CSAPP-处理器体系结构

前言

我们先简单了解一下什么是指令集体系结构 (ISA)

ISA在编译器编写者（CPU软件）和处理器设计人员（CPU硬件）之间提供了一个抽象层

- **处理器设计者**：依据ISA来设计处理器
- **处理器使用者** :依据ISA就知道CPU选用的指令集，就知道自己可以使用哪些指令以及遵循哪些规范

简单来说, ISA 规定了 CPU 如何去执行一段二进制编码, 同时也告诉哪些生成这段二进制编码的程序(如编译器)如何去生成可以被 CPU 正确解读的二进制编码.

这就是为什么同一个可执行程序在不能在不同的机器中执行的原因之一

不同的处理器家族，例如Intel IA32、IBM/Freescale PowerPC和ARM，都有不同的ISA

Y86-64 指令集体系结构

为了更好的学习处理器的体系结构, CSAPP 原书以 X86-64 为原型进行简化, 最后命名为 Y86-64 来方便讲述

程序员可见的状态

注意, 这里的 "程序员" 既是使用汇编代码来写程序的人, 同时也可以是产生机器代码的编译器

Y86-64 有以下的可见单元 :

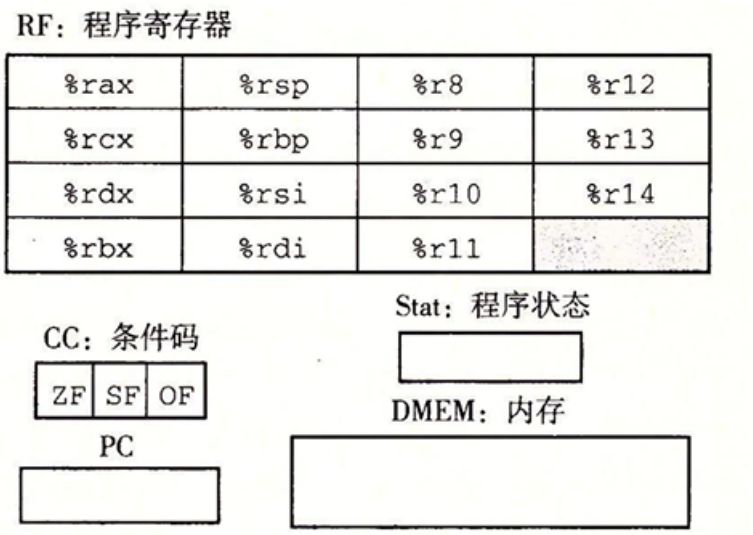


图 4-1 Y86-64 程序员可见状态。同 x86-64 一样，Y86-64 的程序可以访问和修改程序寄存器、条件码、程序计数器(PC)和内存。状态码指明程序是否运行正常，或者发生了某个特殊事件

- 15个程序寄存器。每个程序寄存器存储一个 64 位的字。寄存器 %rsp 被入栈、出栈、调用和返回指令作为栈指针。
- 3个1位的条件码，它们保存着最近的算术或逻辑指令所造成影响的有关信息。
- 程序计数器（PC）存放当前正在执行指令的地址。
- 内存从概念上来说就是一个很大的字节数组，保存着程序和数据。
- 状态码Stat，表明程序执行的总体状态。它会指示是正常运行，还是出现了某种异常

Y86-64 指令

- `movq` 指令分成了4个不同的指令：`irmovq` 、`rrmovq` 、`mrmov` 、`rmmovq` ，分别显式的指明源和目的的格式。

  > 源可以是立即数(i)、寄存器(r)或内存(m)
  >
  > 指令名字的第一个字母就表明了源的类型。目的可以是寄存器(r)或内存(m)，指令名字的第二个字母指明了目的的类型。

- 有4个整数操作指令，见下图 `OPq` 指令

  > 分别是 `addq` 、`subq` 、`andq` 和 `xorq` ，只对寄存器数据进行操作
  >
  > 这些指令会设置 3 个条件码 `ZF` 、`SF` 和 `OF` （零、符号和溢出）。

- 7个跳转指令是 `jmp` 、`jle` 、`jl` 、`je` 、`jne` 、`jge` 和 `jg` 。
- 有6个条件传送指令：`cmovle`、`cmovl`、`cmove`、`cmovne`、`cmovge` 和 `cmovg`。
- `call` 将 `call` 的下条指令地址值压到栈顶，然后将 `PC` 值设置为 `call` 后面跟的目的地址
- `ret` 将 `PC` 的设置为当前栈顶存放的值。
- `pushq` 和 `popq` 指令实现了入栈和出栈。
- `halt` 指令停止指令的执行，执行 `halt` 指令会导致处理器停止，并将状态码设置为 `HLT`

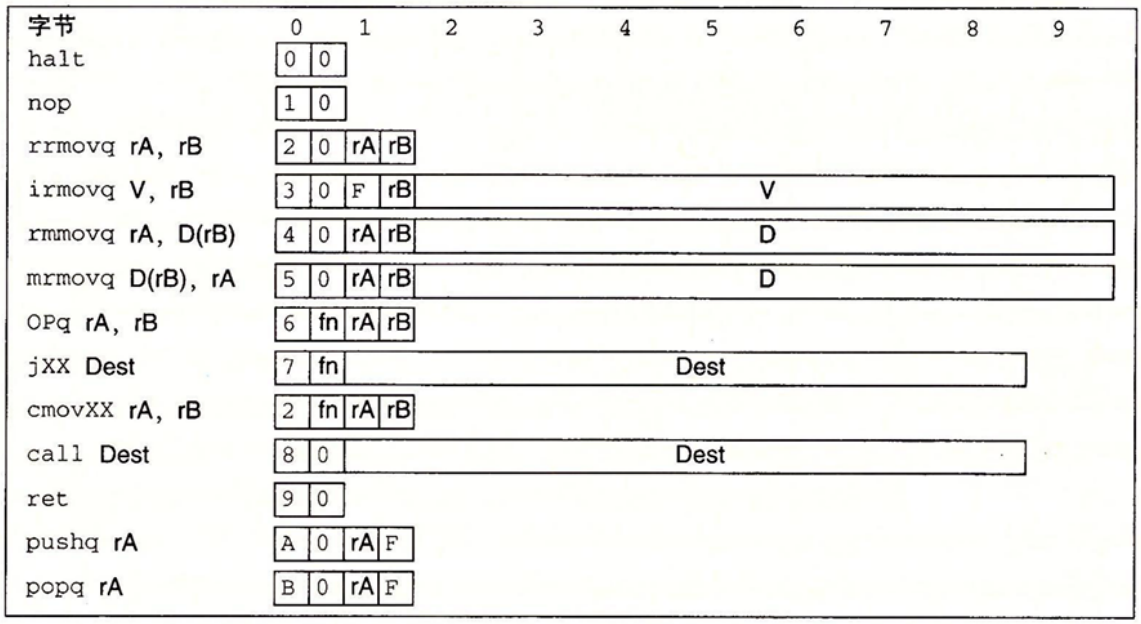


图 4-2 Y86-64 指令集。指令编码长度从 1 个字节到 10 个字节不等。一条指令含有一个单字节的指令指示符，可能含有一个单字节的寄存器指示符，还可能含有一个 8 字节的常数字。字段 fn 指明是某个整数操作(OPq)、数据传送条件(cmovXX)或是分支条件(jXX)。所有的数值都用十六进制表示

指令编码

对于一个完整的二进制指令编码, 前 8 个 bit 就应该能分辨出对应的汇编代码的指令是哪一个

我们将这前 8 个bit 分成两个部分, 分别是前 4 个 bit 和 后 4 个 bit

- 前 4 个 bit 叫做代码 (code) 部分, 代码值为 $0\mathrm{x}0 \sim 0\mathrm{xB}$
- 后 4 个 bit 叫做功能 (function) 部分

具体的使用如下图, 这样我们就能够在前 8 个 bit 就准确区分是哪个指令了

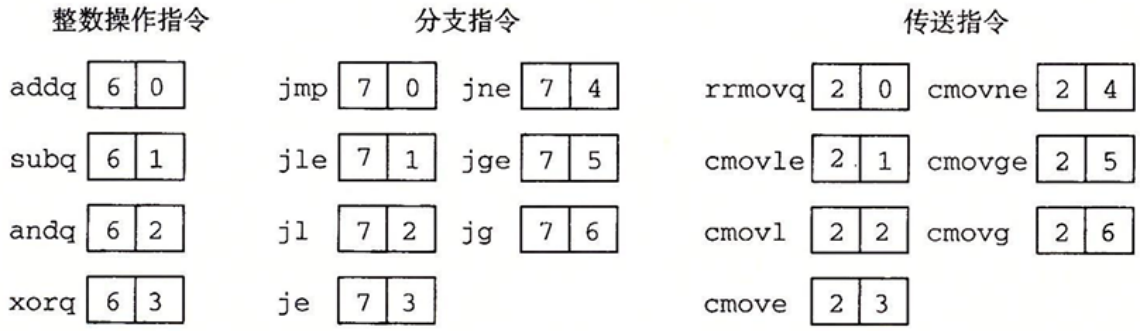


图 4-3 Y86-64 指令集的功能码。这些代码指明是某个整数操作、分支条件还是数据传送条件。这些指令是图 4-2 中所示的 OPq、jXX 和 cmovXX

除此之外, 我们还需要对寄存器进行编号, 这样我们就可以识别我们要执行操作的寄存器具体是哪一个

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

图 4-4 Y86-64 程序寄存器标识符。15 个程序寄存器中每个都有一个相对应的标识符(ID)，范围为 0～0xE。如果指令中某个寄存器字段的 ID 值为 0xF，就表明此处没有寄存器操作数

值得注意的是 $0\mathrm{xF}$: 一个二进制指令编码的第 $9 \sim 16$ 位是用来表示这条指令操作的两个寄存器的, 但有些指令是只操作一个寄存器的, 这个时候就需要将高 4 位或者低 4 位设置成 $0\mathrm{xF}$ 代表只有一个寄存器操作

Y86-64 异常

对于 `Y86-64`，当遇到这些异常的时候，我们就简单地让处理器停止执行指令。在更完整的设计中，处理器通常会调用一个**异常处理程序**（exception handler），这个过程被指定用来处理遇到的某种类型的异常。

Y86-64 的异常表示是使用 `stat` 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

图 4-5 Y86-64 状态码。在我们的设计中，任何 AOK 以外的代码都会使处理器停止

逻辑设计和硬件控制语言 HCL

在硬件设计中，用电子电路来计算对位进行运算的函数，以及在各种存储器单元中存储位。大多数现代电路技术都是用信号线上的高电压或低电压来表示不同的位值。

要实现一个数字系统需要三个主要的组成部分：计算对位进行操作的函数的组合逻辑、存储位的存储器单元，以及控制存储器单元更新的时钟信号。

- **HCL**（Hardware Control Language，硬件控制语言），用来描述不同处理器设计的控制逻辑。
- **硬件描述语言**（Hardware Description Language，HDL）是一种文本表示，用来描述硬件结构而不是程序行为的。最常用的语言是 `Verilog`，它的语法类似于 `C`。20 世纪 80 年代中期，研究者开发出了**逻辑合成**（logic synthesis）程序，它可以根据 HDL 的描述生成有效的电路设计，现在已经成为产生数字电路的主要技术。

HCL 语言只表达硬件设计的控制部分，只有有限的操作集合，也没有模块化。控制逻辑是设计微处理器中最难的部分。我们已经开发出了将 HCL 直接翻译成 Verilog 的工具，将这个代码与基本硬件单元的 Verilog 代码结合起来，就能产生 HDL 描述，根据这个 HDL 描述就可以合成实际能够工作的微处理器

逻辑门

逻辑门是数字电路的基本计算单元。它们产生的输出，等于它们输入位值的某个布尔函数。

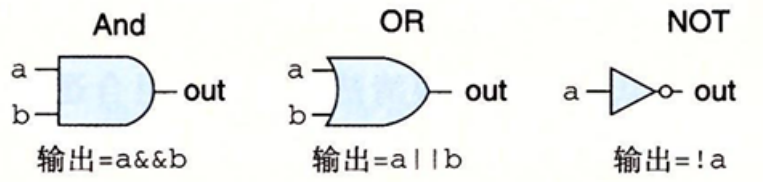


图 4-9 逻辑门类型。每个门产生的输出等于它输入的某个布尔函数

组合电路和 HCL 布尔表达式

将很多的逻辑门组合成一个网，就能构建**计算块**（computational block），称为**组合电路**（combinational circuits）

组合电路有如下的几点限制：

- 每个逻辑门的输入必须连接到下述选项之一:

  1. 一个系统输入(称为主输入)
  2. 某个存储器单元的输出
  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。否则它们可能会使线上的信号矛盾，可能会导致一个不合法的电压或电路故障。
- 这个网必须是无环的。

  > 在网中不能有路径经过一系列的门而形成一个回路, 这样的回路会导致该网络计算的函数有歧义。

下面我们展示了两个非常有意思的组合电路 : 用 HCL 来写这个网的函数为 :

```
图4-10: bool eq = (a && b) || (!a && !b)
图4-11: bool out = (s && a) || (!s && b)
```

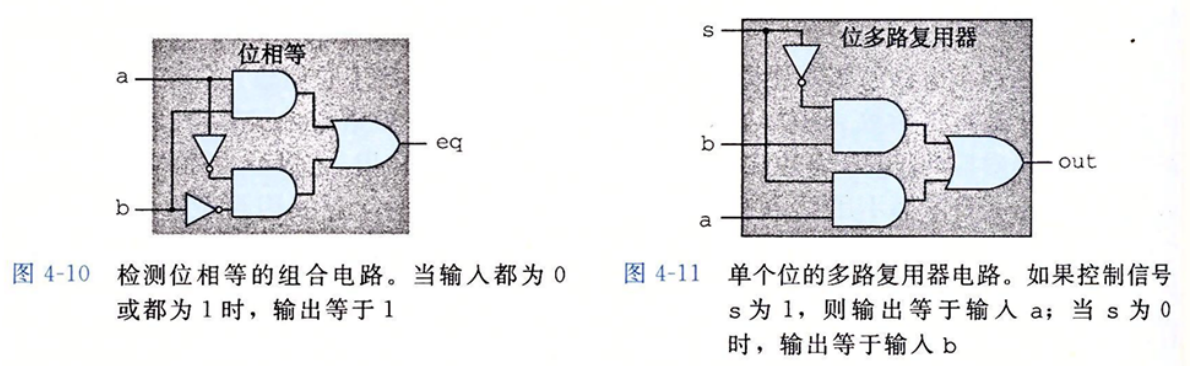


图 4-10 检测位相等的组合电路。当输入都为 0 或都为 1 时，输出等于 1

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

图4-10 是简单组合电路, 而图4-11 是多路复用器（通常称为`MUX`）

多路复用器指 : 根据控制信号的值, 从一组不同的数据信号中选出一个

字级的组合电路和 HCL 整数表达式

通常，我们设计能对数据**字**（word）进行操作的电路。执行字级计算的组合电路根据输入字的各个位，用逻辑门来计算输出字的各个位。在 HCL 中，我们将所有字级的信号都声明为 `int` ，不指定字的大小。这样做是为了简单。在全功能的硬件描述语言中，每个字都可以声明为有特定的位数。

在画字级电路的时候，我们用中等粗度的线来表示携带字的每个位的线路，而用虚线来表示布尔信号结果。

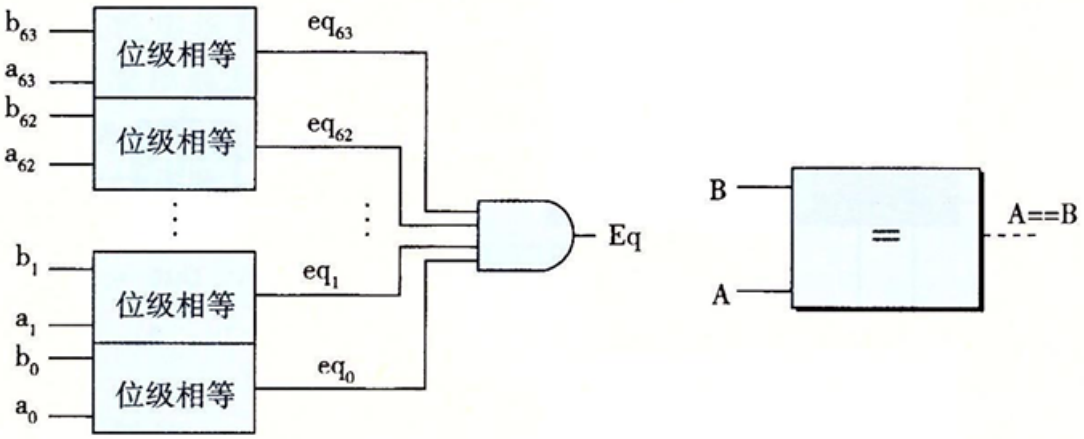


图 4-12　字级相等测试电路。当字 A 的每一位与字 B 中相应的位均相等时，输出等于 1。字级相等是 HCL 中的一个操作

HCL 表达式为：

```
bool Eq = (A == B)
//这里的A, B 都是 int 型
```

字级的多路复用器电路。处理器中会用到很多种多路复用器，使得我们能根据某些控制条件，从许多源中选出一个字。在 HCL 中，多路复用函数是用**情况表达式**（expression）来描述的。选择表达式 `1` 指定默认情况。

情况表达式的通用格式如下：

```
[
    select1 : expr1;
    select2 : expr2;
    ....
    select_k : expr_k;
]
```

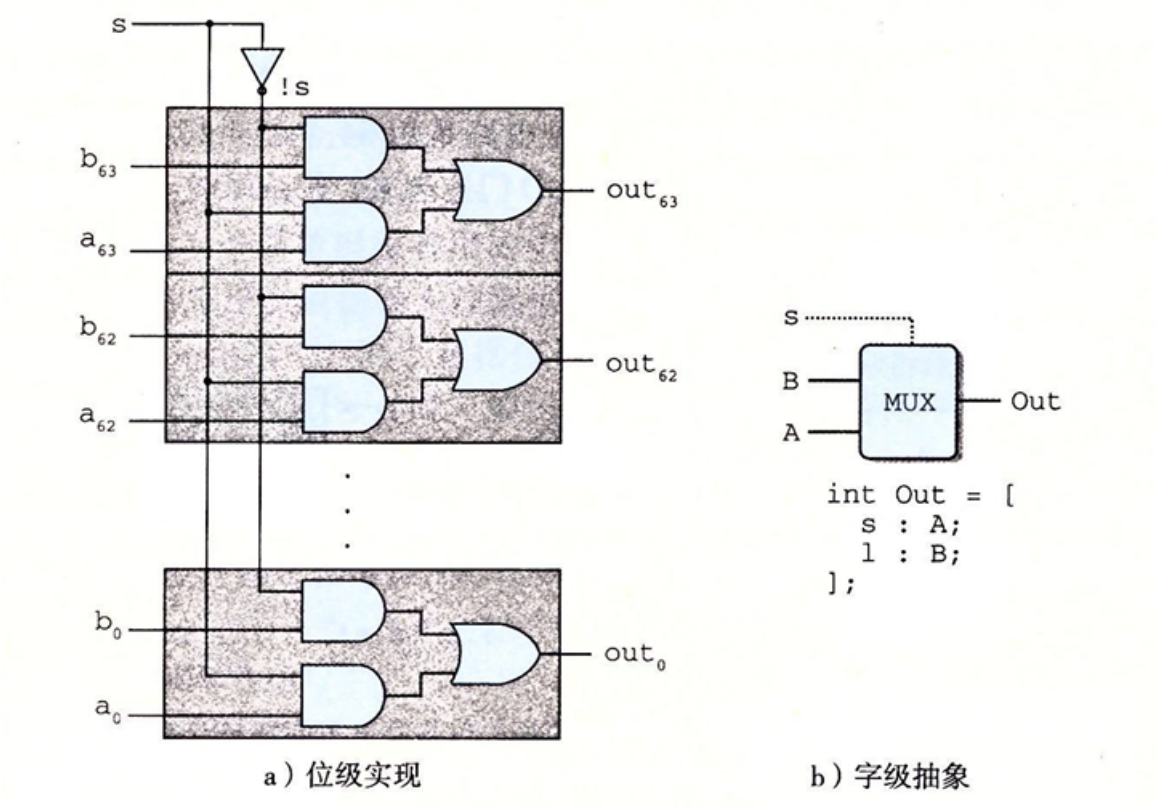


图 4-13　字级多路复用器电路。当控制信号 s 为 1 时，输出会等于输入字 A，否则等于 B。HCL 中用情况(case)表达式来描述多路复用器

对于我们本节的重点是**算术逻辑单元(ALU)**

算术/逻辑单元（ALU）是一种很重要的组合电路。这个电路有三个输入：标号为 A 和 B 的两个数据输入，以及一个控制输入。根据控制输入的设置，电路会对数据输入执行不同的算术或逻辑操作，而控制值和指令集支持的四种不同的整数操作的功能码相对应。

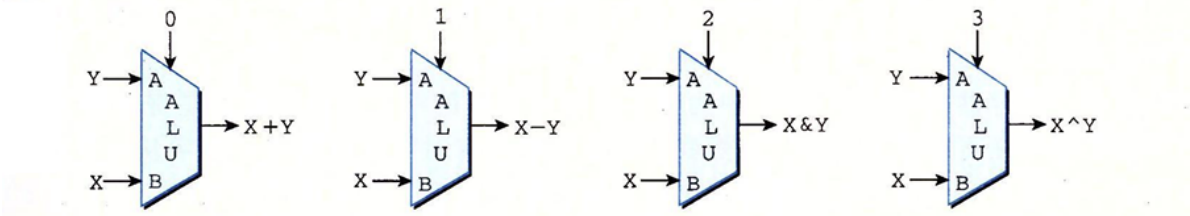


图 4-15　算术/逻辑单元(ALU)。根据函数输入的设置，该电路会执行四种算术和逻辑运算中的一种

集合关系

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做比较，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。举个例子，假设想从一个 2 位信号 code 中选择高位和低位来产生四路复用器的控制信号 s1 和 s0，对应的HCL表达式：

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

将其写成集合关系表述就是：

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

存储器和时钟

组合电路从本质上讲，不存储任何信息。相反，它们只是简单地响应输入信号，产生等于输入的某个函数的输出。为了产生**时序电路**（sequential circuit），也就是有状态并且在这个状态上进行计算的系统，我们必须引入按位存储信息的设备。存储设备都是由同一个时钟控制的，时钟是一个周期性信号，决定什么时候要把新值加载到设备中。考虑两类存储器设备：

- **时钟寄存器**（简称寄存器）存储单个位或字。时钟信号控制寄存器加载输入值。
- **随机访问存储器**（简称内存）存储多个字，用地址来选择该读或该写哪个字。

在硬件和机器级编程中，“寄存器”这个词是两个有细微差别的事情：

- 在硬件中，寄存器直接将它的输入和输出线连接到电路的其他部分。
- 在机器级编程中，寄存器代表的是 CPU 中为数不多的可寻址的字，这里的地址是寄存器 ID 。

下图更详细地说明了一个硬件寄存器以及它是如何工作的。

大多数时候，寄存器都保持在稳定状态(用 x 表示)，产生的输出等于它的当前状态。信号沿着寄存器前面的组合逻辑传播，这时，产生了一个新的寄存器输入(用 y 表示)，但只要时钟是低电位的，寄存器的输出就仍然保持不变。当时钟变成高电位的时候，输入信号就加载到寄存器中，成为下一个状态 y ，直到下一个时钟上升沿，这个状态就一直是寄存器的新输出

这就说明只有时钟上升的时候, CC, PC, Stat 和寄存器的值才会发生变化

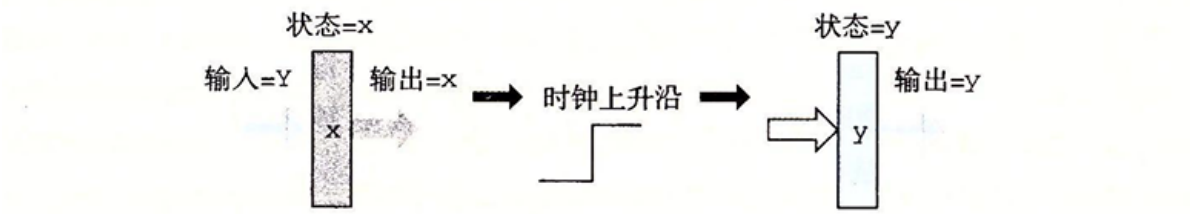
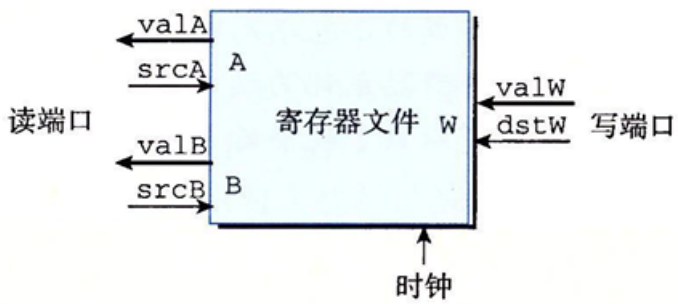


图 4-16　寄存器操作。寄存器输出会一直保持在当前寄存器状态上，直到时钟信号上升。当时钟上升时，寄存器输入上的值会成为新的寄存器状态

接下来我们介绍一下 **寄存器文件**

一般情况下, 我们将我们拥有的所有寄存器集合看成一个文件, 读写都看成对文件的操作

一个典型的寄存器文件，寄存器文件有两个**读端口**（A 和 B），还有一个**写端口**（W）。这样一个多端口随机访问存储器允许同时进行多个读和写操作。每个端口都有一个地址输入，表明该选择哪个程序寄存器。向寄存器文件写入字是由时钟信号控制的，控制方式类似于将值加载到时钟寄存器：



Y86-64 的顺序实现

将处理组织成阶段

对于所有的指令, 我们都可以抽象成下面的几个阶段 :

- **取指**（fetch）：取指阶段从内存读取指令字节，地址为程序计数器 `PC` 的值。从指令中抽取出指令指示符字节的两个四位部分，称为 `icode`（指令代码）和 `ifun`（指令功能）。它可能取出一个寄存器指示符字节，指明一个或两个寄存器操作数指示符 `rA` 和 `rB`。它还可能取出一个四字节常数字 `valC`。它按顺序方式计算当前指令的下一条指令的地址 `valP`。也就是说，`valP` 等于的 `PC` 值加上已取出指令的长度。

  > 在取指阶段, 会直接取出 10 个字节, 因为取指的时候是不知道二进制指令具体的长度的, 但是一条指令最多包含 10 个字节, 所以一次性取 10 个字节能确保一定能完整的取出一条指令
  >
  > 【注】Y86-64是一个非常简单的指令集，所以 `icode + ifun` 可以用一个字节表示，如果是x86-64，肯定不止一个字节。很多RISC指令集是 `32` 位定长的，所以取指方式同这里的也会不一样，不过原理相同

- **译码**（decode）：译码阶段从寄存器文件读入最多两个操作数，得到值 `valA` 和/或 `valB`。通常，它读入指令 `rA` 和 `rB` 字段指明的寄存器，不过有些指令是读寄存器 `%rsp` 的。

  > 对于 `pushq` 指令, 虽然 $\mathrm{rb} = 0\mathrm{xF}$, 但是写入栈需要知道栈指针的值, 所以还需要读取 `%rsp`

- **执行（execute）**：在执行阶段，算术/逻辑单元（ALU）要么执行指令指明的操作（根据 ifun 的值），计算内存引用的有效地址，要么增加或减少栈指针。得到的值我们称为 valE 。在此，也可能设置条件码。
  - 对一条条件传送指令来说，这个阶段会检验条件码和传送条件（由 ifun 给出），如果条件成立，则更新目标寄存器。
  - 对一条跳转指令来说，这个阶段会决定是不是应该选择分支。
- **访存**（memory）：访存阶段可以将数据写入内存，或者从内存读出数据。读出的值为 `valM`。
- **写回**（write back）：写回阶段最多可以写两个结果到寄存器文件。
- **更新 PC**（PC update）：将 `PC` 设置成下一条指令的地址。

处理器无限循环，执行这些阶段。幸好每条指令的整个流程都比较相似。因为我们想使硬件数量尽可能少，在设计硬件时，一个非常简单而一致的结构是非常重要的。降低复杂度的一种方法是让不同的指令共享尽量多的硬件

我们用下面这段代码来来实际模拟一下上面的过程 :

```
1   0x000: 30f20900000000000000 |     irmovq $9,  %rdx
2   0x00a: 30f31500000000000000 |     irmovq $21, %rbx
3   0x014: 6123                 |     subq %rdx, %rbx          # subtract
4   0x016: 30f48000000000000000 |     irmovq $128,%rsp         # Problem 4.13
5   0x020: 4043640000000000000  |     rmmovq %rsp, 100(%rbx)   # store
6   0x02a: a02f                 |     pushq %rdx               # push
7   0x02c: b00f                 |     popq  %rax               # Problem 4.14
8   0x02e: 734000000000000000   |     je done                  # Not taken
9   0x037: 804100000000000000   |     call proc                # Problem 4.18
10  0x040:                      | done:
11  0x040: 00                   |     halt
12  0x041:                      | proc:
13  0x041: 90                   |     ret                      # Return
14                              |
```

图 4-17　Y86-64 指令序列示例。我们会跟踪这些指令通过各个阶段的处理

跟踪 `subq` 指令的执行

我们跟踪第 3 行中的 subq 指令。可以看到前面两条指令分别将寄存器号 `%rdx` 和号 `%rbx` 初始化成 9 和 21。我们还能看到指令位于地址 0x02a，由两个字节组成，值分别为 0x61 和 0x23。这条指令处理的各个阶段如下表所示，左边列出了处理一个 `OPq` 指令的通用的规则，而右边列出的是对这条具体指令的计算。

阶段	OPq rA, rB	subq %rdx, %rbx
取指	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2	icode:ifun ← M_1[0x014]=6:1 rA:rB ← M_1[0x015]=2:3 valP ← 0x014+2=0x016
译码	valA ← R[rA] valB ← R[rB]	valA ← R[%rdx]=9 valB ← R[%rbx]=21
执行	valE ← valB OP valA Set CC	valE ← 21−9=12 ZF ← 0, SF ← 0, OF ← 0
访存		
写回	R[rB]← valE	R[%rbx]← valE=12
更新 PC	PC ← valP	PC ← valP=0x016

这个跟踪表明我们达到了理想的效果，寄存器号 `%rbx` 设成了 12，三个条件码都设成了 0，而 `PC` 加了 2。

跟踪 `rmmovq` 指令的执行

我们跟踪第 5 行 `rmmovq` 指令的处理情况。

可以看到，前面的指令已将寄存器 `%rsp` 初始化成了 128，而 `%rbx` 仍然是 `subq` 指令(第 3 行)算出来的结果 12。我们还可以看到，指令位于地址 0x020，有 10 个字节。前两个的值为 0x40 和 0x43，后 8 个是数字 0x0000000000000064 (十进制数 100)按字节反过来得到的数。各个阶段的处理如下：

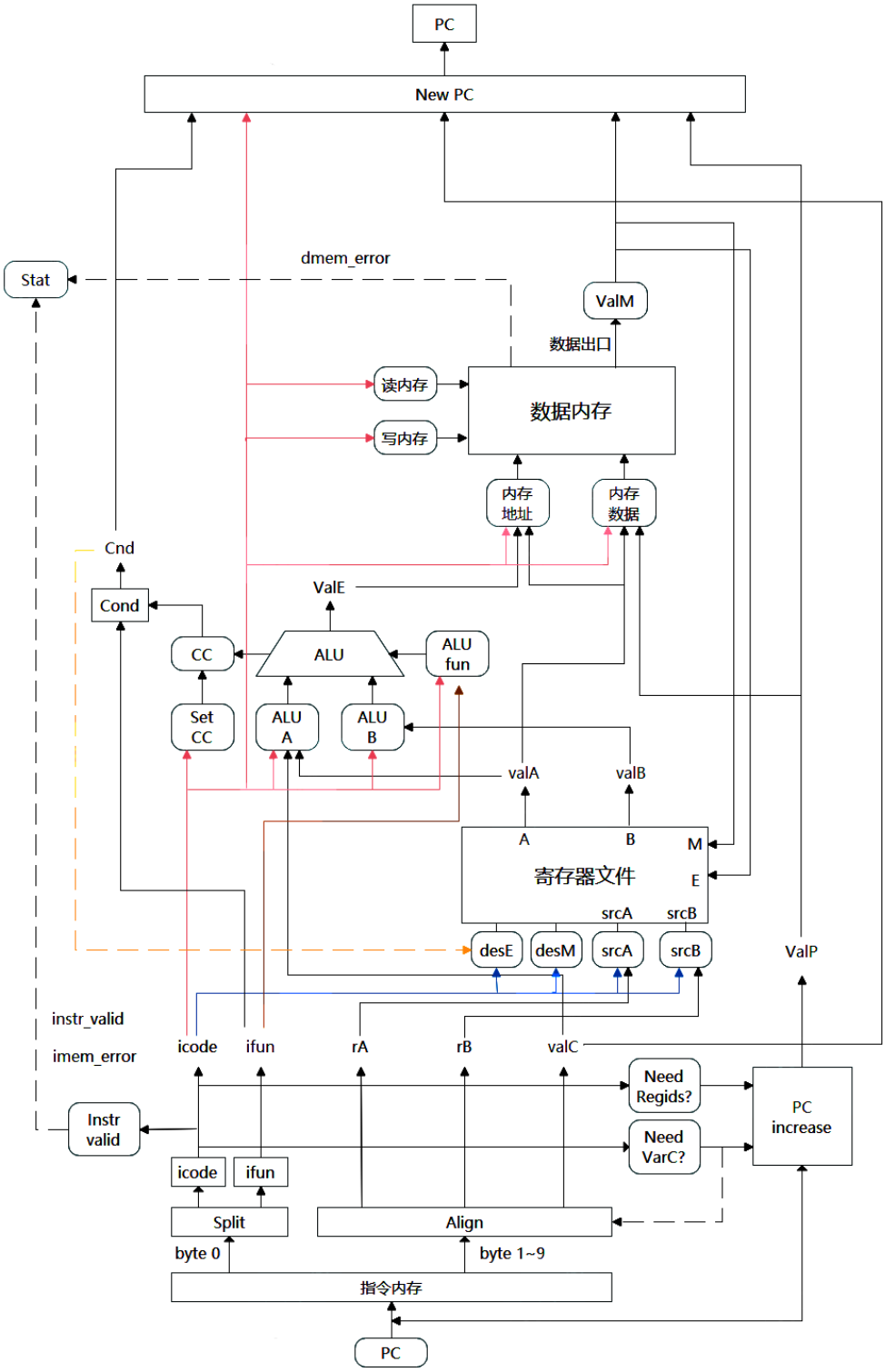
阶段	通用	具体
	rmmovq rA, D(rB)	rmmovq %rsp, 100(%rbx)
取指	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← M_8[PC+2] valP ← PC+10	icode:ifun ← M_1[0x020]=4:0 rA:rB ← M_1[0x021]=4:3 valC ← M_8[0x022]=100 valP ← 0x020+10=0x02a
译码	valA ← R[rA] valB ← R[rB]	valA ← R[%rsp]=128 valB ← R[%rbx]=12
执行	valE ← valB+valC	valE ← 12+100=112
访存	M_8[valE]← valA	M_8[112]← 128
写回		
更新 PC	PC ← valP	PC ← 0x02a

> 跟踪记录表明这条指令的效果就是将 128 写入内存地址 112，并将 `PC` 加 10

..... [其他指令自行看书或者视频]

SEQ 硬件结构和时序

我们将整个 SEQ 硬件结构绘制在下图, 等本节全部学完可以对照进行复习



一个时钟变化会引发一个经过组合逻辑的流，来执行整个指令。组合逻辑电路与时序电路不同，不存储任何信息，只是简单地响应输入信号，产生符合组合逻辑的输出，如果输入信号再次更新，则输出也会跟着变化，**并不会**将先前的输出存下来，时序电路则有专门的存储设备将输出保存下来。

SEQ 的时序实现包括**组合逻辑**和**两种存储设备**：时钟寄存器（`PC` 和 条件码寄存器），随机访问存储器（寄存器文件，指令内存 和 数据内存）；组合逻辑不需要任何时序或控制——只要输入变化了，值就通过逻辑门网络传播。可以将读随机访问存储器看成是组合逻辑一样的操作，即当输入某个合法地址值，输出则是地址中存放的值。

有以下四个硬件单元的时序要进行明确的控制——`PC`、条件码寄存器、数据内存 和 寄存器文件。这些单元通过一个时钟信号来控制，它触发将新值装载到寄存器以及将值写到随机访问存储器。每个时钟周期，`PC` 都会装载新的指令地址；只有在执行整数运算指令时，才会装载条件码寄存器；只有在执行 `rmmovq` 、 `pushq` 或 `call` 指令时，才会写数据内存；寄存器文件的两个写端口允许每个时钟周期更新两个程序寄存器，不过我们可以用特殊的寄存器 ID `0xF`作为端口地址，来表明此端口不应该执行写操作

要控制处理器中活动的时序，只需要寄存器和内存的时钟控制，且所有的状态的更新都只在时钟上升开始下一个周期时同时发生。保持这样的等价性要遵循一个组织计算原则：

- **从不回读**：`处理器从来不需要为了完成一条指令的执行而去读由该指令更新了的状态。`

也就是说, 如果我们执行 `pushq` 操作, 我们有两种实现方式：

1. 将 `%rsp` 更新为 `%rsp-8`, 接着我们读取 `%rsp` 的值并将内容写入该地址

   > 很显然这是不符合我们的原则的, 写的操作在读的操作之前, 我们要等待指令更新后才能去读

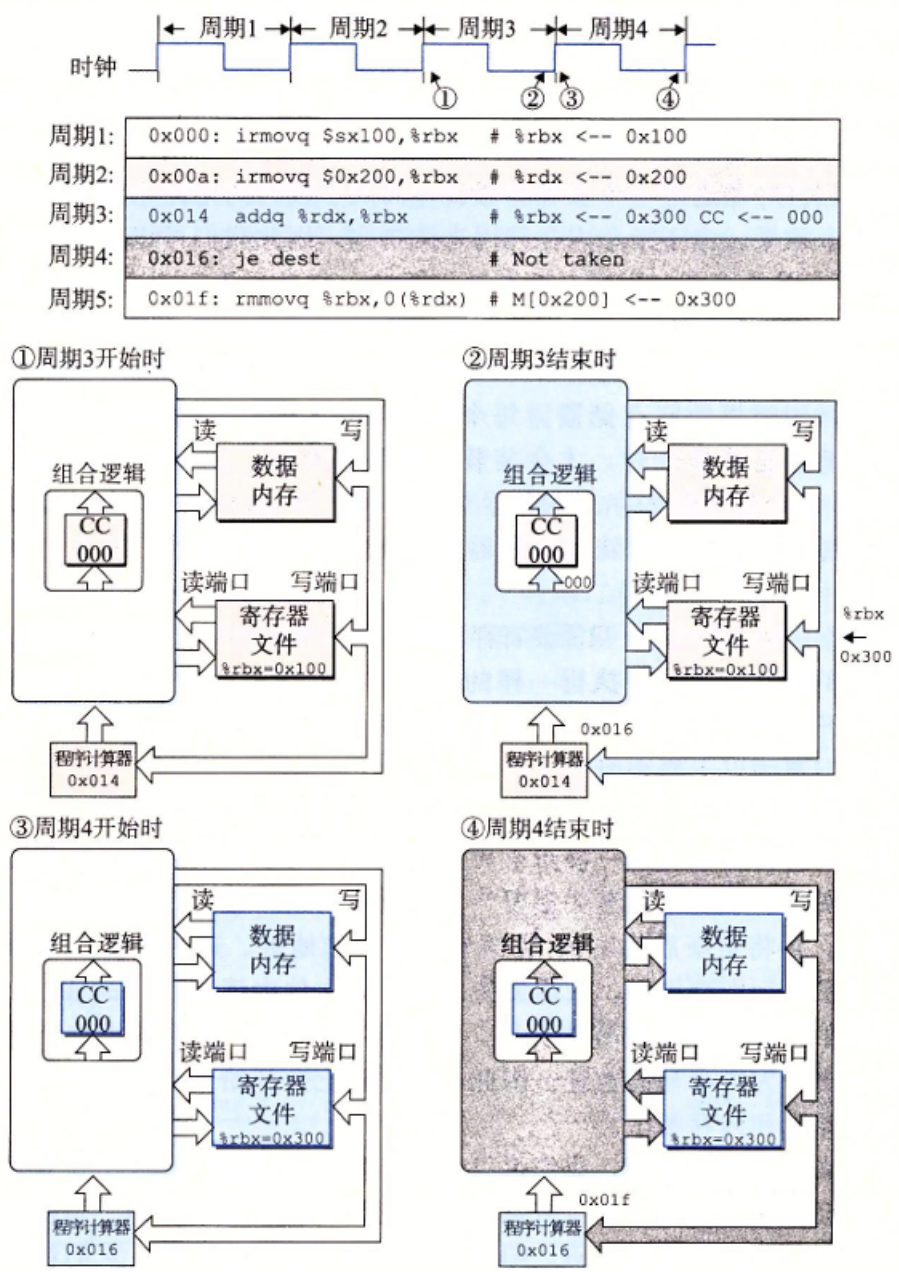
2. 使用 `ValE` 记录 `%rsp - 8` 的值, 接着同时更新 `%rsp` 和写入该地址

   > 这种方法是正确的, 它使读和写一起进行了

以下是汇编代码，SEQ硬件处理其中的3、4行指令：

```
1     0x000:   irmovq $0x100,%rbx   # %rbx <-- 0x100
2     0x00a:   irmovq $0x200,%rdx   # %rdx <-- 0x200
3     0x014:   addq %rdx,%rbx       # %rbx <-- 0x300 CC <-- 000
4     0x016:   je dest              # Not taken
5     0x01f:   rmmovq %rbx,0(%rdx)  # M[0x200] <-- 0x300
6     0x029: dest: halt
```

跟踪SEQ的两个执行周期，每个周期开始时，状态单元（寄存器和内存）是根据前一条执行设置的，信号传播通过组合逻辑，创建出新的状态单元的值，在下一个周期开始时（时钟上升），这些值会被加载到状态单元。如下图中，当周期3开始时，上升前状态单元中的值是周期1组合逻辑创建，上升后状态单元被修改该为周期2组合逻辑创建；当周期3结束时，状态单元保持不变，周期四开始时（上升后），状态单元时周期3组合逻辑创建



SEQ 各阶段的实现

建议看视频并且结合书上图片

【CSAPP-深入理解计算机系统】4-4. Y86-64处理器硬件结构 *哔哩哔哩* bilibili

看完可以对照上一节第一张大图进行巩固

流水线的通用原理

个人觉得 CSAPP 原书举的例子十分的贴切：

洗车店洗车的时候会有许多的步骤，如果按照我们之前顺序实现的步骤，就是只有一辆车经历完所有的步骤后下一辆车才能开始

而本节我们介绍的流水线方法就是拆分整个洗车过程为多个步骤，一辆车完成一个步骤后下一辆车立马接上

流水线化的一个重要特性就是提高了系统的**吞吐量**（throughput），也就是单位时间内服务的顾客总数，不过它也会轻微地增加**延迟**（latency），也就是服务一个用户所需要的时间。

计算流水线

一个很简单的非流水线化的硬件系统。它是由一些执行计算的逻辑以及一个保存计算结果的寄存器组成的。时钟信号控制在每个特定的时间间隔加载寄存器。图中的计算块是用组合逻辑来实现的，意味着信号会穿过一系列逻辑门，在一定时间的延迟之后，输出就成为了输入的某个函数。

在现代逻辑设计中，电路延迟以**皮秒**（picosecond，简写成“ps”），也就是 10^{-12} 秒为单位来计算。

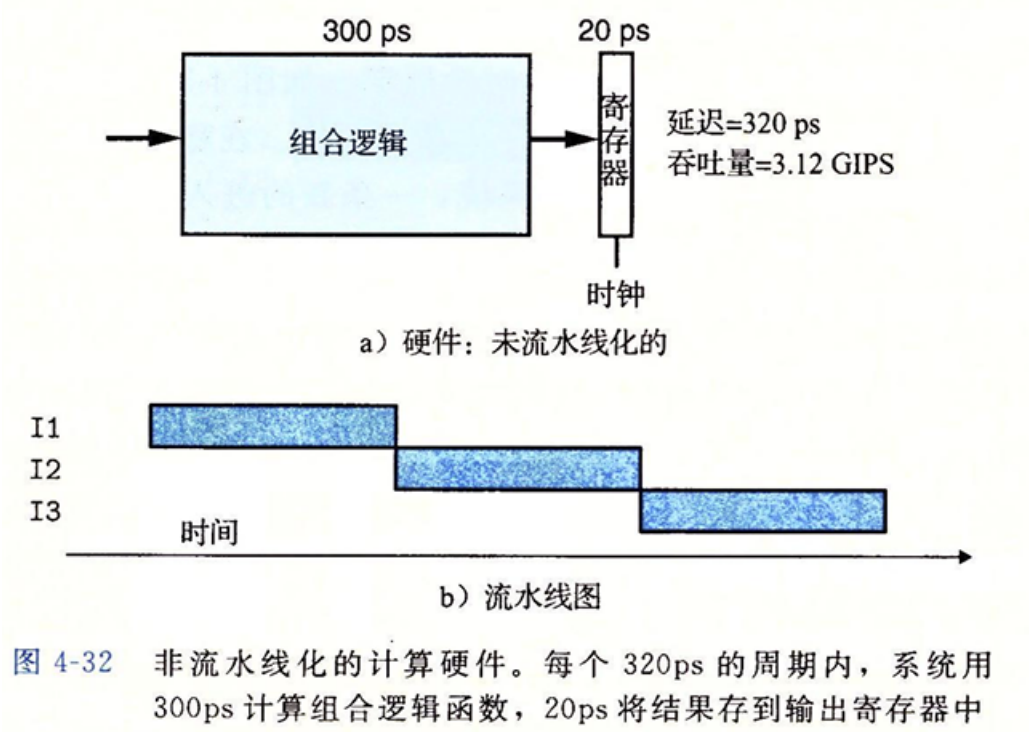


图 4-32　非流水线化的计算硬件。每个 320ps 的周期内，系统用 300ps 计算组合逻辑函数，20ps 将结果存到输出寄存器中

如上图, 当我们完成一条指令就需要 300ps 的延迟, 之后花费 20ps 的时间将结果放入寄存器, 这样整个过程才算结束

为了衡量一种方法的效率, 我们引入了一个计量方式 : **吞吐量**

我们以 **每秒千兆条指令**（GIPS），也就是每秒十亿条指令，为单位来描述**吞吐量**。

我们简单计算一下上图的吞吐量 :

$$\text{吞吐量} = \frac{1\text{条指令}}{(20+300)\text{ps}} \cdot \frac{1000\text{ps}}{1\text{ns}} \approx 3.12\text{GIPS}$$

但如果我们将系统执行的计算分成三个阶段 `A B C`。然后在各个阶段之间放上**流水线寄存器**（pipeline register），这样每条指令都会按照三步经过这个系统，从头到尾需要三个完整的时钟周期。在稳定状态下，三个阶段都应该是活动的，每个时钟周期，一条指令离开系统，一条新的进入。

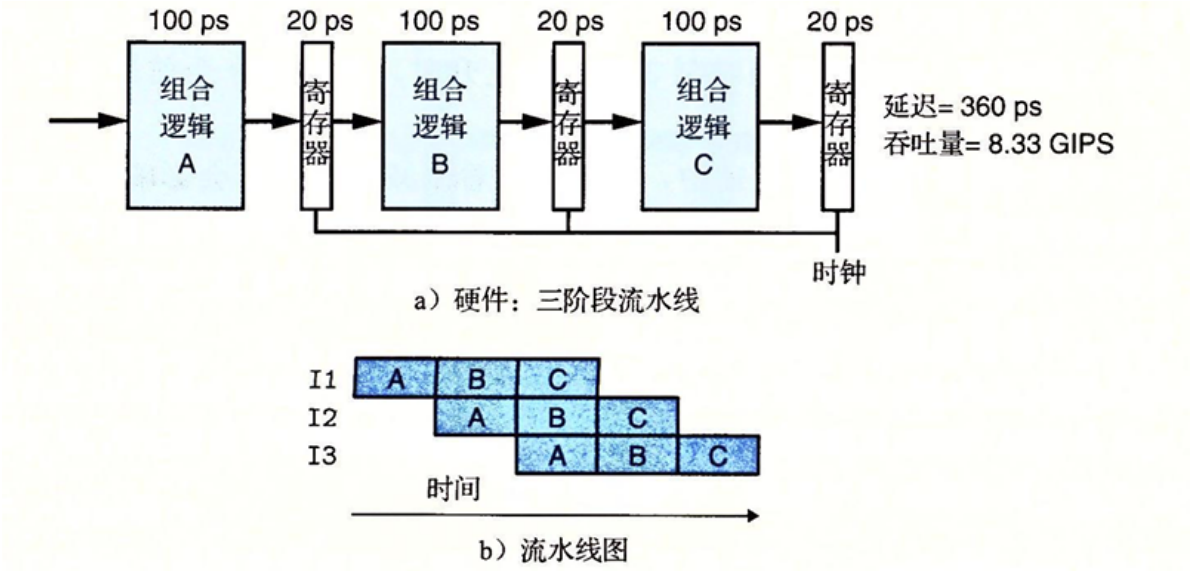


图 4-33　三阶段流水线化的计算硬件。计算被划分为三个阶段 A、B 和 C。每经过一个 120ps 的周期，每条指令就行进通过一个阶段

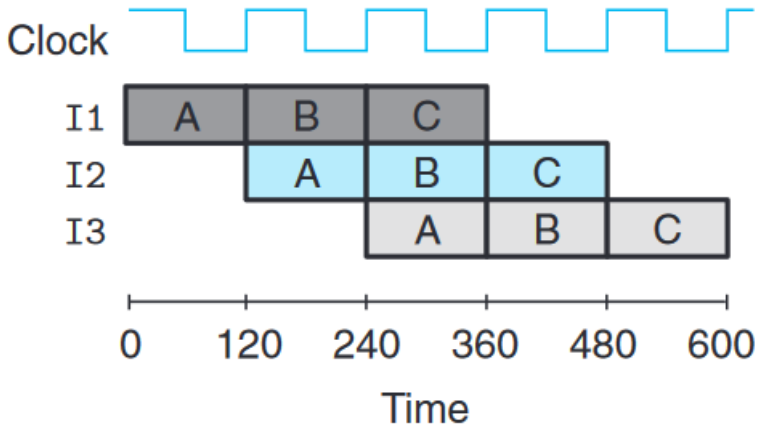
我们将系统吞吐量提高到原来的 `2.67` 倍，代价是增加了一些硬件，以及延迟的少量增加 `1.12` 倍。延迟变大是由于增加的流水线寄存器的时间开销。

$$\text{吞吐量} = \frac{1\text{条指令}}{(20+100)\text{ps}} \cdot \frac{1000\text{ps}}{1\text{ns}} \approx 8.33\text{GIPS} \approx 2.67 \times 3.12$$

流水线操作的详细说明

流水线阶段之间的指令转移是由时钟信号来控制的，每隔 120ps，信号从 0 上升至 1 ，开始下一组流水线阶段的计算。

> 120ps 是按照上图得出的结果, 因为每个步骤的延时都是 120ps



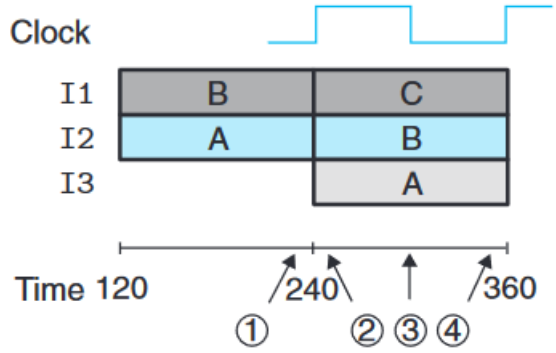
这种在组合逻辑块之间采用时钟寄存器的简单机制（当时钟上升时，输入被加载到流水线寄存器中，成为寄存器的输出），足够控制流水线中的指令流。随着时钟周而复始地上升和下降，不同的指令就会通过流水线的各个阶段，不会相互干扰。

> 注意是从 0 上升至 1 才开始下一组流水线阶段的计算, 从 1 到 0 的过程系统仍在执行的过程中

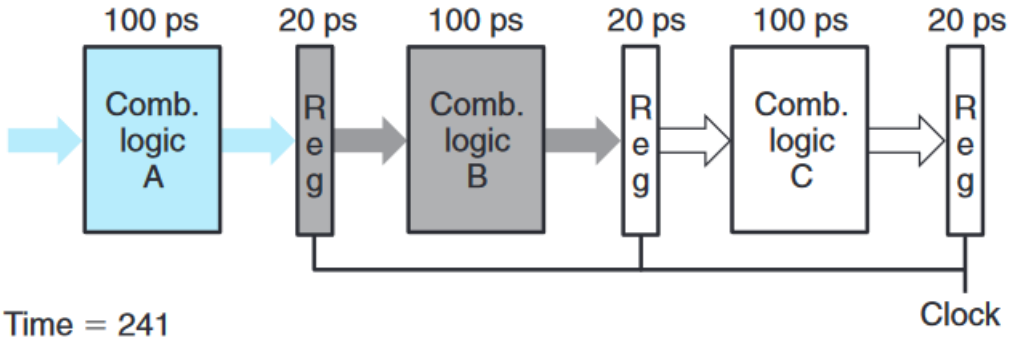
时钟周期对流水线行为的影响：

- 减缓时钟不会影响流水线的行为。信号传播到流水线寄存器的输入，但是直到时钟上升时才会改变寄存器的状态。
- 如果时钟运行得太快，就会有灾难性的后果。值可能会来不及通过组合逻辑，因此当时钟上升时，寄存器的输入还不是合法的值。

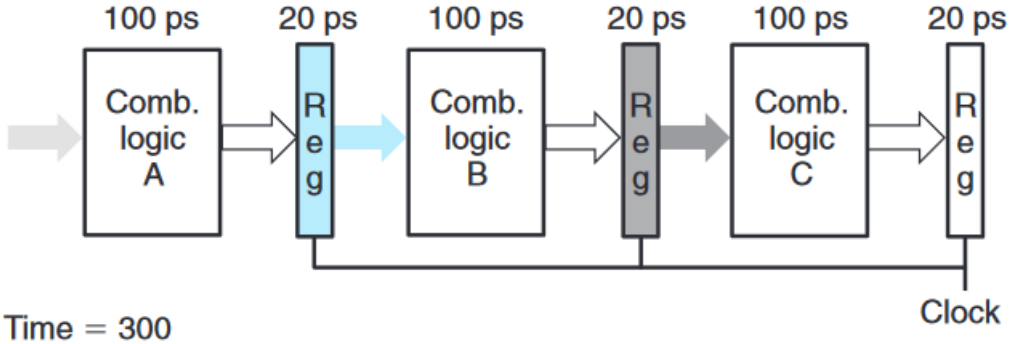
我们简单分析一下 239ps ~ 359ps 的上述流水线的过程



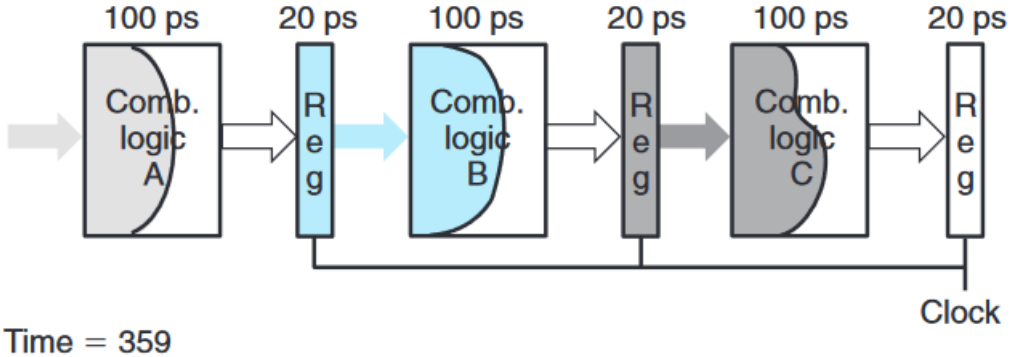
① Time = 239



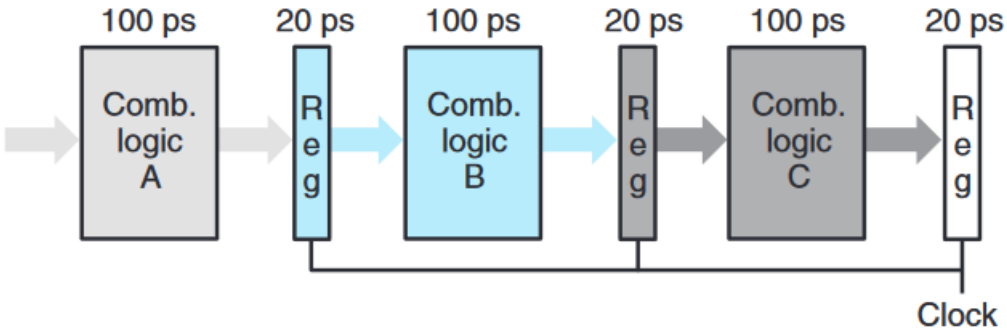
② Time = 241



③ Time = 300



④ Time = 359



- Time = 239

> 这个时候模块A和模块B已经完成了计算, 计算出来的结果保存在模块本身

- $\text{Time} = 241$

> 模块A将结果传输给下一个寄存器, 模块B将结果传输给下一个寄存器
>
> 同时模块A和模块B清空自身存储的结果准备下一次计算
>
> 注意这个时候I3对于模块A的输入已经被传输到了模块A的前一个寄存器, 等传输完成就可以执行模块A了

- $\text{Time} = 300$

> 模块A, B, C都在执行过程中

- $\text{Time} = 354$

> 模块A, B, C全部运算完成并储存在模块中, 等时钟变化后传输给下一个寄存器

流水线的局限性

不一致的划分

运行时钟的速率是由最慢的阶段的延迟限制的。每个时钟周期，阶段 A 和都阶段 C 都会空闲，只有阶段 B 会一直处于活动状态。

对硬件设计者来说，将系统计算设计划分成一组具有相同延迟的阶段是一个严峻的挑战。通常，处理器中的某些硬件单元，如 ALU 和内存，是不能被划分成多个延迟较小的单元的。

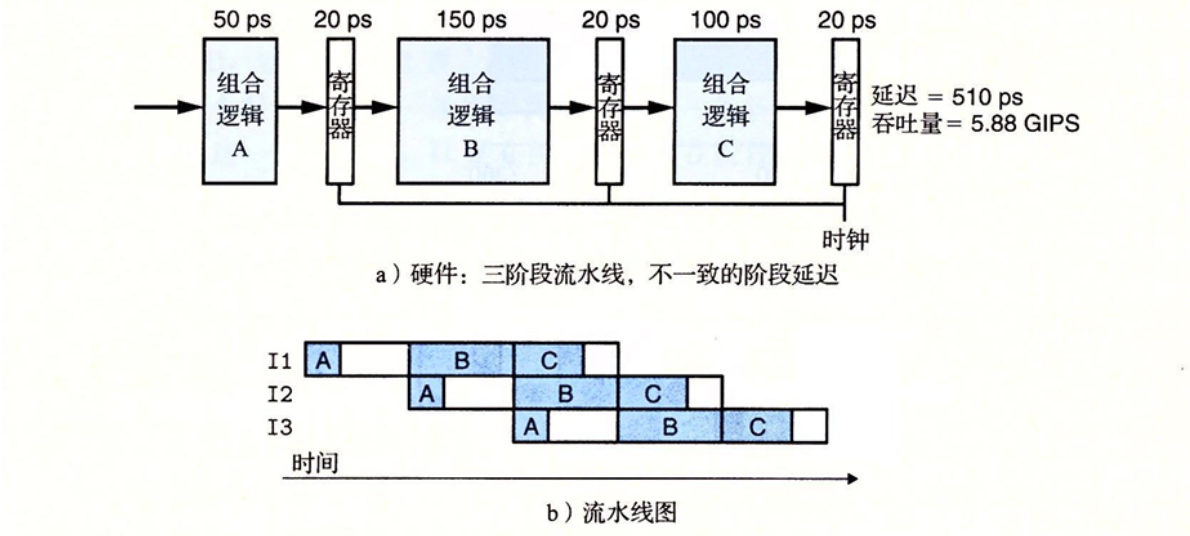


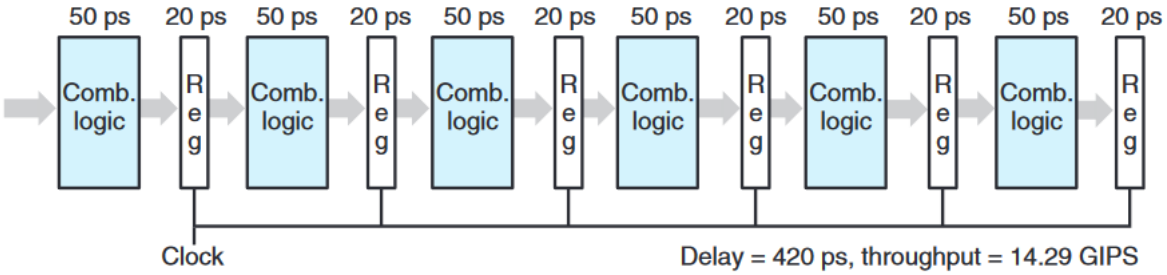
图 4-36　由不一致的阶段延迟造成的流水线技术的局限性。系统的吞吐量受最慢阶段的速度所限制

流水线过深，收益反而下降

我们将计算分为了 6 个阶得到了一个六阶段流水线，虽然我们将每个计算时钟的时间缩短了两倍，但是由于通过流水线寄存器的延迟，吞吐量并没有加倍。这个延迟成了流水线吞吐量的一个制约因素，这个延迟占到了整个时钟周期的 28.6%。

为了提高时钟频率，现代处理器采用了很深的（15 或更多的阶段）流水线。

- 处理器架构师将指令的执行划分成很多非常简单的步骤，这样一来每个阶段的延迟就很小。
- 电路设计者小心地设计流水线寄存器，使其延迟尽可能得小。
- 芯片设计者也必须小心地设计时钟传播网络，以保证时钟在整个芯片上同时改变。



带反馈的流水线系统

如果不同的指令之间是相互独立的, 那么我们的流水线系统将一直不断正确运行下去, 但是实际上指令之间会有相互依赖关系

我们观察下面两组汇编代码

```
irmovq $50, %rax
addq %rax, %rbx
mrmovq 100( %rbx ), %rdx
```

此时你会发现 `addq` 有一个参数 `%rax` 依赖于第一行的指令 (需要第一行的指令完全执行完才能知道 `%rax` 的值)

但是我们的流水线系统使第一行还没执行完成就让第二行指令开始执行了

这种依赖叫做 **数据依赖**

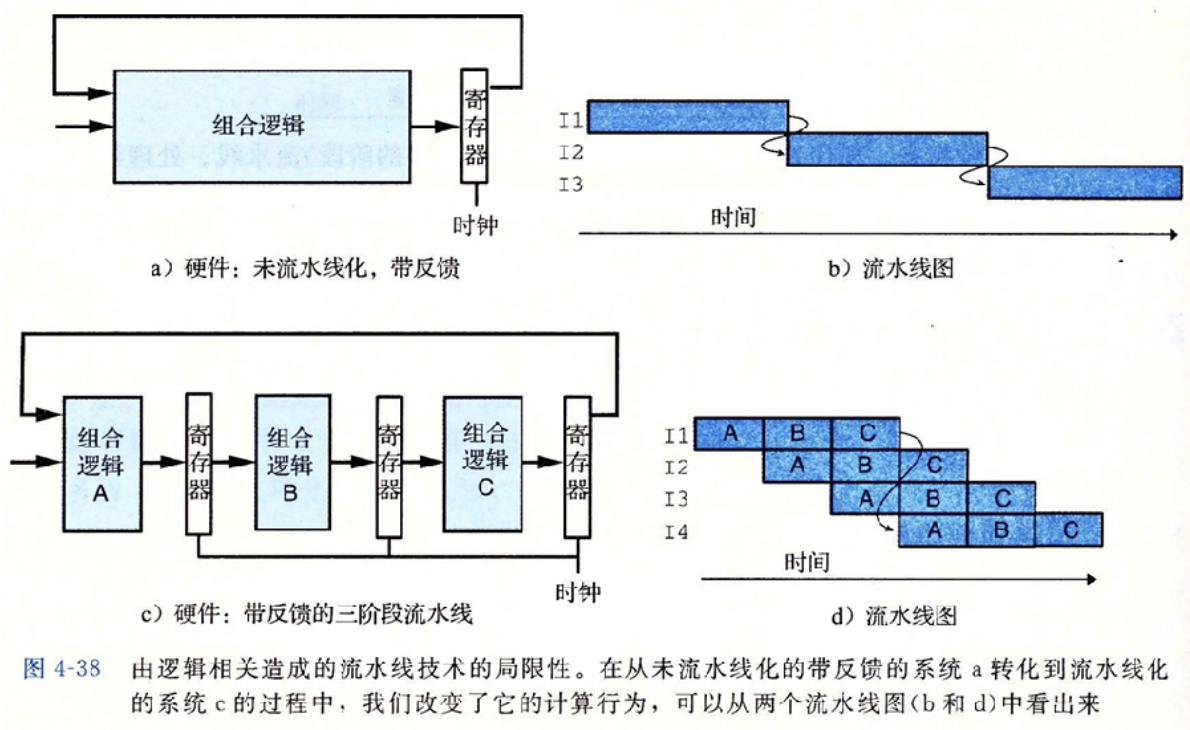
```
loop:
    subq %rdx, %rbx
    jne targ
    irmovq $10, %rdx
    jmp loop
targ:
    halt
```

我们发现跳转指令需要上一条指令的 `CC` 来判断是否跳转

这种依赖叫做 **控制依赖**

这些相关都是由反馈路径来解决的，这些反馈将更新了的寄存器值向下传送到寄存器文件，将新的值向下传送到 PC 寄存器。

将流水线引入含有反馈路径的系统中的危险。在一个三阶段流水线，我们将改变系统的行为



a）硬件：未流水线化，带反馈　　b）流水线图

c）硬件：带反馈的三阶段流水线　　d）流水线图

图 4-38　由逻辑相关造成的流水线技术的局限性。在从未流水线化的带反馈的系统 a 转化到流水线化的系统 c 的过程中，我们改变了它的计算行为，可以从两个流水线图(b 和 d)中看出来

我们在流水线中一般不使用反馈

Y86-64 的流水线实现

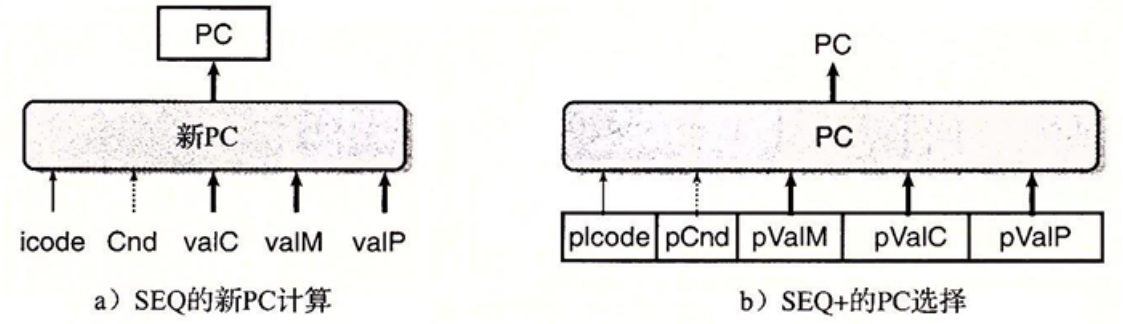
SEQ+：重新安排计算阶段

为了能够更好的实现流水线, 我们以顺序实现为基础, 简单调整 `更新PC` 的位置

我们将 `更新PC` 的位置从时钟周期结束改变到时钟周期开始的时候, 即现在的步骤为 :

更新PC ⇒ 取指 ⇒ 译码 ⇒ 执行 ⇒ 访存

`SEQ` 到 `SEQ+` 中对状态单元的改变是一种很通用的改进的例子，这种改进称为**电路重定时**（circuit retiming）。重定时改变了一个系统的状态表示，但是并不改变它的逻辑行为。通常用它来平衡一个流水线系统中各个阶段之间的延迟。



a）SEQ的新PC计算　　b）SEQ+的PC选择

图 4-39　移动计算 PC 的时间。在 SEQ＋中，我们将计算当前状态的程序计数器的值作为指令执行的第一步

> **为什么要把PC移动到时钟周期之前以及它的影响**
>
> 假设我们现在有六个指令 A, B, C, D, E, F , 我们用下标来表示当前这条指令在进行哪一个阶段
>
> 我们先不考虑 `更新PC` 这个阶段, 那么我们执行一条指令就只要经历四个阶段 : 取指, 译码, 执行, 访存

> 我们可以通过下标来看具体执行到了那一个阶段了, 如 C_3 代表 C 指令在经历执行阶段
>
> 假设某一个时钟周期开始的时候流水线的情况是 : A_4, B_3, C_2, D_1 , 那么下一个时钟周期开始的时候, 流水线的情况应该是 : B_4, C_3, D_2, F_1, 如果我们 `更新PC` 步骤是在四个阶段之后, 那么计算出来的地址应该是 B 的地址, 这跟我们需要的 PC 是不符合的, 但如果我们将 `更新PC` 放在取指阶段之前, 那么 D_1 时计算出来的 `icode` , `valC`, `valP` 将会帮我们计算出 F 的地址
>
> 我们同时发现, 计算 `PC` 需要 `pCnd` 和 `pValM` , 这两个值分别是在执行和访存的时候得出的, 所以如果执行跳转相关指令的时候 `PC` 值是不能够正确计算的, 这个时候我们会使用 `分支预测` 的方法 (分支预测我们之后会讲到)
>
> > 补充 :
> >
> > 当执行跳转指令（如jmp, jXX）时，我们需要将PC设置为跳转目标的地址。这个地址就是pValM的值，它在内存访问阶段从内存中取出。
> >
> > 同样，当执行调用指令（call）时，我们也需要将PC设置为被调用函数的地址。这个地址同样是pValM的值，它在内存访问阶段从内存中取出。
> >
> > 因此，更新PC需要使用到pValM，是因为在处理跳转和调用指令时，我们需要将PC设置为pValM的值，以便于处理器能够正确地跳转到目标地址

插入流水线寄存器

插入流水线寄存器，并对信号重新排列，得到 `PIPE-` 处理器。流水线寄存器用黑色方框表示，每个寄存器包括不同的字段，用白色方框表示。

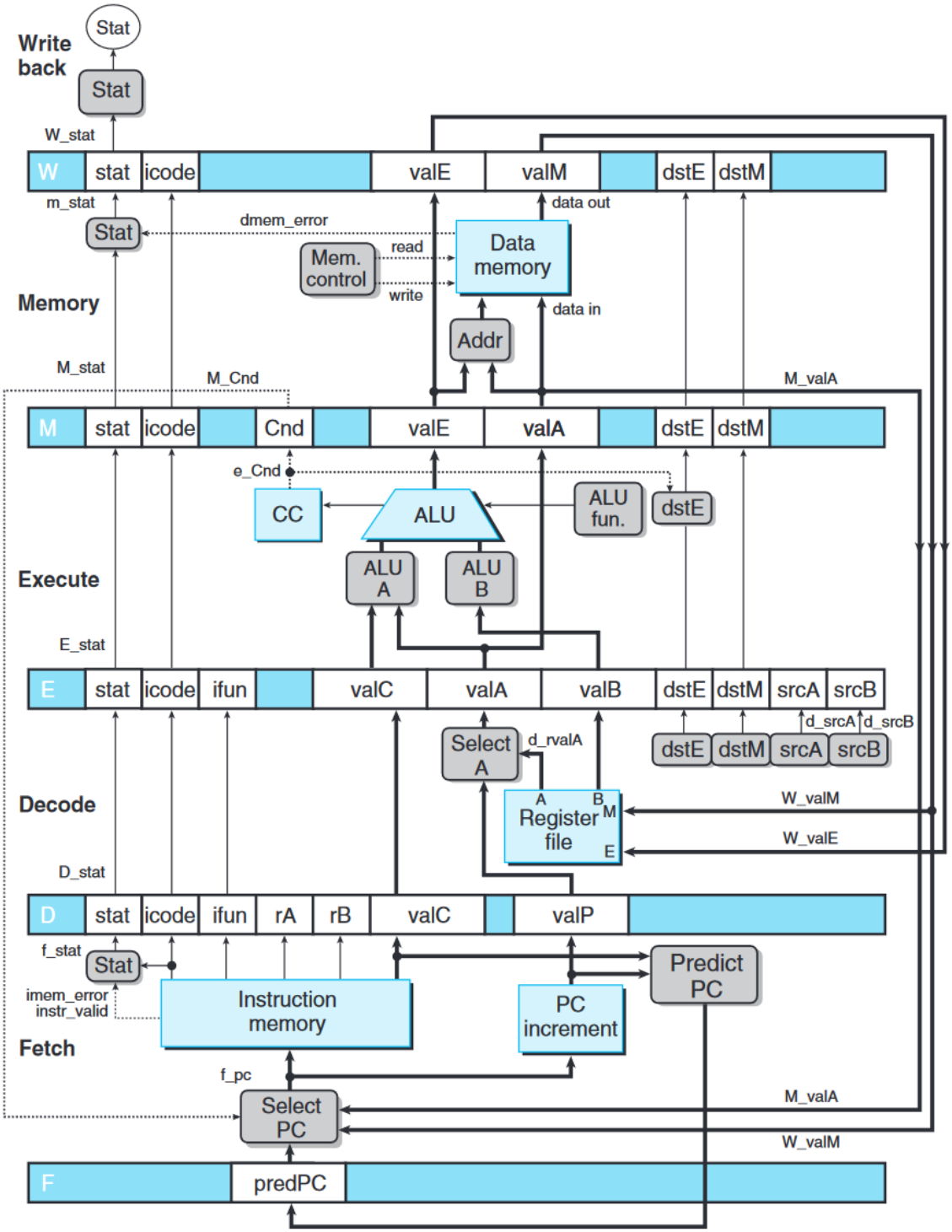


Figure 4.41 Hardware structure of PIPE−, an initial pipelined implementation. By inserting pipeline registers into SEQ+ (Figure 4.40), we create a five-stage pipeline. There are several shortcomings of this version that we will deal with shortly.

- `F`：保存程序计数器的预测值。
- `D`：位于取指和译码阶段之间。它保存关于最新取出的指令的信息，即将由译码阶段进行处理。
- `E`：位于译码和执行阶段之间。它保存关于最新译码的指令和从寄存器文件读出的值的信息，即将由执行阶段进行处理。
- `M`：位于执行和访存阶段之间。它保存最新执行的指令的结果，即将由访存阶段进行处理。它还保存关于用于处理条件转移的分支条件和分支目标的信息。
- `W`：位于访存阶段和反馈路径之间，反馈路径将计算出来的值提供给寄存器文件写，而当完成 `ret` 指令时，它还要向 PC 选择逻辑提供返回地址。

`Predict PC` 块会从 PC 增加器计算出的 `valP` 和取出的指令中得到的 `valC` 中进行选择。这个值存放在流水线寄存器 F 中，作为程序计数器的预测值。

`select PC` 块从三个值中选择一个作为指令内存的地址。

> 动画演示上述过程：
>
> 【CSAPP-深入理解计算机系统】4-6. 流水线硬件结构 *哔哩哔哩* bilibili

对信号进行重新排列和标号

> 这一部分建议直接看书，书上描述的非常通俗

在流水线化的设计中，与各个指令相关联的信号值（如 `valC srcA`）有多个版本，会随着指令一起流过系统。

- 我们采用的命名机制，通过在信号名前面加上大写的流水线寄存器名字作为前缀，存储在流水线寄存器中的信号可以唯一地被标识。
- 我们还需要引用某些在一个阶段内刚刚计算出来的信号。它们的命名是在信号名前面加上小写的阶段名的第一个字母作为前缀。

一些通用规则：

- 作为一条通用原则，我们要保存处于一个流水线阶段中的指令的所有信息。否则，可能将处于写回阶段的指令的值写入，而寄存器 ID 却来自于处于译码阶段的指令。
- 在硬件设计中，像这样仔细确认信号是如何使用的，然后通过合并信号来减少寄存器状态和线路的数量，是很常见的。

预测下一个 PC

除了条件转移指令和 `ret` 以外，根据取指阶段中计算出的信息，我们能够确定下一条指令的地址：

- 如果取出的指令是条件分支指令，要到几个周期后，也就是指令通过执行阶段之后，我们才能知道是否要选择分支。
- 如果取出的指令是 `ret`，要到指令通过访存阶段，才能确定返回地址

在大多数情况下，我们能达到每个时钟周期发射一条新指令的目的。对大多数指令类型来说，我们的预测是完全可靠的。对于 `call` 和 `jmp`（无条件转移），下一条指令的地址是指令中的常数字 `valC`，而对于其他指令来说就是 `valP`。

猜测分支方向并根据猜测开始取指的技术称为**分支预测**。

- **总是选择**（always taken）分支预测策略。总是预测选择了条件分支（预测 PC 的新值为指令中的常数字 `valC`），成功率大约为 `60%`。
- **从不选择**（never taken，NT）策略。成功率大约为 `40%`。
- **反向选择**、**正向不选择**（backward taken， forward not-taken，BTFNT）策略，当分支地址比下一条地址低时就预测选择分支，而分支地址比较高时，就预测不选择分支。这种策略的成功率大约为 `65%`。这种改进源自一个事实，即循环是由后向分支结束的，而循环通常会执行多次。前向分支用于条件操作，而这种选择的可能性较小。
- **使用栈的返回地址预测**

  > 对大多数程序来说，预测返回值很容易，因为过程调用和返回是成对出现的。高性能处理器中运用了这个属性，在取指单元中放入一个硬件栈，保存过程调用指令产生的返回地址。同分支预测一样，在预测错误时必须提供一个恢复机制，因为还是有调用和返回不匹配的时候。通常，这种预测很可靠。这个硬件栈对程序员来说是不可见的。

流水线冒险