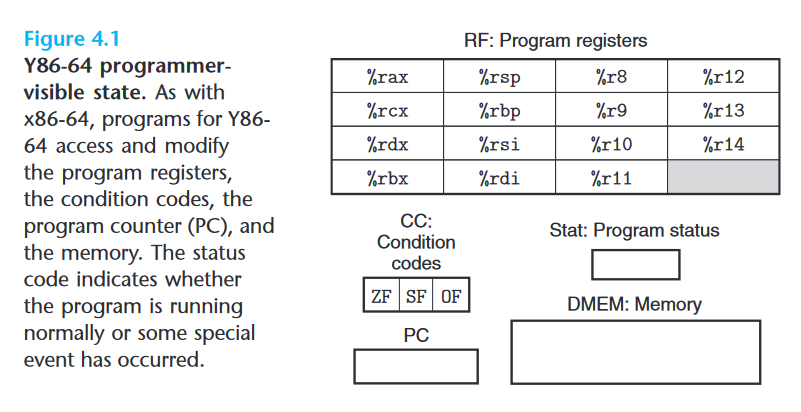
编译器编写者只需要知道架构内存在哪些指令以及这些指令是怎么被编码的

处理器设计者则必须要设计执行这些指令的机器

# 4.1 *The Y86-64 Instruction Set Architecture*

**|** 可以将Y86-64视为X86-64的精简版，包括状态的不同组成部分，指令集以及指令集的编码格式、编程惯例和异常处理机制

## 4.1.1 *Programmer-Visible State*



Y86-64中的每条指令都可以读或者修改处理器的一些状态，这些状态称为“程序员注释1可见状态”

Y86-64的处理器状态包括：15个64位的寄存器、ZF、SF、OF三个状态码、PC、Stat、主存

15个寄存器包括%rax、%rcx、%rdx、%rbx、%rsp、%rbp、%rsi、%rdi、%r8~%r14

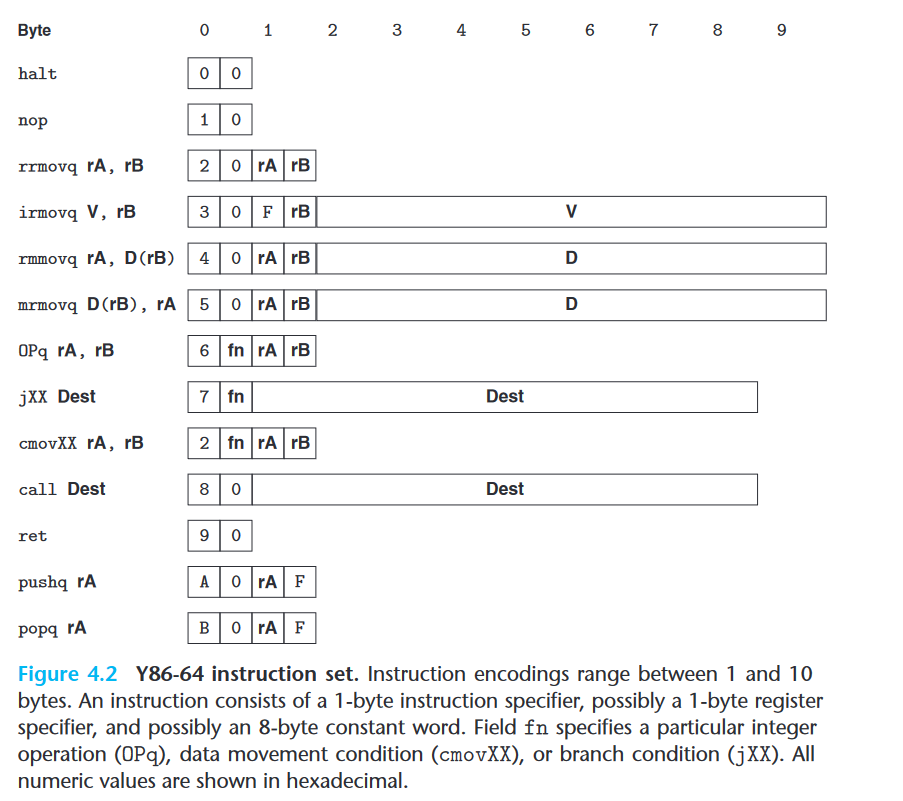
ZF、OF、SF三个用于整数算术、逻辑运算效果的指令

PC存储着当前正在执行的指令的地址

Stat状态码表示程序执行的整体状态，可以表示当前程序是在正常操作执行还是发生某种异常

主存在概念上是存储程序和数据的大的字节数组，Y86-64使用虚拟存储，由操作系统和硬件合作将虚拟地址翻译为物理地址

## 4.1.2 *Y86-64 Instructions*



Y86-64所支持的指令集如上图所示，指令所操作的对象是8字节整数

1. 将movq指令按照源和目的操作数的形式划分为irmovq、rmmovq、mrmovq、rrmovq

其中源操作数可以是i立即数、r寄存器、m存储器(指令助记符的第一个字符)，目的操作数可以是r寄存器、m存储器(指令助记符的第二个字符)

存储器操作数的内存引用方式是简单的基址和偏移量形式，不支持变址和缩放的内存引用形式

同x86-64，不支持内存和内存之间的数据直接传递，也不支持将立即数直接传送到内存

1. OPq指令表示基于整数的四种运算指令：addq、subq、andq、xorq

不像x86-64允许整数运算指令直接操作存储器数据，OPq指令仅仅在寄存器数据上操作

OPq指令可以设置SF、ZF、OF条件码位

1. jXX指令表示7种跳转指令：jmp、jle、jl、je、jne、jge、jg

跳转指令执行跳转的条件同x86-64

1. cmovXX表示6种条件传送指令：cmovle、cmovl、cmove、cmovne、cmovge、cmovg

cmovXX指令的格式同rrmovq，是寄存器和寄存器之间的数据传送

1. call和ret

call指令将返回地址入栈，并将目的地址送PC

ret指令则取返回地址送PC

1. pushq和popq

同x86-64的pushq、popq

pushq先将%rsp减8然后入栈数据，popq先取%rsp内存引用上的数据，然后加%rsp 8

1. halt指令暂停指令的执行

x86-64中有类似的指令hlt，但是并不允许应用程序执行该条指令，因为它会暂停整个系统的操作

在Y86-64中，halt指令的执行会使得处理器停止，设置状态码为HLT

## 4.1.3 *Instruction Encoding*

[行内引用](https://www.wolai.com/9YFhQHtAXYeR1vApt5a9Ad#bfeH8u7JiqiB1dwofAHX2K)Figure4.2也展示出了Y86-64指令集的位级编码：

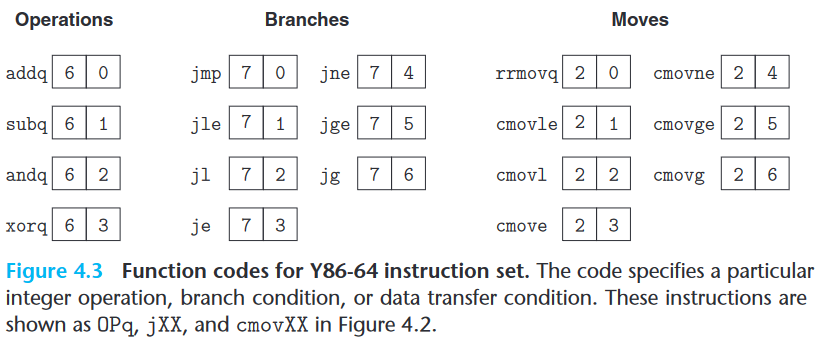
* 1. 指令编码长度范围在1字节到10字节之间
  2. 每条指令都以1字节的指令指示符开始，可能会有1字节的寄存器指示符、8字节的常量
  3. fn字段域表示了一个特定的整数运算或者是条件传送或者是分支指令

### *Instruction specifier*

每条指令都有一字节的指令指示符来表示当前指令类型。这一字节被分为两个4位部分：高4位表示代码，低4位表示功能

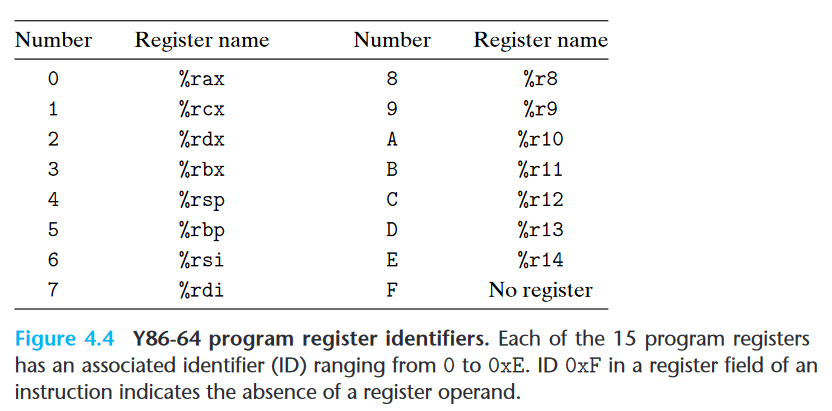
代码字段的值范围在0~0xB之间

功能字段仅用于区分一组共享代码字段的指令集合的某个指令，如下



值得注意的是rrmovq与cmovXX具有同样的格式，可以被视为jmp类的cmov，它的功能字段值恰好为0

### *Register specifier*



寄存器指示符的编码是从0x0~0xE。Y86-64和X86-64寄存器编号是匹配的(0xF为%r15)

**在Y86-64中的0xF表示不访问寄存器**，如irmovq、popq、pushq这些只需要一个寄存器操作数的指令，另一个寄存器操作数字段的值即为0xF

### 8B Constant

8B的常量字可以作为irmovq中的立即数、rmmovq和mrmovq中的内存引用的位移以及分支和调用的目的地。这里需要注意的是Y86-64中的分支和调用目的地址以绝对地址给出——X86-64中是采用PC相对寻址。且与X86-64一样，所有整数采用小端编码

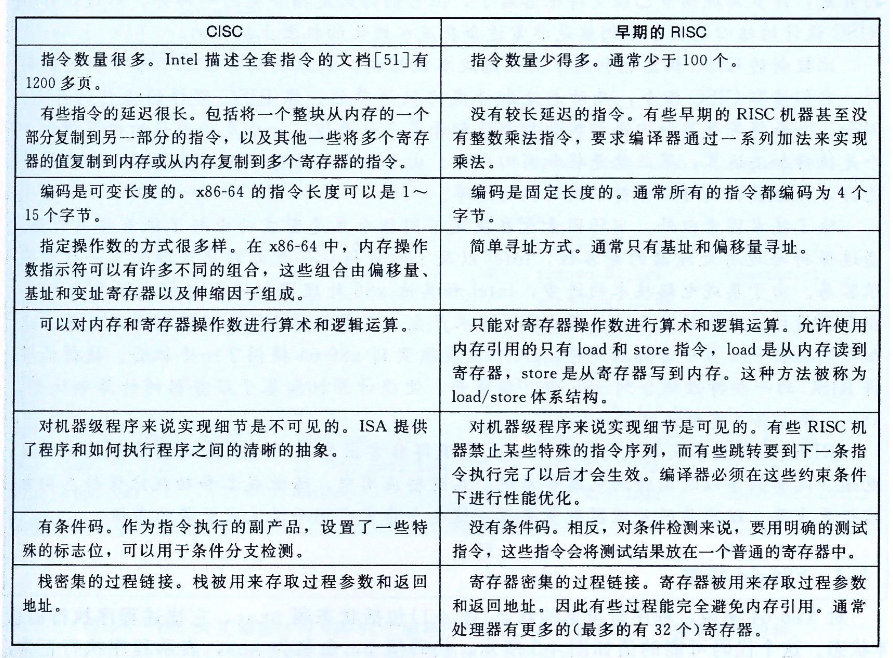
### 示例

例如指令“rmmovq %rsp,0x123456789abcd(%rdx)”的机器编码为"0x4042cdab896745230100"

✨*One important property of any instruction set is that the byte encodings must have a unique interpretation. An arbitrary sequence of bytes either encodes a unique instruction sequence or is not a legal byte sequence.*

Y86-64就是这样，只要知道指令指示符这一字节我们就能决定其他附加字节的长度和含义。但是如果不知道一段代码序列的起始位置，我们就不能准确地确定怎样将序列划分成单独的指令

### RISC and CISC

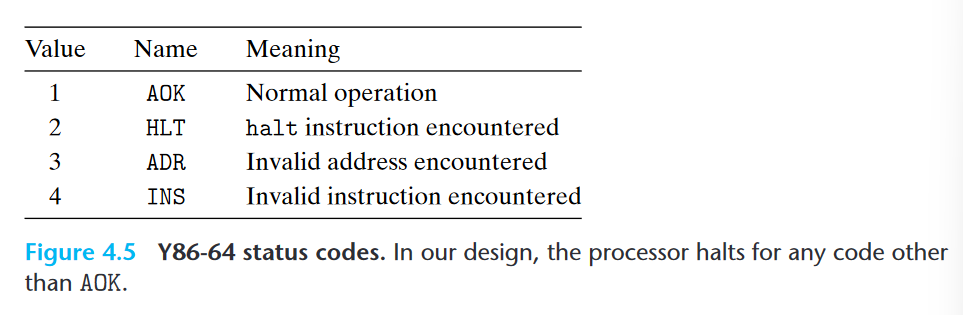


而Y86-64指令集既有CISC指令集的属性也有RISC指令集的属性。和CISC一样，它有条件码，长度可变的指令，并用栈来保存返回地址。和RISC一样的是，采用load-store机制和规则编码，通过寄存器来传参——Y86-64可以看作是采用了CISC x86-64指令集，但利用RISC的一些规则进行了简化

RISC和CISC的发展趋势是相结合的，而且各在不同的市场上表现得很出色

## 4.1.4 *Y86-64 Exception*

在4.1.1[行内引用](https://www.wolai.com/9YFhQHtAXYeR1vApt5a9Ad#8avVKS5oFQqkz6CNBJZa4z)中讲到，程序员的可见状态中包含一个Stat状态码位，其值如下：



值为1时，表示AOK，即表示程序正常执行

值为2时，表示HLT，是由指令halt引起的，表示处理器暂停执行指令

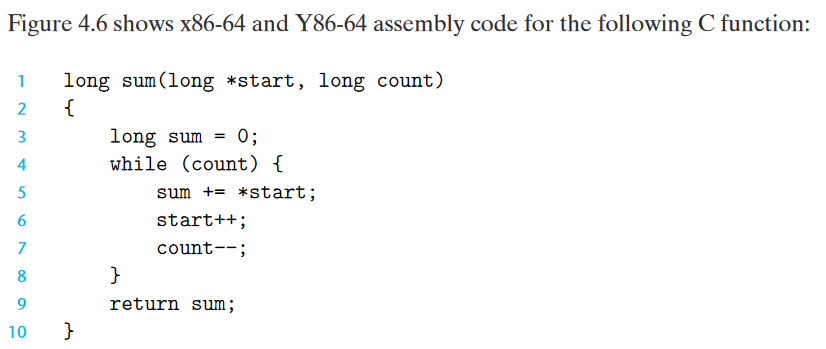
值为3时，表示ADR，即处理器在获取指令或读取或写入数据时尝试读取或写入无效的内存地址

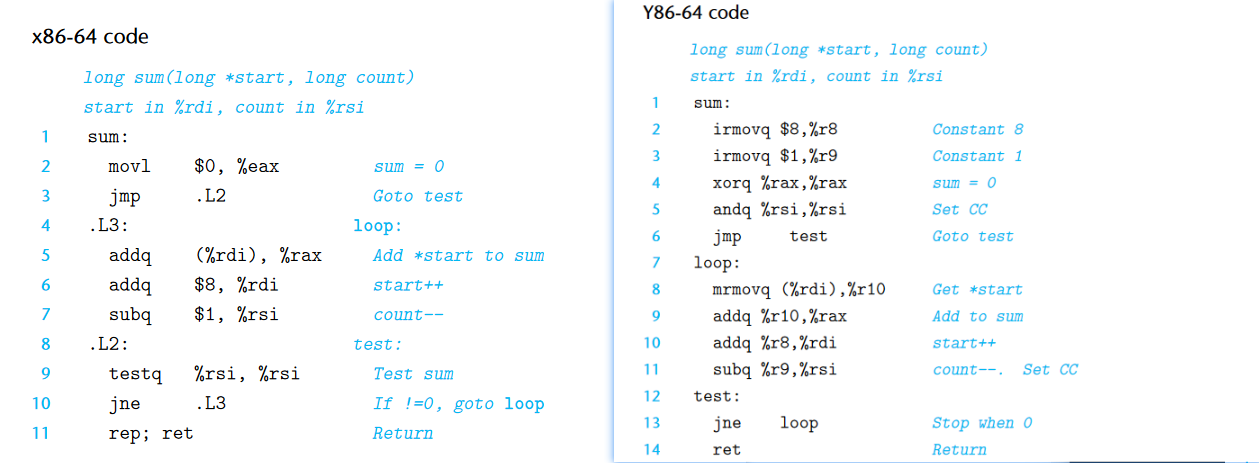
根据具体的实现细节限制最大的地址范围，超出此范围的地址的访问均会触发ADR异常

值为4时，表示INS，即无效的指令

## 4.1.5 *Y86-64 Procedure*

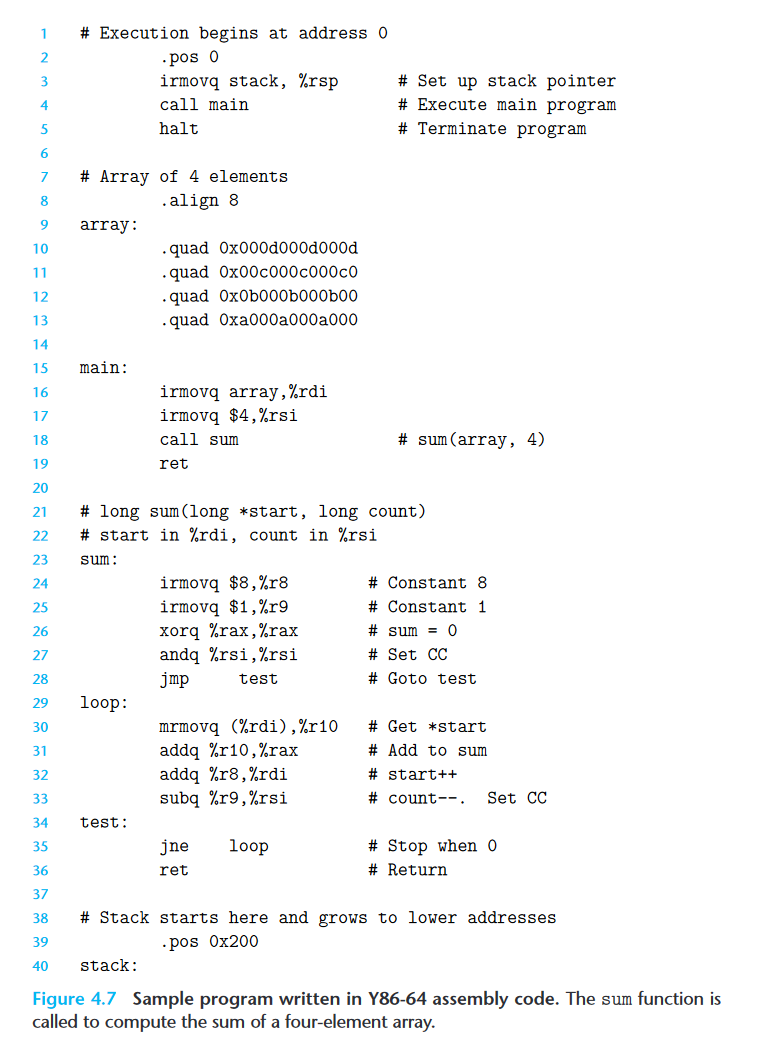
对于下列C函数，观察它的x86-64汇编代码和Y86-64汇编代码：





可以看到以下几点不同：

1. Y86-64是load-store架构，所有的算术运算均是寄存器和寄存器之间的，不存在寄存器和立即数运算，因此需要2-3行的汇编代码irmovq；也不存在存储器操作数和寄存器之间的运算，因此需要第8行的mrmovq

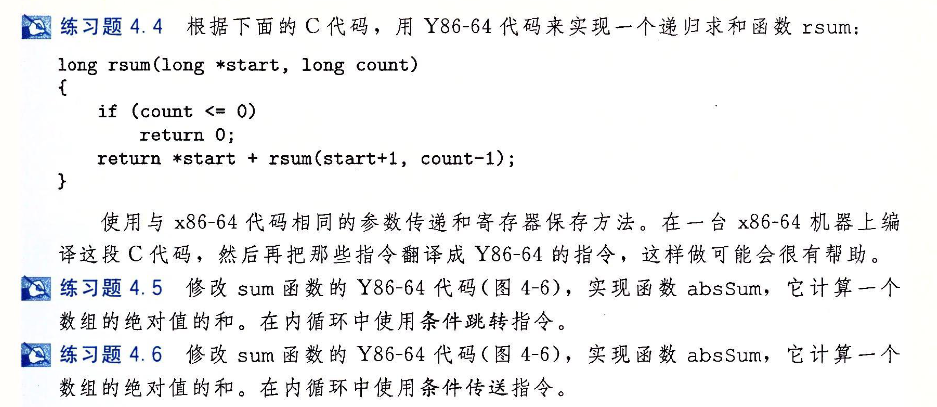


其中"."开头的是汇编器伪指令，告诉汇编器调整生成代码的地址或插入一些数据字

.pos 0表示汇编器应该从地址0开始生成代码——所有Y86-64汇编程序均以该伪指令开始

.align 8表示汇编器插入数据的对齐偏移量为8B

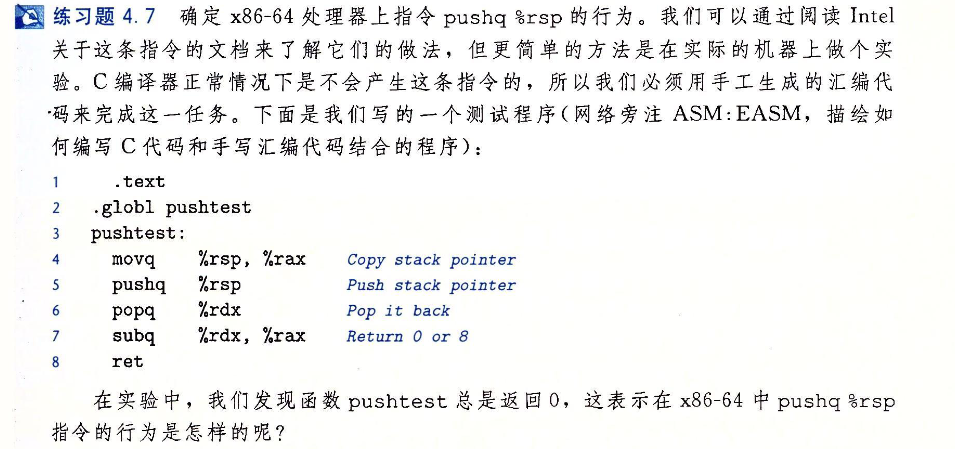
在实验的过程中会用到yas和yis



## 4.1.5 *Some Y86-64 Instruction Details*

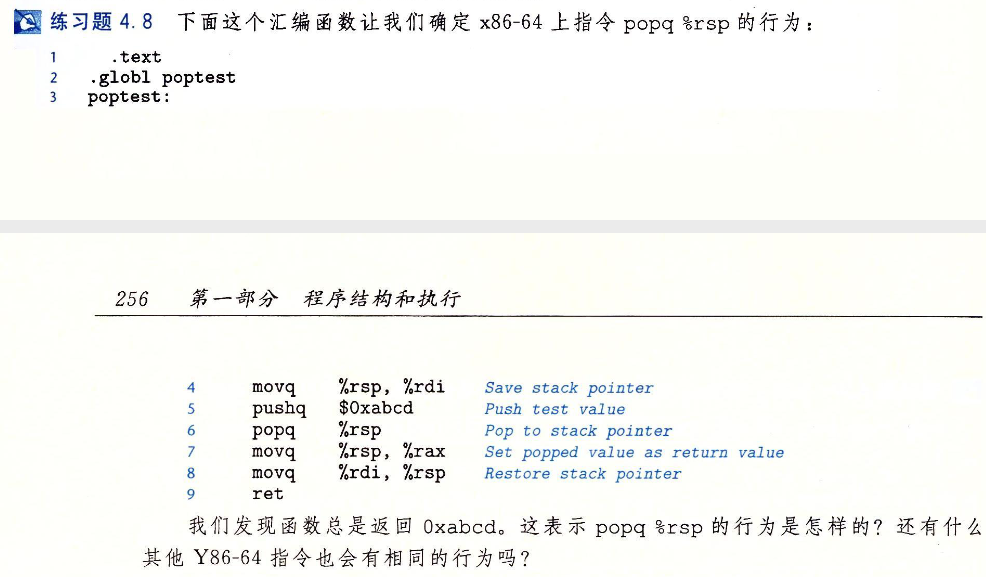
需要注意两种特别的指令的组合

1. pushq %rsp是存放原来的%rsp还是减8之后的%rsp



总返回0说明，**push %rsp保存的是原来的%rsp即未减8的**

1. popq %rsp是恢复栈中的值还是原来的%rsp+8



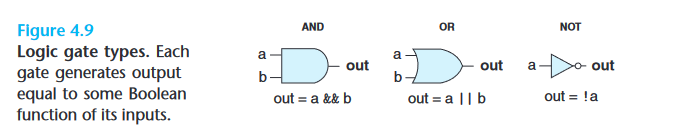
返回的总是0xabcd，表明**popq %rsp得到的是栈中的值**

# 4.2 *Logic Design and the Hardware Control Language HCL*

大多数现代的逻辑设计电路将逻辑1设置为1.0v的高电平，将0设置为0.0v的低电平。实现数字系统需要三个主要组件：用于计算位功能的组合逻辑电路、用于存储位的存储元件以及用于调节存储元件更新的时钟信号。

HCL硬件控制语言是用来描述硬件设计的控制部分，只表示有限的操作集合。使用将HCL直接转换为Verilog的工具，将转换后的代码与基本的硬件单元的Verilog代码结合起来，就能产生HDL硬件描述语言，从而合成能够实际能够工作的微处理器

## 4.2.1 Logic Gates



逻辑门是数字电路的基本计算元件，逻辑门的输出是其输入位值的布尔函数，如与AND、或OR、非NOT。上图即为逻辑门部件的示意图以及HCL描述。

注意这里使用&&、||、!代替C中的&、|、~。因为逻辑门是对单个位，而不是整个字进行运算的。

✨逻辑门总是处于激活状态，当输入改变时，在很短的时间内，输出也会随之改变

## 4.2.2 *Combinational Circuits and HCL Boolean Expressions*

通过将多个逻辑门组装成一个网络，就可以构建称为组合电路的计算块。但是网络搭建的方式需要受到一定的限制：

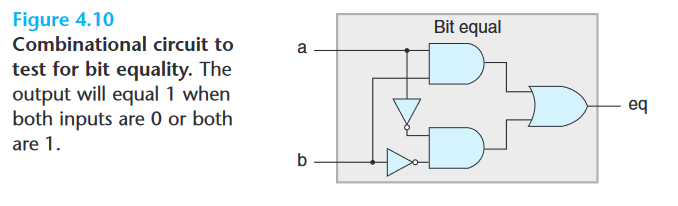
* 1. 逻辑门的输入必须是“一个系统输入（称为主输入）”、“某个相连接的存储器单元的输出”、“某个逻辑门的输出”
  2. 两个及其以上的逻辑门的输出不能连接在一起

两个或以上的逻辑门可能将导线驱向不同的电压，从而导致无效电压或者电路故障

* 1. 网络必须是无环的

如果网络中存在一条回路，那么会引起所表示的函数功能的歧义

下图的逻辑电路可以判断两个位是否相等，其对应的HCL为

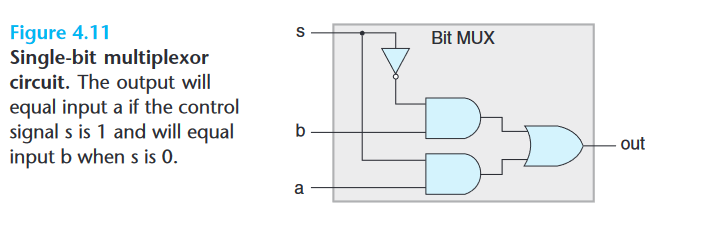


bool eq=(a&&b)||(!a&&!b)

C

✨HCL使用C风格的语法，通过“=”将信号名称与表达式相关联。然而，与 C 不同的是，我们并不将其视为执行计算并将结果分配到某个内存位置。相反，它只是一种为表达式命名的方法

下图的逻辑电路可以实现一位多选器，其对应的HCL为



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

C

HCL表达式和C表达式有着很明显的相似之处

它们都使用布尔运算来计算其输入的函数

但是也有一些差异：

* 1. 组合逻辑电路HCL表达式当输入改变时，输出即改变  
     C表达式只有当程序执行到该处时才进行计算输出
  2. C的逻辑表达式可以表示范围内的任意整数——将0视为False，非0视为True。而组合逻辑HCL只有0和1两个位值
  3. 当C逻辑表达式中的第一部分可以被计算时，第二部分即不进行执行。而组合逻辑HCL只要输入改变，输出就会进行改变。

例如在C中并不会调用func函数，而在HCL中，只要a/b/c有改变就都会计算

## 4.2.3 *Word-Level Combinational Circuits and HCL Integer Expressions*

**|** *Combinational circuits that perform word-level computations are constructed using logic gates to compute the individual bits of the output word, based on the individual bits of the input words.*

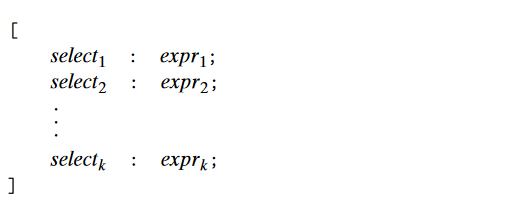
|  |  |
| --- | --- |
|  | Figure4.12展示了一个用于判断两个64位字A、B是否相等的组合逻辑电路  是通过1位判断相等的逻辑电路来判断64位字的每个位是否相等，最后将每位的判断结果相与得到最终的判断结果——为1表示相等，为0表示不等  其HCL如下  bool Eq=(A==B);  C |

✨在 HCL 中，我们将任何字级信号声明为 int，而不指定字大小

在功能齐全的HDL中，每个字都可以声明为具有特定的位数

|  |  |
| --- | --- |
|  | Figure4.13展示了字级的多路选择器，同字级的比较器一样，是对字的每一位运行一位多路选择器，然后最后输出等同字大小的数据  其HCL如下：  int Out=[  s:A;  1:B; ]  C |

✨多路选择器的HCL描述使用case表达式，其通用的格式为：



表达式包含着一系列的case语句，每一个case i包含着一个布尔表达式和其对应的值

和C中的switch-case不同，不要求不同的选择表达式是互斥的。从逻辑上讲，选择表达式按顺序求值，并选择第一个产生 1 的情况注释2。如Figure4.13的HCL

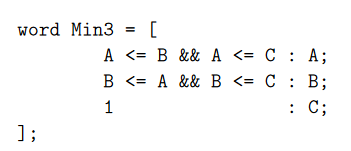
可以是任意的布尔表达式也可以是任意的整数

比如四路多路选择器的HCL为

word Out4=[  
 !s1&&!s0:A;#00  
 !s1:B;#01 !S1的另一种情况已在第一个判断，则不满足即进入第二个  
 !s0:C;#10  
 1:D;  
];

C

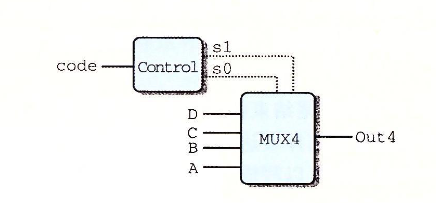
比如三路最小的HCL为



## 4.2.4 *Set Membership*

在处理器设计中，很多时候都需要将一个信号与许多可能匹配的信号作比较，以此来检测正在处理的某个指令代码是否属于某一类指令代码

例如，生成四路选择器的选择信号s1、s0：



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

C

采用集合的通用表达形式：则对上式可简化为：

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

C

## 4.2.5 *Memory and Clocking*

**|** *Combinational circuits, by their very nature, do not store any information. Instead, they simply react to the signals at their inputs, generating outputs equal to some function of the inputs. To create sequential circuits—that is, systems that have state and perform computations on that state—we must introduce devices that store information represented as bits. Our storage devices are all controlled by a single clock, a periodic signal that determines when new values are to be loaded into the devices. We consider two classes of memory devices:*

1. 时钟寄存器

时钟寄存器（或简称寄存器）存储一个位或字，通过时钟信号控制寄存器的输入值的加载

1. 随机访问存储器

随机存取存储器（或简称存储器）存储多个字，使用地址来选择应读取或写入哪个字

随机存储器的示例包括：

* + 1. 处理器的虚拟存储系统
    2. 寄存器文件：使用寄存器标识符充当地址

✨正如我们所看到的，当谈到硬件和机器语言编程时，“寄存器”一词的含义略有不同

在硬件中：寄存器通过其输入和输出线直接连接到电路的其余部分

在机器语言编程中：寄存器表示CPU中可寻址字的一部分集合，地址是寄存器的ID

因此为了避免歧义，分别将这两种寄存器称为“硬件寄存器”和“程序寄存器”

### 硬件寄存器

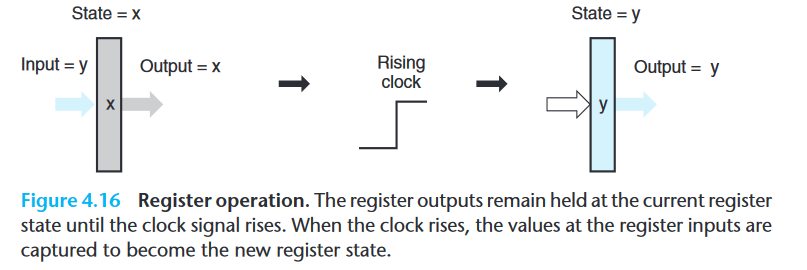


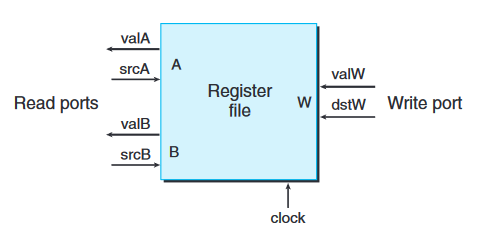
Figure4.16详细介绍了硬件寄存器以及它是怎么进行工作的

* 1. 在大多数情况下寄存器都保持在一个稳定状态（图中的x），产生的输出等于它当前状态
  2. 只要时钟信号是低电平的，寄存器的输出就保持不变
  3. 当时钟变成高电位时，通过组合逻辑传递的输入信号y就加载到寄存器中，成为寄存器的当前稳定状态y直到下一个时钟上升沿，此前的稳定态x成为寄存器的输出
  4. 寄存器可以作为电路不同部分的组合逻辑之间的屏障——每当每个时钟到达上升沿时，值才会从寄存器的输入传送到输出

Y86-64处理器使用硬件寄存器去实现PC、CC、Stat

### 程序寄存器

下面的图展示了一个典型的寄存器文件：



该寄存器堆有两个读端口，名为 A 和 B，以及一个写端口，名为 W。这种多端口随机存取存储器允许同时进行多个读和写操作。

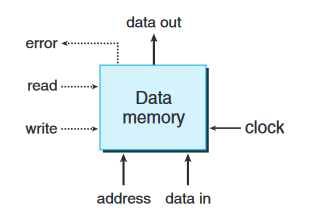
寄存器堆不是组合电路，因为它具有内部存储。然而，在之后的实现中，可以从寄存器文件中读取数据，就好像它是一个组合逻辑块，其地址作为输入，数据作为输出

寄存器的读端口并不需要受时钟信号的控制，时钟信号控制的是寄存器写——在每一次上升沿写端口的数据valW就会写入dstW地址所标识的寄存器

✨因为寄存器堆既可以进行读也可以进行写，那么“当同时尝试写或者读同一个寄存器时会出现什么呢？”答案是——如果寄存器堆中的某个寄存器同时被读和写，那么随着时钟上升，读出来的数据是被新写入的数据

### 数据存储器

为了存储程序数据，处理器内部存在一个随机访问的存储器，其电路结构如下：



读：给定内存单元地址至address引脚，并设置write引脚为低电平，那么在几个延迟后，该存储单元的数据就会加载至data\_out引脚

写：给定将进行更改的内存单元的地址至address，以及新数据至data\_in，并设置write为高电平，那么在时钟上升沿，如果地址有效的话，就会对指定单元进行更新

如果给定的读/写地址超出了可寻址的范围，那么会设置error注释3引脚为高电平，否则为低电平

### 指令存储器

为了存储程序指令，处理器内部存在一个只读的存储器

✨在大多数实际系统中，数据存储器和指令存储器被合并成具有两个端口的单个存储器：一个用于读取指令，另一个用于读取或写入数据

# 4.3 *Sequential Y86-64 Implementations*

Sequential Processor（SEQ）每个时钟周期都做完当前指令所需的全部操作——时钟周期长度为最复杂的指令完成所需要的时间

## 4.3.1 *Organizing Processing into Stages*

### 指令执行阶段序列

一般来说，处理一条指令涉及许多操作。我们按照特定的阶段顺序组织它们，试图使所有指令遵循统一的顺序：

* 1. 取指

✨取指阶段会从内存中读取PC值为地址的内存单元处的指令；从指令说明符中提取的两部分icode、ifun分别表示指令的类型以及做什么样的运算。此外指令中可能会有寄存器操作数位rA、rB，也可能会有8字节的常量valC。取指阶段还需要计算下一条顺序执行指令的地址——当前地址+当前指令长度

* + 1. 取出icode、ifunc
    2. 取出可能有的寄存器指示符rA、rB
    3. 取出可能有的8字节立即数
    4. 计算顺序执行的下一条指令的地址
  1. 译码

✨根据指令的rA、rB字段，从寄存器文件中取得对应的valA、valB；此外还有些指令是读取%rsp的

* 1. 执行

在执行阶段，算术/逻辑单元（ALU）要么执行指令指定的操作（根据ifun的值）注释4，要么计算内存引用的有效地址，要么递增或递减堆栈指针，计算结果为valE，可能会根据结果位设置条件码

对于条件传送指令，该阶段将评估条件代码和移动条件（由 ifun 给出）。类似地，对于跳转指令，它决定是否应该采取分支

* 1. 访存

访存阶段可以将数据写入存储器，也可以从存储器读取数据。将读取的值称为 valM

* 1. 写回：最多可以写两个结果到寄存器文件——目的寄存器+%rsp
  2. 更新PC：PC被设置为下一条指令的地址

### 设计原则

**|** *The processor loops indefinitely, performing these stages. In our simplified implementation, the processor will stop when any exception occurs—that is, when it executes a halt or invalid instruction, or it attempts to read or write an invalid address. In a more complete design, the processor would enter an exception-handling mode and begin executing special code determined by the type of exception*

在进行硬件设计时，使用非常简单且统一的结构非常重要，因为我们希望最大限度地减少硬件总量，并且最终必须将其映射到集成电路芯片的二维表面上——一个降低硬件复杂度的方法是**让不同的指令尽可能多的复用硬件**注释5

这个简单且统一的结构即是上述提出的6阶段的指令执行阶段序列，接下来要做的即是**将Y86-64指令集中的每条不同的指令所需要的计算放入这个通用框架**

### Y86-64指令集通用框架

#### OPq rA,rB

1. 取指：取icode、ifunc、可能有的寄存器指示符、8字节立即数并计算顺序执行的下一条PC地址
2. 译码：取寄存器操作数
3. 执行：根据ifunc计算结果
4. 访存

OPq指令不需要访存

1. 写回
2. 更新PC

#### rrmovq rA,rB

1. 取指：取icode、ifunc、可能有的寄存器指示符、8字节立即数并计算顺序执行的下一条PC地址
2. 译码：取寄存器操作数

只需要取操作数rA

1. 执行：根据ifunc计算结果

这一步是多余的redudant，只是为了实现一个通用的处理方式

1. 访存

OPq指令不需要访存

1. 写回
2. 更新PC

#### irmovq V,rB

1. 取指：取icode、ifunc、可能有的寄存器指示符、8字节立即数并计算顺序执行的下一条PC地址
2. 译码：取寄存器操作数

irmovq指令不需要取寄存器操作数

1. 执行：根据ifunc计算结果
2. 访存

OPq指令不需要访存

1. 写回
2. 更新PC

#### rmmovq rA,D(rB)

1. 取指
2. 译码
3. 执行
4. 访存
5. 写回

无

1. 更新PC

#### mrmovq D(rB),rA

1. 取指
2. 译码
3. 执行
4. 访存
5. 写回
6. 更新PC

#### pushq rA

1. 取指
2. 译码
3. 执行
4. 访存
5. 写回
6. 更新PC

#### popq rA

1. 取指
2. 译码

这里取两次%rsp的值也是多余的，只是为了让popq后续的处理与之前指令的处理逻辑相类似

1. 执行
2. 访存
3. 写回
4. 更新PC

✨pushq和popq的处理逻辑也表明了

pushq %rsp是存%rsp的旧值入栈、popq %rsp是%rsp中的是栈中的值

[4.1.5 Some Y86-64 Instruction Details](https://www.wolai.com/9YFhQHtAXYeR1vApt5a9Ad#x2oaPfttXjgvHATb84Cw3c)

#### jXX Dest

1. 取指
2. 译码

jXX Dest指令不需要读寄存器，因此不需要译码

1. 执行

有条件跳转指令执行阶段需要判断跳转条件是否满足

无条件跳转的Cnd始终为1

1. 访存

JXX Dest指令不需要访存

1. 写回

jXX Dest指令不需要写寄存器

1. 更新PC

#### call Dest

✨call Dest指令相当于 push 下一条地址;j Dest

1. 取指
2. 译码

call指令需要调用栈来保存返回地址，因此需要读取栈顶寄存器的值

1. 执行

对栈顶寄存器的值进行减栈

1. 访存

写下一条指令地址入栈

1. 写回

更新%rsp

1. 更新PC

#### ret

✨ret指令相当于 popq rA ;j rA

1. 取指
2. 译码

ret指令需要popq，因此需要读取两次%rsp的值

1. 执行

对栈顶寄存器操作

1. 访存
2. 写回
3. 更新PC

✨一般都是valB参与ALU运算，valA作赋值或者内存引用

#### cmovq

[rrmovq rA,rB](https://www.wolai.com/9YFhQHtAXYeR1vApt5a9Ad#5cYHtz74VUTaopPuZmRaZC)是cmovq的无条件版本，可以修改rrmovq的通用格式如下，即可实现cmovq的执行格式：

1. 取指：取icode、ifunc、可能有的寄存器指示符、8字节立即数并计算顺序执行的下一条PC地址
2. 译码：取寄存器操作数

只需要取操作数rA

1. 执行：根据ifunc计算结果

这一步是多余的redudant，只是为了实现一个通用的处理方式

1. 访存

OPq指令不需要访存

1. 写回
2. 更新PC

## 4.3.2 SEQ Hardware Structure

4.3.1中已经介绍了Y86-64 SEQ的指令的通用处理结构，接下来的任务是创建硬件设计来实现结构中的6个阶段，并将它们连接起来

### 各阶段对应的硬件单元

各个阶段所涉及到的硬件单元为：

1. 取指：程序计数器PC、指令寄存器insMem和PC增量器
2. 译码：两个读端口、两个写端口的寄存器文件
3. 执行：算术逻辑单元ALU和条件码寄存器CC

算术逻辑单元的作用：

* 1. 对操作数执行算术逻辑运算
  2. 对栈指针%rsp进行增减
  3. 加0实现赋值mov
  4. 计算有效地址

条件码寄存器只有3位

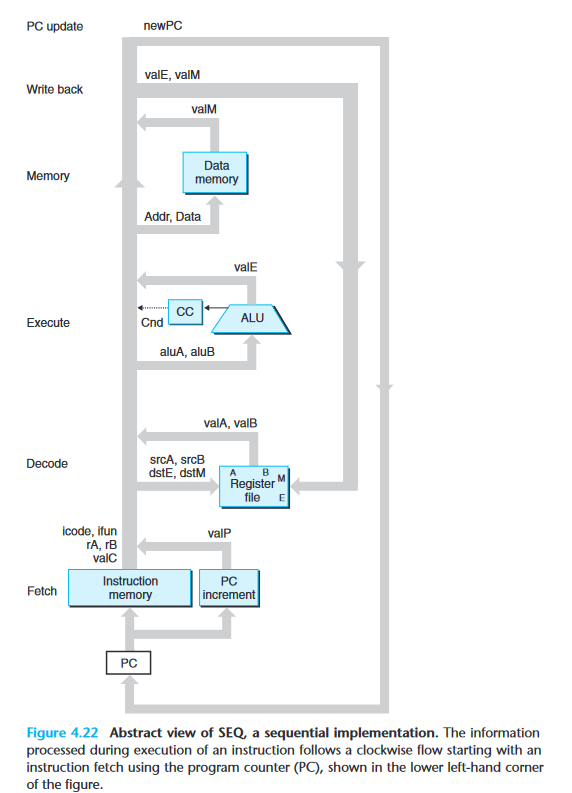
1. 访存：数据存储器dataMem
2. 写回：译码阶段的寄存器文件，E端口写valE、M端口写valM
3. 更新PC：根据valP、valC、valM更新

valP是顺序执行或者条件转移不满足的情况

valC是无条件跳转、以及条件转移满足的情况

valM是返回地址

其对应的SEQ抽象视图如下：



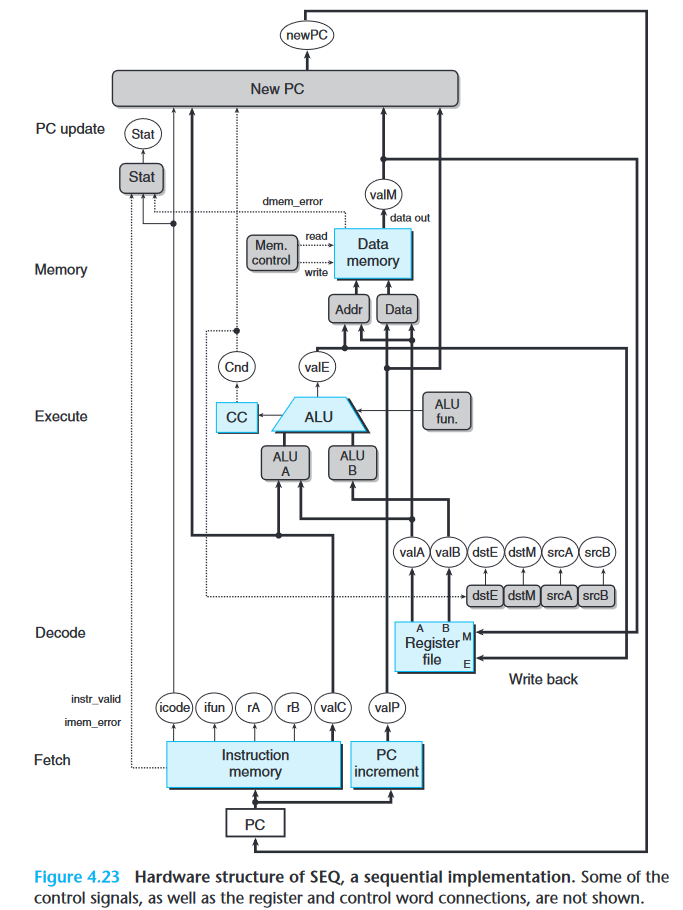
### 硬件图的画图惯例

1. 采用自下而上画处理器和流程的方法
2. 硬件寄存器用白色方框表示——SEQ中唯一的硬件寄存器为PC
3. 硬件单元用浅蓝色方框表示——如存储器、ALU等

这些硬件单元视为“黑盒”，不用关心内部实现细节

1. 控制逻辑块用灰色圆角矩形表示
2. 线路的名称在白色圆圈中说明，只是一个说明标识
3. 宽度为字长的数据连接用中等粗度的线表示  
   宽度为字节或更窄的数据连接用细线表示  
   宽度为位的数据连接用虚线表示

### SEQ详细硬件图



## 4.3.3 SEQ Timing

在4.3.1中对各类型指令的分析中，是将它们看成是用程序符号写的，赋值是从上到下顺序执行的。然后硬件结构的操作运行根本完全不同，一个时钟的变化会引发一个经过组合逻辑的流，来执行整个指令。

SEQ的实现包括组合逻辑和两种存储器设备（如PC、CC这样的时钟寄存器，如寄存器文件、insMem、dataMem这样的随机访问存储器）

组合逻辑以及随机访问存储器的读并不需要时钟信号的控制——只要输入变化了，值就会通过逻辑门网络传播

而PC、CC这样的时钟寄存器，以及随机访问存储器的写操作则需要通过一个时钟信号来控制

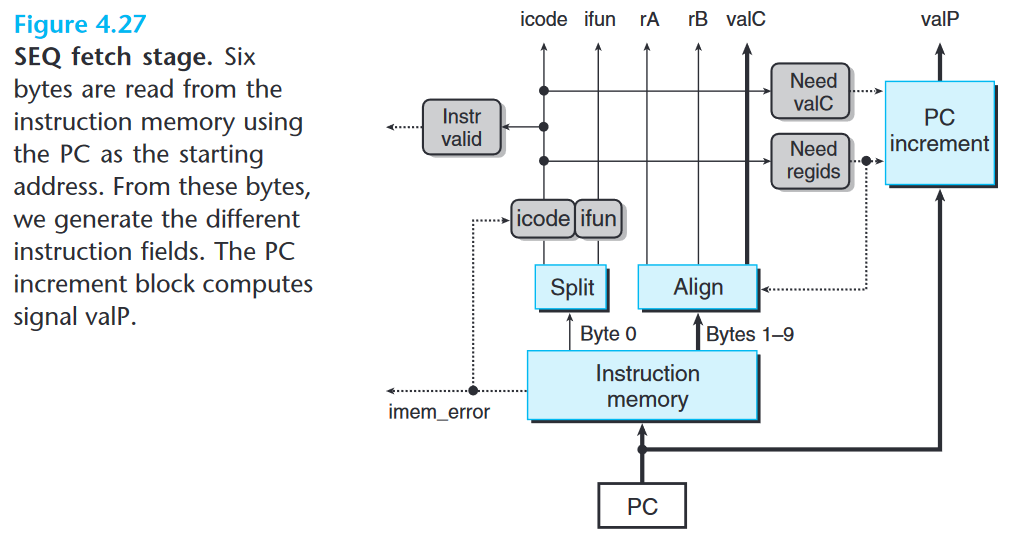
为了确保硬件结构的操作按照4.3.1中各类型指令的顺序执行，需要引入时钟控制机制以同步处理器内时钟寄存器和写内存的操作时序。这样，尽管所有状态的更新在时钟上升开始下一个周期时实际上是同时发生的，但我们仍然能够实现类似于4.3.1中指令顺序执行的效果。

✨之所以引入时钟控制机制使得硬件结构操作和指令逻辑操作效果一致的原因是：Y86-64本质上的从不回读原则，即处理器从来不需要为了完成一条指令的执行而去读由该条指令更新了的状态

**|** 也因为“从不回读”的原则，*some instructions (the integer operations) set the condition codes, and some instructions (the conditional move and jump instructions) read these condition codes, but no instruction must both set and then read the condition codes*.

## 4.3.4 *SEQ Stage Implementations*

### 取指阶段



如上图所示，取指阶段包括指令存储器、分割以及对齐和PC增量器这三种组合逻辑组件、以及时钟寄存器PC寄存器和一些控制逻辑

在取指阶段会根据PC指定的指令存储器地址开始取出10个字节

其中第一个字节Byte 0被Split部件分割成两个4位的控制逻辑——icode和ifun。当指令地址无效时，即出现imem\_error注释6信号时，当前指令的icode和ifun为nop指令的Byte 0；否则为内存中的Byte 0

剩余的9个字节会经过Align部件的处理，将其分割成寄存器指示符和常量字

如果need\_regids有效，则会设置Byte1为寄存器指示符，并将其分割成两个4位的rA、rB；如果need\_regids无效，则会设置rA、rB均为0xF

Align部件会根据need\_regids的有效与否，设置valC为Byte2~Byte9或者Byte1~Byte8

根据icode的值，可以计算出三个1位的控制逻辑instr\_valid、need\_valC、need\_regids

instr\_valid注释7表示当前指令是一个有效的Y86-64指令集指令

need\_valC表示当前指令中包含一个常量字Byte1~Byte9或者Byte2~Byte9

//需要常数字的指令有irmovq、rmmovq、mrmovq  
bool need\_valC = icode in {IIRMOVQ,IRMMOVQ,IMRMOVQ,IJXX,ICALL};

C++

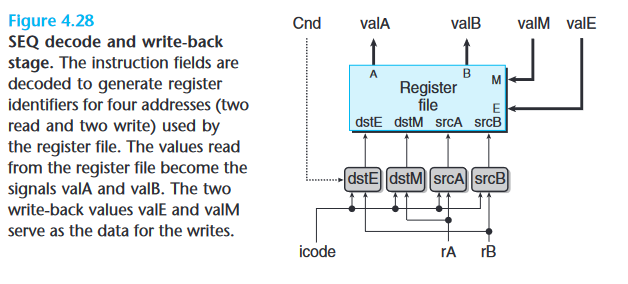
need\_regids表示当前指令中包含寄存器指示符字节Byte1

//需要寄存器的指令有opq、rrmovq、irmovq、rmmovq、mrmovq、cmovXX、pushq、popq  
bool need\_regids = icode in { IRRMOVQ, IOPQ, IPUSHQ, IPOPQ, IIRMOVQ, IRMMOVQ, IMRMOVQ };

C++

PC增量计算部件会根据PC值以及need\_regids、need\_valC信号来产生valP

### 译码、写回阶段



译码阶段和写回阶段均需要使用寄存器文件，因此一起讨论

寄存器文件共有8个引脚，可分为4种端口——两个同时读两个同时写。每种端口都会有数据连接引脚和地址连接引脚。其中地址连接引脚即为寄存器的ID、数据连接引脚为64位数据

两个读端口：A和B。地址引脚是srcA、srcB；输出数据引脚是valA、valB

两个写端口：E和M。E端口是用于ALU结果的写，dstE为地址引脚，valE为输入数据引脚；M端口是用于存储器结果的写，dstM为地址引脚，valM为输入数据引脚

**|** 0xF表示不需要访问寄存器

Figure4.28 [行内引用](https://www.wolai.com/9YFhQHtAXYeR1vApt5a9Ad#e8dKQmGVDv6izsDwWFT3t4)下方的4个控制逻辑块是用来根据rA、rB和icode生成4种端口的地址引脚的

//4位的数据量称为word  
word srcA={  
 icode in {IRRMOVQ,IRMMOVQ,IOPQ,IPUSHQ} : rA;  
 icode in {IRET,IPOPQ} : 0x4;//0x4为%rsp的编号  
 1 : 0xF;   
};//只是需要读rA寄存器的指令  
word srcB={  
 icode in {IRMMOVQ,IMRMOVQ,IOPQ} : rB;  
 icode in {IPUSHQ,IPOPQ,ICALL,IRET} : 0x4;  
 1 : 0xF;  
};//只是需要读rB寄存器的指令  
word dstE={  
 icode in {IRRMOVQ} && Cnd : rB;  
 icode in {IOPQ,IIRMOVQ,IRRMOVQ} : rB;  
 icode in {IPUSHQ,IPOPQ,ICALL,IRET} : 0x4;  
 1 : 0xF;  
};//写ALU结果的指令  
word dstM={  
 icode in {IMRMOVQ,IPOPQ} : rA;  
 1 : 0xF;  
};//写MEM结果的指令

C++

### 执行阶段

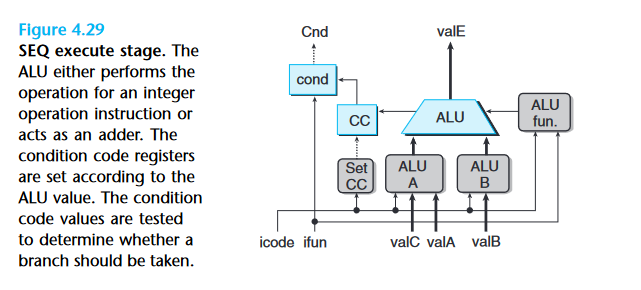


Figure4.19展示了执行阶段的控制逻辑。这一单元会基于alufun信号对输入的aluA和aluB做指定的运算

alu所做的操作都是将aluB作为第一操作数，aluA作第二操作数

word aluA={  
 icode in {IOPQ,IRRMOVQ,ICMOVXX} : valA;  
 icode in {IIRMOVQ,IRMMOVQ,IMRMOVQ} : valC;  
 icode in {IPUSHQ,ICALL} : -8;  
 icode in {IPOPQ,IRET} : 8;  
};  
word aluB={  
 icode in {IOPQ,IRMMOVQ,IMRMOVQ,IPUSHQ,IPOPQ,ICALL,IRET} : valB;  
 icode in {IRRMOVQ,IIRMOVQ,ICMOVXX} : 0;  
};

C++

ALU除了IOPQ对应的运算以外，其他基本执行的都是加法，因此可以编码alufun为

word alufun={  
 icode==IOPQ : ifun;  
 1 : ADD;  
};

C++

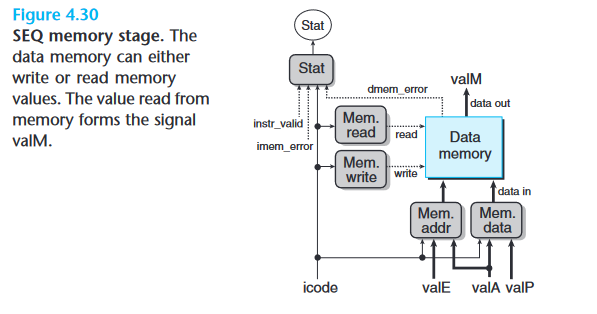
执行阶段还需要根据ALU的操作结果设置Cond，在Y86-64中只有opq指令需要设置Cond，因此可以得到

bool set\_cc= icode==IOPQ;

C++

cond这个硬件单元是使用条件码CC和功能码ifun一起决定是否采用条件分支或者数据转移。cond会根据CC、ifun的输入输出一个一位信号Cond，并根据这一信号设置条件转移的dstE和是否分支的nextPC

### 访存阶段



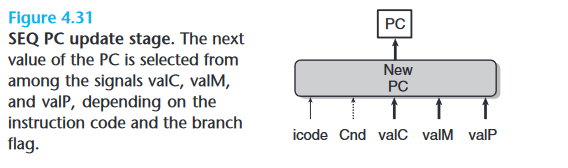
如Figure4.30所示，访存阶段需要完成读取程序数据以及写程序数据的任务。控制块memAddr生成访问的存储器地址，memData生成访问的存储器数据，memRead则生成读信号，memWrite生成写信号

word memAddr={  
 icode in {IRMMOVQ,IMRMOVQ,IPUSHQ,ICALL} : valE;  
 icode in {IPOPQ,IRET} : valA;  
};  
  
word memData={  
 icode in {IRMMOVQ,IPUSHQ} : valA;  
 icode == ICALL : valP;  
};  
  
bool memRead = icode in {IMRMOVQ,IPOPQ,IRET};  
bool memWrite = icode in {IRMMOVQ,IPUSHQ,ICALL}

C++

访存阶段还需要根据icode、imem\_error、instr\_valid和dmem\_error生成stat状态码

### 更新PC阶段



新PC的值可以是valC、valM或者valP。其对应的HCL描述如下：

word new\_pc={  
 icode == IJXX && Cnd : valC;  
 icode == ICALL : valC;  
 icode == IRET : valM;  
 1:valP;  
};

C++

✨SEQ比较简单，但是它十分缓慢。时钟必须运行地足够慢，以便每一种指令的操作都可以在一个时钟周期内执行完毕

SEQ的实现机制并未充分利用板载的硬件资源，因为每一个硬件单元在整个时钟周期内只有一定的比例处于活跃态

# 4.4 *General Principles of Pipelining*

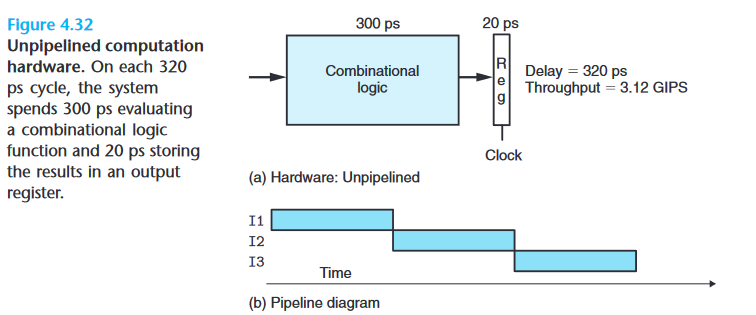
**|** *Some general properties and principles of pipelined systems*

1. *In a pipelined system, the task to be performed is divided into a series of discrete stages.*
2. *Must move at the same rate*
3. *A key feature of pipelining is that it increases the throughput of the systembut it may also slightly increase the latency*

*throughput:the number of customers served per unit time*

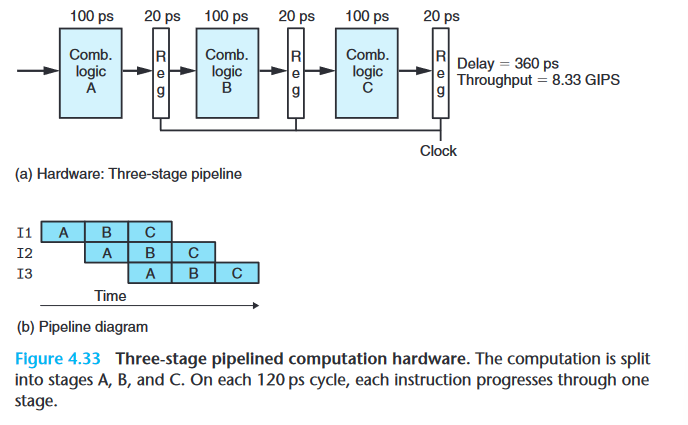
*latency:the time required to service an individual customer*

## 4.4.1 *Computational Pipelines*



在目前的逻辑设计中，衡量电路延迟是使用ps数量级，即

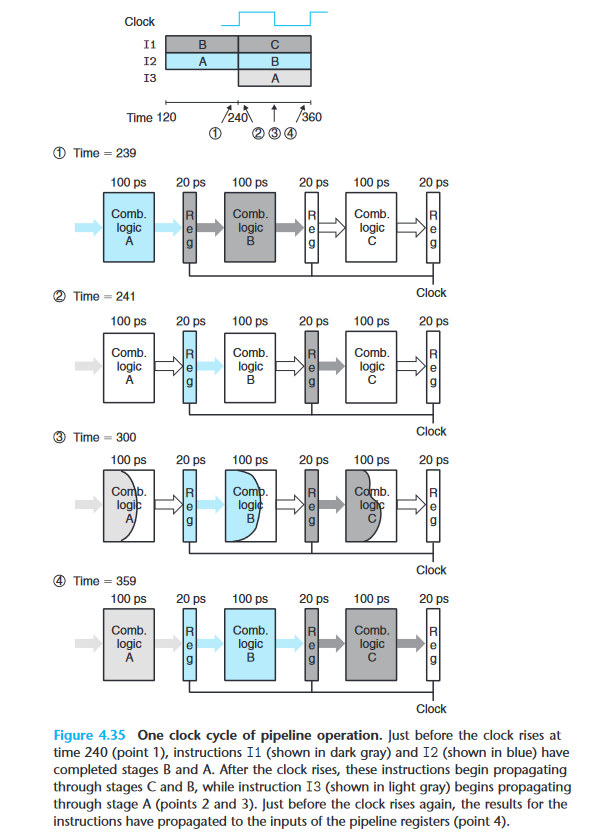
则在Figure4.32的示例中，假设组合逻辑计算块的延迟是300ps、时钟寄存器的加载延迟是20ps，则整个系统的latency是320ps，throughput是3.12GIPS



假设如Figure4.33所示，将Figure4.32中的组合逻辑分为三个阶段A、B和C，每个阶段耗时100ps。并在阶段之间插入时钟寄存器以实现流水。这样则当该系统处于稳定时，每个时钟周期有一条指令离开系统并有一条指令进入系统。可以计算出时钟周期是120ps，latency是360ps，throughput是8.33GIPS

✨吞吐量的增加使得硬件的开销和延迟增加注释8

## 4.4.2 *A Detailed Look at Pipeline Operation*

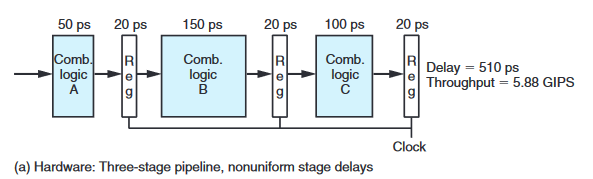


在240ps的时钟上升之前，组合逻辑的输出并没有打入到寄存器中。因此组合逻辑B和Reg1是在计算I1的，组合逻辑A是I2，组合逻辑C是不活跃的  
在240ps的时钟上升之后，之前的组合逻辑输出已经打入到了寄存器中但是还没有过20ps，仍没有传输到紧接着寄存器的组合逻辑块，因此组合逻辑A、B、C均空，Reg2是I1，Reg1是I2  
在300ps时，上一阶段的组合逻辑输出已经传播出寄存器，因此Reg2和组合逻辑C是I1；Reg1和组合逻辑B是I2；组合逻辑A是I3  
在359ps时，组合逻辑的计算已完毕，等待传播到寄存器

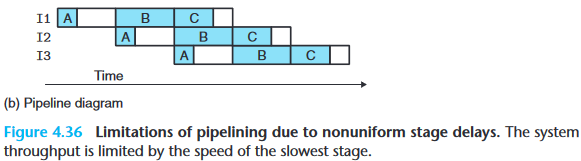
## 4.4.3 *Limitations of Pipelining*

### *Nonuniform Partitioning*

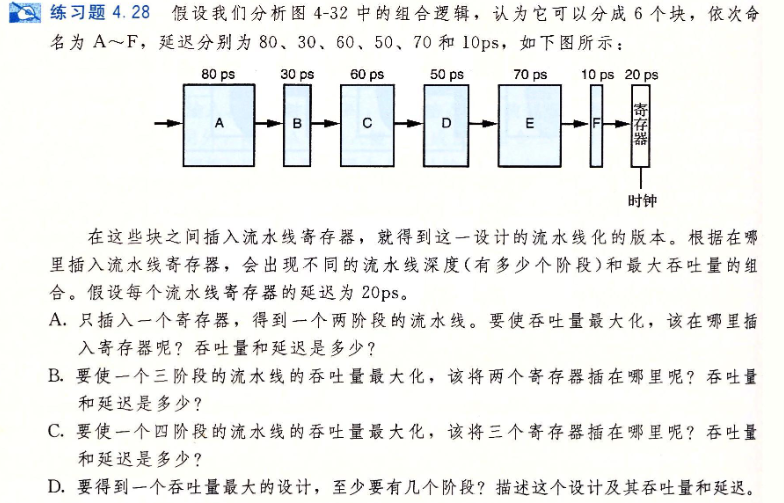
在实际应用中，可能会出现如下图这样存在着划分的组合逻辑段所需要的时间是不等长的系统，该系统的latency是510ps，throughput是5.88GIPS——因为**时钟周期会设置为瓶颈段所需要的时间**



其流水线时空图如下：



**|** 例题



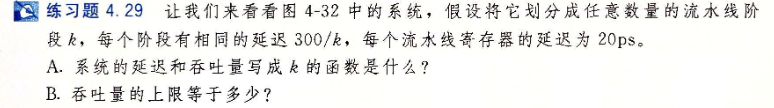
A. ABC 寄存器 DEF 时钟周期是190ps  
B. AB CD EF 时钟周期是130ps  
C.A BC D E 时钟周期是110ps  
D.A B C D EF 时钟周期是100ps

### *Diminishing Returns of Deep Pipelining*

因为在各个组合逻辑块之间需要插入寄存器来控制，所以如果划分的段太多，那么就会增加太多的额外硬件成本以及寄存器延迟，使得latency中的寄存器延迟的比例升高，且对吞吐量的增加的比例也下降

**|** *Modern processors employ very deep pipelines (15 or more stages) in an attempt to maximize the processor clock rate. The processor architects divide the instruction execution into a large number of very simple steps so that each stage can have a very small delay.*

**|** 例题



A.

B.k趋于无穷时，throughput上限是50GIPS

## 4.4.4 *Pipelining a System with Feedback*

这里的反馈回路即是带有冒险的流水线，包括数据冒险、控制冒险和结构冒险

# 4.5

[注释1] 使用汇编语言或者机器语言编写程序的程序员

[注释2] 允许非独占选择表达式使 HCL 代码更具可读性。实际的硬件多路复用器必须具有互斥的信号来控制哪个输入字应传递到输出，如图 4.13 中的信号 s 和 !s。要将 HCL 案例表达式转换为硬件，逻辑综合程序需要分析选择表达式集，并通过确保仅选择第一个匹配案例来解决任何可能的冲突

[注释3] 组合逻辑生成error

[注释4] mov类指令的一个操作数是0

[注释5] 比如ALU的复用

[注释6] 当PC的值超出内存寻址范围

[注释7] instr和imem\_error可以用来生成存储器中的Stat状态字

[注释8] 因为在分割的组合逻辑块之间插入了时钟寄存器