[读书笔记]CSAPP：12[B]处理器体系结构：顺序实现

我们想要构建顺序实现的Y86-64处理器，首先需要构建Y86-64指令集体系结构，然后基于该ISA来构建处理器。

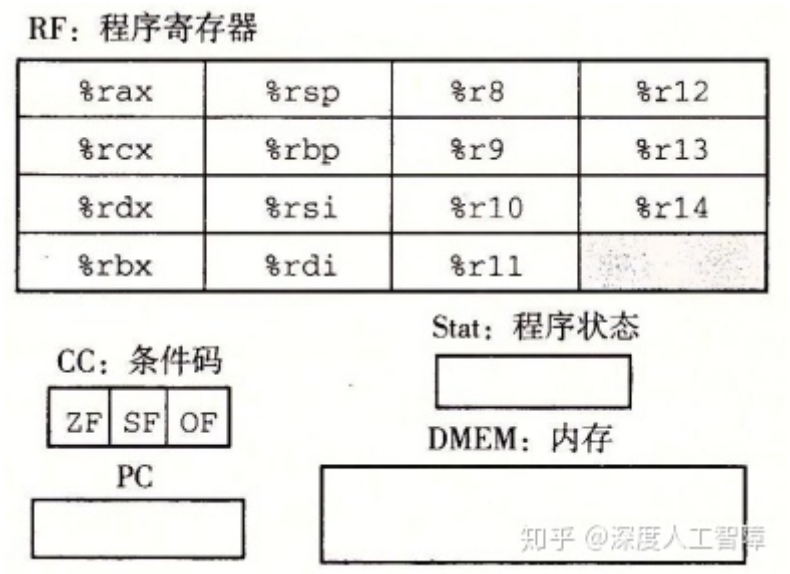
1 Y86-64指令集体系结构

想要定义一个指令集体系结构，需要包含：

- 定义状态单元
- 指令集和他们的编码
- 编程规范和异常事件处理

1.1 状态单元

我们将每条指令会读取或修改处理器状态的部分称为**程序员可见**状态，如下图所示

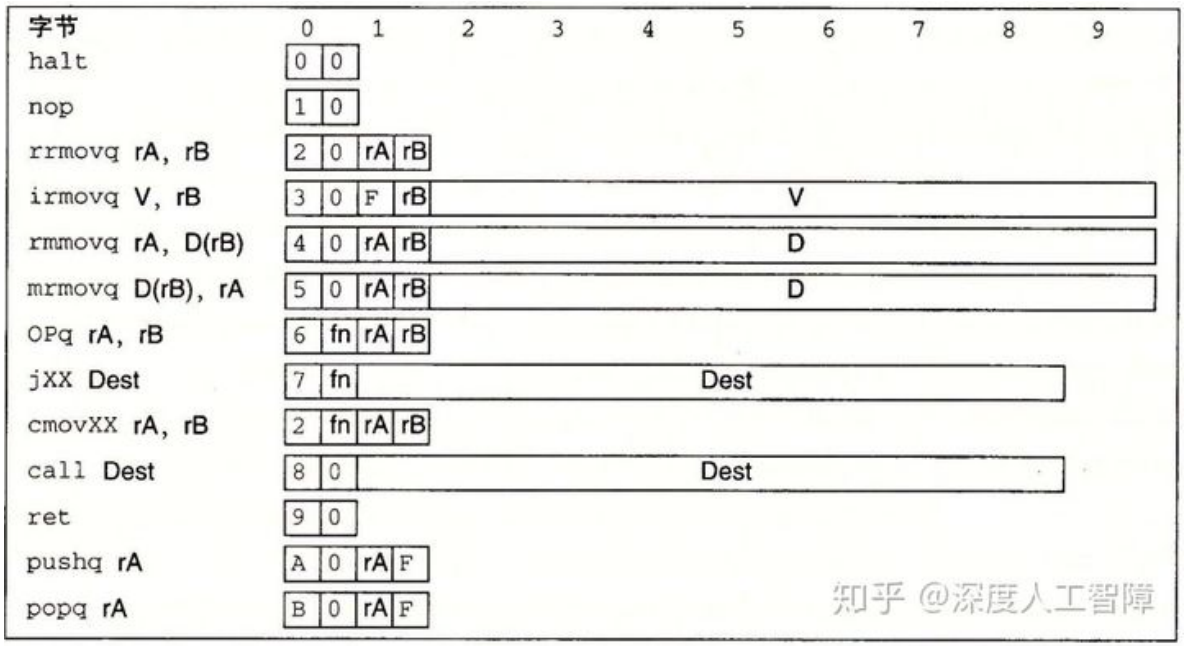


- **程序寄存器RF**：这里对x86-64的寄存器进行省略，降低复杂度。其中`%rsp`用于指示出栈、入栈、函数调用和返回地址。
- **条件码CC**：保存最近算数或逻辑指令造成的影响。
- **程序计数器PC**：保存当前正在执行的指令的地址。
- **内存DEME**：操作系统将物理内存抽象为一个单一的字节数组。
- **程序状态Stat**：表明当前程序运行的状态，取值如下图所示。当出现异常时，处理器会调用异常处理程序，这里为了简化，直接让处理器停止执行指令。

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

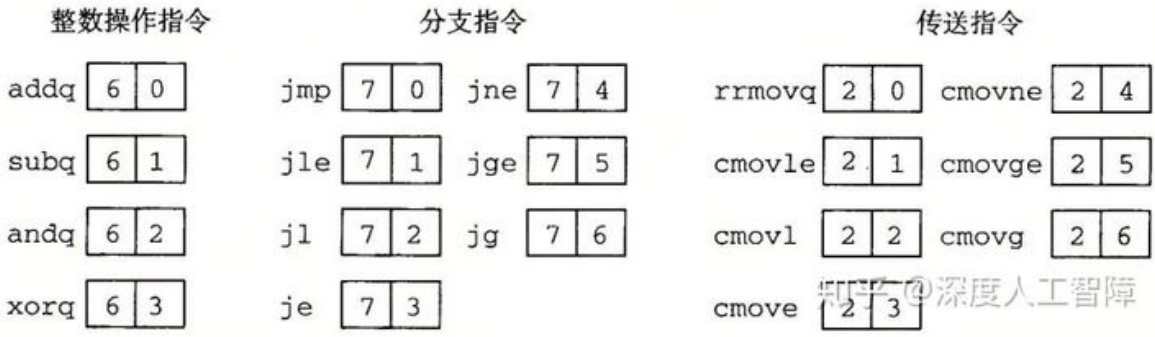
1.2 Y86-64指令及其编码

指令集的设计要求字节编码要有唯一的解释。这里实现如下图所示的Y86-64指令集，左侧为汇编指令，右侧为指令对应的编码。



- 传送指令分成了`rrmovq`、`irmovq`、`rmmovq`和`mrmovq`。其中第一个字母表示源，第二个字母表示目的，`r`寄存器、`i`立即数、`m`内存。这里为了简化，只支持基址和偏移量形式的内存引用。
- 整数操作指令`OPq`，包含`addq`、`subq`、`andq`和`xorq`。会设置条件码`ZF`、`SF`和`OF`。
- 跳转指令`jXX`，包括`jmp`、`jle`、`jl`、`je`、`jne`、`jge`和`jg`。这里`jXX`使用绝对地址。
- 条件传送指令`cmovXX`，包括`cmovle`、`cmovl`、`cmove`、`cmovne`、`cmovge`和`cmovg`。
- `call`使用绝对地址。
- `push`和`pop`是对栈的调整，`push %rsp`是先将`%rsp`的内容保存到栈中，再对`%rsp`减8；`pop %rsp`等价于`mrmovq (%rsp), %rsp`。
- `halt`指令停止指令的执行。

我们将指令的高8位称为**指令指示符**，可以用来确定指令类型，其中高4位为**代码部分**，低4位为**功能部分**。代码部分能确定指令执行的步骤，而功能部分能确定ALU要进行什么运算以及确定条件码的组合，比如功能部分我`1`表示让ALU进行减法运算并设置对应的条件码，则整数操作指令会得到计算结果，分支指令和传送指令会根据条件码来确定是否跳转和传送。`OPq`、`jXX`和`cmovXX`的编码如下图所示



对于指令操作数的编码：

- 当指令不包含操作数时，比如`ret`，只需要一个字节长度。
- 当指令包含寄存器作为操作数时，需要额外添加**寄存器指示符字节**，可以使用如下图所示的寄存器标识符，当只有一个寄存器操作数时，需要将另4个字节标识为`0xF`。此时需要两个字节长度。

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

- 当指令包含立即数、带偏移量的内存引用和目标地址，需要额外的8字节进行编码。

例： `rmmovq %rsp, 0x123456789abcd(%rdx)` 的编码

`rmmovq` 的编码形式为

其中，`%rsp` 的寄存器标志符为 `0x4`，`%rdx` 的寄存器标志符为 `0x2`。这里立即数不满足8字节，所以需要在其前面填充得到8字节，即 `0x000123456789abcd`，由于采用小端法，所以要对字节顺序进行调整得到 `cd ab 89 67 45 23 01 00`。最终将所有编码都拼接起来，得到 `40 42 cd ab 89 67 45 23 01 00`。

1.3 Y86-64程序例子

```
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:
```


以上是Y86-64完整的汇编代码。其中以`.`开头的为**汇编器伪指令（Assembler Directives）**，用于让汇编器调整地址。`.pos address`表示从`address`处开始产生代码，如第2行的`.pos 0`表示从0开始产生代码，第`.pos 0x200`表示从`0x200`处开始产生栈。`.align 8`表示在8字节边界处对其。

这里可以通过汇编器YAS将其转变为可执行代码

```
                            | # Execution begins at address 0
0x000:                      |   .pos 0
0x000: 30f40002000000000000 |   irmovq stack, %rsp      # Set up stack pointer
0x00a: 803800000000000000   |   call main               # Execute main program
0x013: 00                   |   halt                    # Terminate program
                            |
                            | # Array of 4 elements
0x018:                      |   .align 8
0x018:                      | array:
0x018: 0d000d000d000000     |   .quad 0x000d000d000d
0x020: c000c000c0000000     |   .quad 0x00c000c000c0
0x028: 000b000b000b0000     |   .quad 0x0b000b000b00
0x030: 00a000a000a00000     |   .quad 0xa000a000a000
                            |
0x038:                      | main:
0x038: 30f71800000000000000 |   irmovq array,%rdi
0x042: 30f60400000000000000 |   irmovq $4,%rsi
0x04c: 805600000000000000   |   call sum                # sum(array, 4)
0x055: 90                   |   ret
                            |
                            | # long sum(long *start, long count)
                            | # start in %rdi, count in %rsi
0x056:                      | sum:
0x056: 30f80800000000000000 |   irmovq $8,%r8         # Constant 8
0x060: 30f90100000000000000 |   irmovq $1,%r9         # Constant 1
0x06a: 6300                 |   xorq %rax,%rax        # sum = 0
0x06c: 6266                 |   andq %rsi,%rsi        # Set CC
0x06e: 708700000000000000   |   jmp     test          # Goto test
0x077:                      | loop:
0x077: 50a70000000000000000 |   mrmovq (%rdi),%r10    # Get *start
0x081: 60a0                 |   addq %r10,%rax        # Add to sum
0x083: 6087                 |   addq %r8,%rdi         # start++
0x085: 6196                 |   subq %r9,%rsi         # count--.  Set CC
0x087:                      | test:
0x087: 747700000000000000   |   jne    loop           # Stop when 0
0x090: 90                   |   ret                   # Return
                            |
                            | # Stack starts here and grows to lower addresses
0x200:                      |   .pos 0x200
0x200:                      | stack:
```


这里实现了一个指令集模拟器YIS，可以模拟Y86-64机器代码程序的执行

```
Stopped in 34 steps at PC = 0x13.  Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%rax:   0x0000000000000000      0x0000abcdabcdabcd
%rsp:   0x0000000000000000      0x0000000000000200
%rdi:   0x0000000000000000      0x0000000000000038
%r8:    0x0000000000000000      0x0000000000000008
%r9:    0x0000000000000000      0x0000000000000001
%r10:   0x0000000000000000      0x0000a000a000a000

Changes to memory:
0x01f0: 0x0000000000000000      0x0000000000000055
0x01f8: 0x0000000000000000      0x0000000000000013
```

2 Y86-64的顺序实现

2.1 处理指令的阶段

处理一条指令我们可以将其划分成若干个阶段：

1. **取指（Fetch）：** 根据程序计数器PC从内存中读取指令字节。然后完成以下步骤
2. 1. 从指令中提取出指令指示符字节，并且确定出指令代码（`icode`）和指令功能（`ifun`）
   2. 如果存在寄存器指示符，则从指令中确定两个寄存器标识符`rA`和`rB`
   3. 如果存在常数字，则从指令中确定`valC`
   4. 根据指令指令长度以及指令地址，可确定下一条指令的地址`valP`
3. **译码（Decode）：** 如果存在`rA`和`rB`，则译码阶段会从寄存器文件中读取`rA`和`rB`的值`valA`和`valB`。对于`push`和`pop`指令，译码阶段还会从寄存器文件中读取`%rsp`的值。
4. **执行（Execute）：** 算术逻辑单元（ALU）会根据`ifun`的值执行对应的计算，得到结果`valE`，包括
5. 1. 计算运算结果，会设置条件码的值，则条件传送和跳转指令会根据`ifun`来确定条件码组合，确定是否跳转或传送。
   2. 计算内存引用的有效地址
   3. 增加或减少栈指针
6. **访存（Memory）：** 写入内存或从内存读取数据`valM`。
7. **写回（Write Back）：** 将结果写入寄存器文件中。
8. **更新PC（PC Update）：** 将PC更新为`valP`，使其指向下一条指令。

接下来将Y86-64的指令按照以上框架进行整理

阶段	OPq rA, rB	rrmovq rA, rB	irmovqV, rB
取指	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valC ← M_8[PC+2] valP ← PC+10
译码	valA ← R[rA] valB ← R[rB]	valA ← R[rA]	
执行	valE ← valB OP valA Set CC	valE ← 0+valA	valE ← 0+valC
访存			
写回	R[rB]← valE	R[rB]← valE	R[rB]← valE
更新 PC	PC ← valP	PC ← valP	PC ← valP

Y86-64 指令 OPq、rrmovq 和 irmovq 在顺序实现中的计算。这些指令计算了一个值，并将结果存放在寄存器中。符号 icode:ifun 表明指令字节的两个组成部分，而 rA:rB 表明寄存器指示符字节的两个组成部分。符号 M_1[x]表示访问(读或者写)内存位置 x 处的一个字节，而 M_8 表示访问八个字节

这里可以发现，相同`icode`具有相同的步骤，而相同的`ifun`在执行阶段具有相同的计算方式，比如`addq`、`jmp`和`rrmovq`的`ifun`都是0，所以都进行加法计算。

注意： `OPq`中会将`ifun`传入给ALU来确定`OP`的类型。

阶段	rmmovq rA，D(rB)	mrmovq D(rB)，rA
取指	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valC ← M_8[PC+2] valP ← PC+10	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valC ← M_8[PC+2] valP ← PC+10
译码	valA ← R[rA] valB ← R[rB]	valB ← R[rB]
执行	valE ← valB+valC	valE ← valB+valC
访存	M_8[valE]← valA	valE ← M_8[valE]
写回		
		R[rA]← valM
更新 PC	PC ← valP	PC ← valP

阶段	pushq rA	popq rA
取指	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2
译码	valA ← R[rA] valB ← R[%rsp]	valA ← R[%rsp] valB ← R[%rsp]
执行	valE ← valB+(−8)	valE ← valB+8
访存	M_8[valE]← valA	valE ← M_8[valA]
写回	R[%rsp]← valE	R[%rsp]← valE R[rA]← valM
更新 PC	PC ← valP	PC ← valP

注意： `pop` 在译码阶段读了两次栈顶指针的值，这是为了使后续流程和别的指令相似。

阶段	jXX Dest	call Dest	ret
取指	icode:ifun ← M_1[PC] valC ← M_8[PC+1] valP ← PC+9	icode:ifun ← M_1[PC] valC ← M_8[PC+1] valP ← PC+9	icode:ifun ← M_1[PC] valP ← PC+1
译码		valB ← R[%rsp]	valA ← R[%rsp] valB ← R[%rsp]
执行	Cnd ← Cond(CC，ifun)	valE ← valB+(−8)	valE ← valB+8
访存		M_8[valE]← valP	valM ← M_8[valA]
写回		R[%rsp]← valE	R[%rsp]← valE
更新 PC	PC ← Cnd?valC:valP	PC ← valC	PC ← valM

这里都会对PC值进行修改，`jXX` 在执行阶段会根据 `ifun` 和条件码来设置是否跳转 `Cnd`。`call` 需要将下一条指令的地址作为返回值存入栈中，并修改栈顶指针的值。`ret` 需要从栈顶指针出获得返回地址，将其设置为PC，并且还要修改栈顶指针的值。

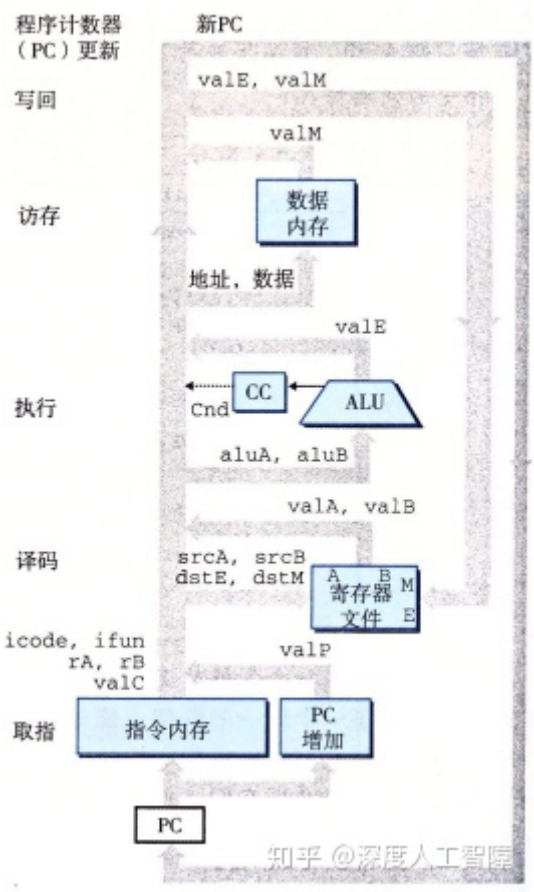
这里的每一行表示某个值的计算（比如`valP`）或者激活某个硬件单元（比如内存），我们列出这些计算和动作，其中增加了`valA`的源`srcA`、`valB`的源`srcB`、写入计算结果`valE`的寄存器`dstE`、写入内存值`valM`的寄存器`dstM`。

阶段	计算	OPq rA, rB	mrmovq D(rB), rA
取指	icode:ifun rA, rB valC valP	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valC ← M_8[PC+2] valP ← PC+10
译码	valA, srcA valB, srcB	valA ← R[rA] valB ← R[rB]	 valB ← R[rB]
执行	valE Cond. codes	valE ← valB OP valA Set CC	valE ← valB+valC
访存	Read/write		valM ← M_8[valE]
写回	E port, dstE M port, dstM	R[rB]← valE	 R[rA]← valM
更新 PC	PC	PC ← valP	PC ← valP

接下来将创建硬件设计来实现这6个阶段，并将其连接起来。

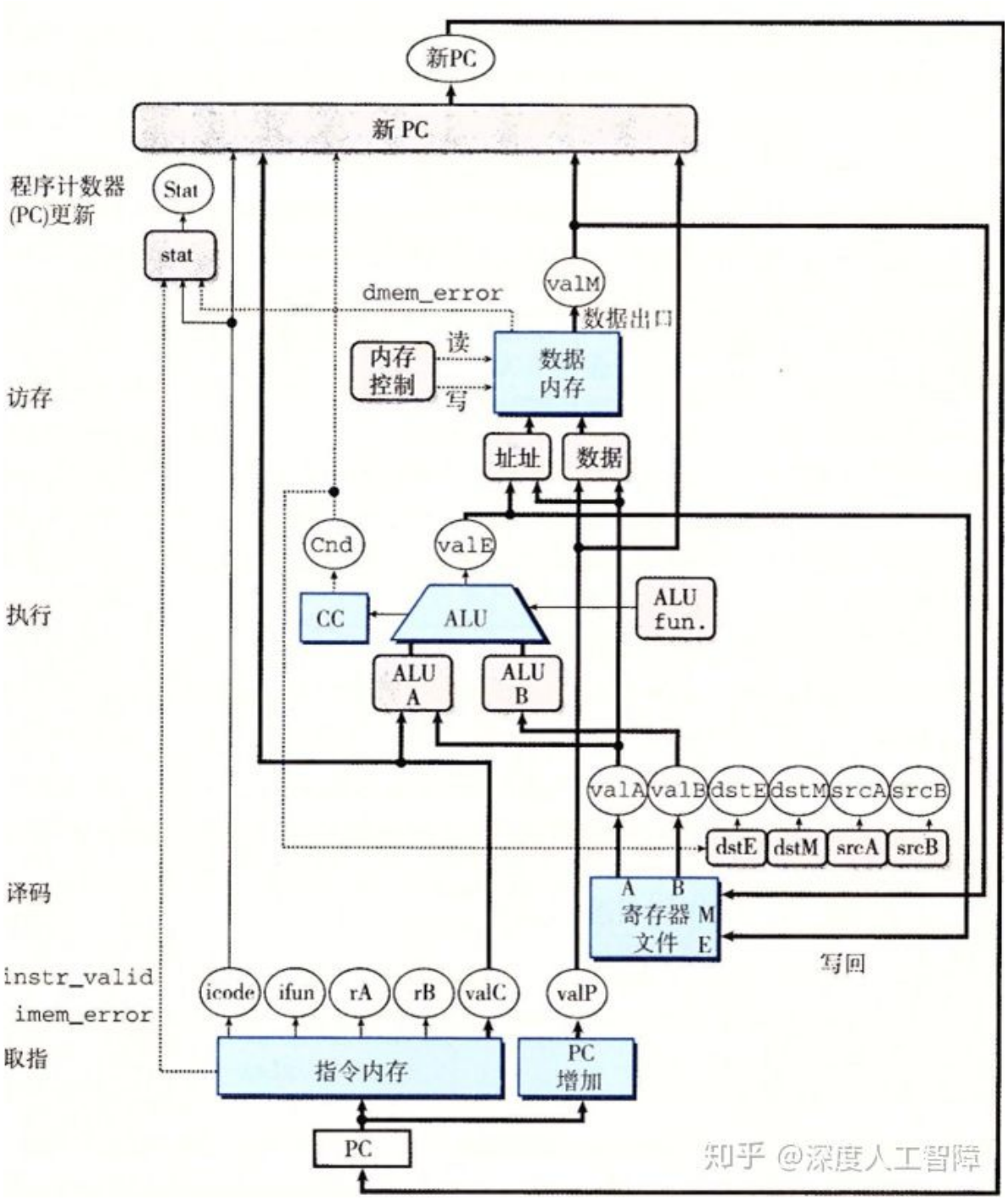
2.2 SEQ硬件结构

我们可以得到顺序实现的SEQ抽象视图



- 数据内存和指令内存都是在相同的内存空间中，只是根据不同的功能对其进行划分
- 寄存器文件包含两个读端口`A`和`B`，以及两个写端口`M`和`E`，分别接收来自内存的值`valM`以及ALU计算的结构`valE`。
- PC更新的值可能来自于：下一条指令地址`valP`、来自内存的值`valM`、调用指令或跳转指令的目标地址`valC`。

更加详细的图如下所示

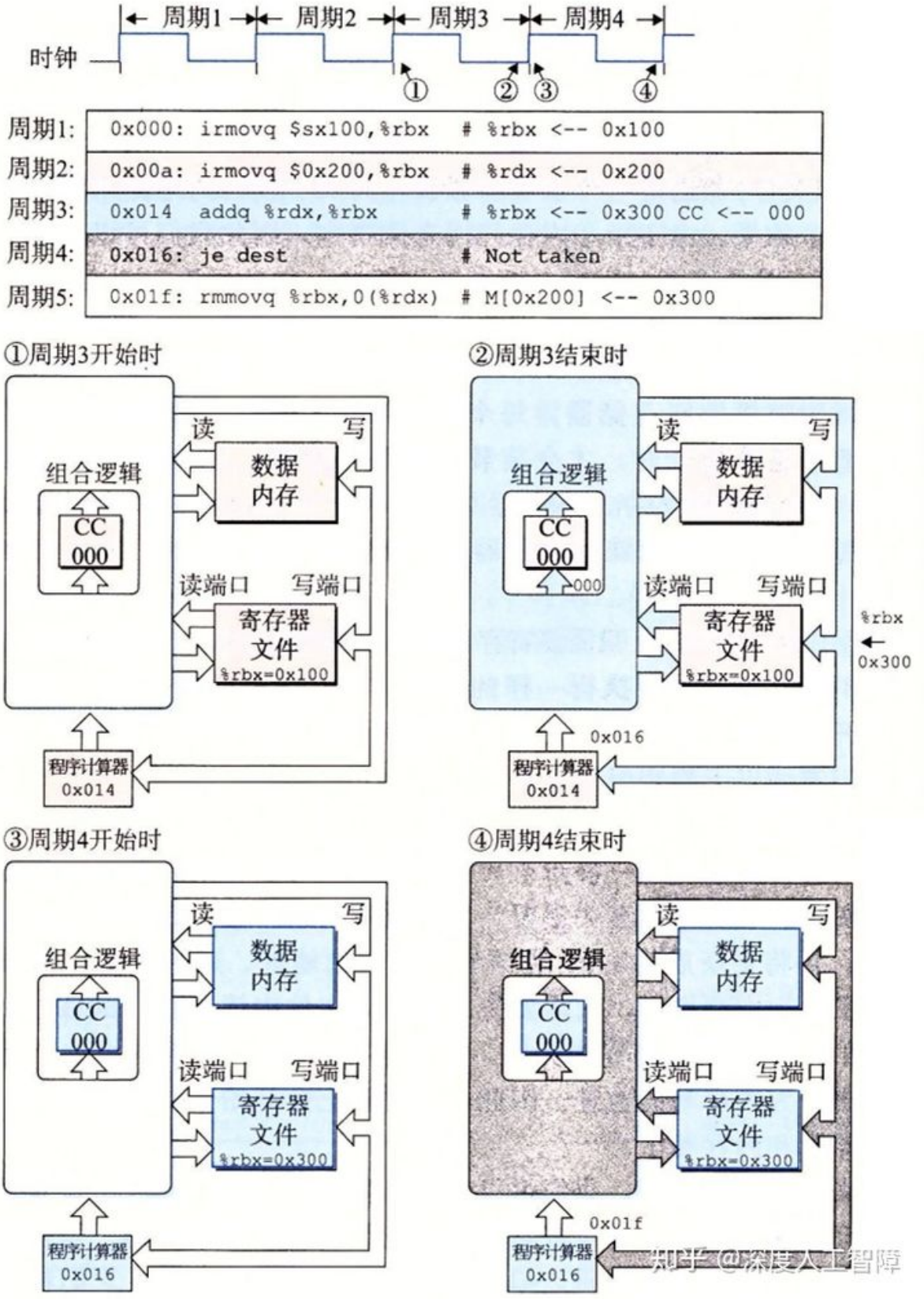


- 白色方框为时钟寄存器；蓝色方框为硬件单元，当做黑盒子而不关心细节设计；白色圆圈表示线路名字。
- 宽度为字长的数据使用粗线；宽度为字节或更窄的数据用细线；单个位的数据用虚线，主要表示控制值。
- 灰色圆角矩形表示控制逻辑块，能在不同硬件单元之间传递数据，以及操作这些硬件单元，使得对每个不同的指令执行指定的运算。是本章的重点，会给出对应的HCL表达式。

SEQ的实现包括组合逻辑和两种存储器：时钟寄存器（程序计数器和条件码寄存器）和随机访问存储器（寄存器文件、指令内存和数据内存）。我们知道组合逻辑和存储器的读取是没有时序的，只要输入一给定，输出就会发生对应的变化。但是存储器的写入是受到时钟的控制的，只有当时钟为高电位时，才会将值写入存储器中。

所以涉及到写数据的存储器（程序计数器、条件码寄存器、寄存器文件和数据内存）就需要对时序进行明确的控制，才能控制好指令各阶段的执行顺序。为了保证每条指令执行的结果能和上一节中介绍的顺序执行的结果相同，我们要保证指令的计算**不会回读**，即处理器不需要为了完成一条指令的执行而去读取由该指令更新的状态。因为该指令更新的状态是写入数据，需要经过一个时钟周期，如果该指令需要读取更新过的状态，就需要空出一个时钟周期。

我们通过寄存器和内存的时钟控制，由此设计了上一节中的指令执行阶段，这样能够保证即使所有状态同时更新，也能等价于顺序执行各个阶段，也保证了能够在一个周期中完成一条指令。



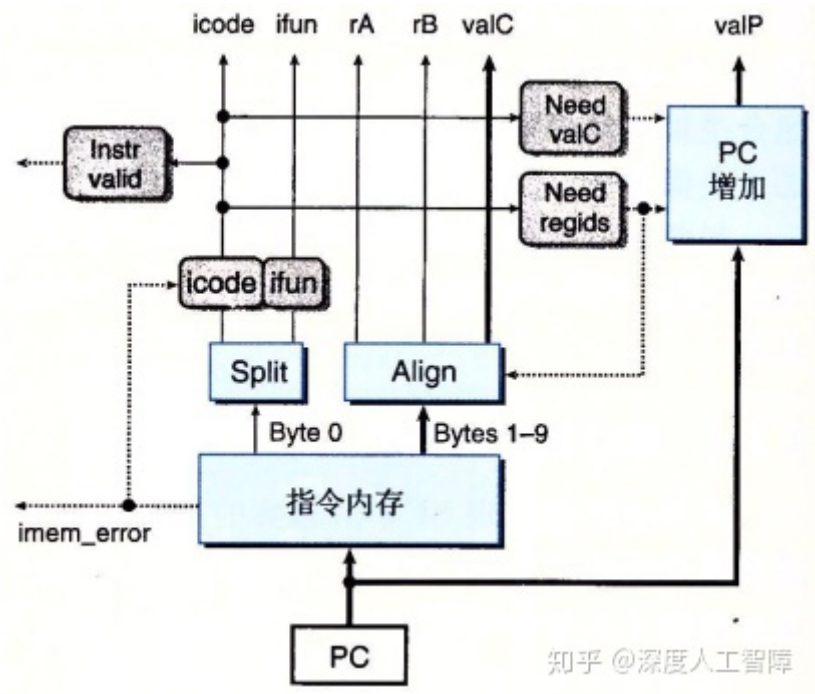
如上图所示，每次时钟从低电平变为高电平时，就会执行一条指令。开始执行`addq %rdx，%rbx`时，在时钟周期起点①处，会在寄存器和内存中写入上一条指令计算的结果，此时逻辑电路（白色部分）还没开始计算，到了时钟周期结尾②处，会执行得到该条指令的结果，并且更新程序计数器指向下一条指令，但是由于时钟还处于低电平，所以还未写入内存和寄存器中。当开始执行`je dest`时，在时钟周期起点③处，由于时钟变为了高电平，所以会将`addq %rdx，%rbx`计算的结果写入寄存器和内存中，此时就会指向`je dest`指令，但是逻辑电路还没还是计算，到了时钟周期结尾④处时，就会计算得到该条指令的结果，但是还没写入内存和寄存器中。

2.3 SEQ的HCL表达式

我们首先对指令进行编码

名称	值(十六进制)	含义
IHALT	0	halt 指令的代码
INOP	1	nop 指令的代码
IRRMOVQ	2	rrmovq 指令的代码
IIRMOVQ	3	irmovq 指令的代码
IRMMOVQ	4	rmmovq 指令的代码
IMRMOVQ	5	mrmovq 指令的代码
IOPL	6	整数运算指令的代码
IJXX	7	跳转指令的代码
ICALL	8	call 指令的代码
IRET	9	ret 指令的代码
IPUSHQ	A	pushq 指令的代码
IPOPQ	B	popq 指令的代码
FNONE	0	默认功能码
RRSP	4	% rsp 的寄存器 ID
RNONE	F	表明没有寄存器文件访问
ALUADD	0	加法运算的功能
SAOK	1	①正常操作状态码
SADR	2	②地址异常状态码
SINS	3	③非法指令异常状态码
SHLT	4	④halt 状态码

2.3.1 取指阶段



该部分访问**内存硬件单元。**首先以PC作为第一个字节的地址，一次从内存中读取10个字节。灰色部分是我们需要确定的HCL表达式

- `icode`为第一字节的高4位，当指令地址越界时，指令内存会返回`imem_error`信号，此时直接将其表示为`nop`指令，否则获得高4位值

```
word icode = [
  imem_error : INOP;
  1          : imem_icode;
];
```

- `ifun`为第一字节的低4位，当出现`imem_error`信号时，会使用默认功能码，否则获得低4位值

```
word ifun = [
  imem_error : FNONE;
  1          : imem_ifun;
];
```

- `instr_valid`表示是否为合法指令

```
bool instr_valid = icode in {
  INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ, IOPQ, IJXX, ICALL, IRET, IPUSHQ,
IPOPQ
};
```

- `need_regids`表示该指令否包含寄存器指示符字节，如果指令不含有寄存器指示符字节，则会将其赋值为`0xFF`。

```
bool need_regids = icode in {
  IRRMOVQ, IOPQ, IPUSHQ, IPOPQ, IIRMOVQ, IRMMOVQ, IMRMOVQ
};
```

- `need_valC`表示该指令是否含有常数字节

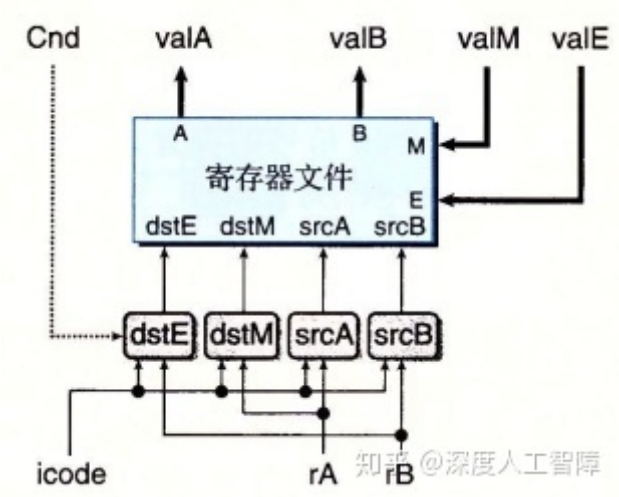
```
bool need_valC = icode in {
  IIRMOVQ, IRMMOVQ, IMRMOVQ, IJXX, ICALL
};
```

PC增加器会根据PC值、`need_valC`和`need_regids`来确定`valP`值，则

```
valP = PC+1+need_regids+8*need_valC
```

2.3.2 译码和写回阶段

这两个阶段都涉及**寄存器文件**，会根据`icode`、条件信号`Cnd`、`rA`和`rB`来设置写入的目的和读取的源。



- 写入的目的`dstE`和`dstM`

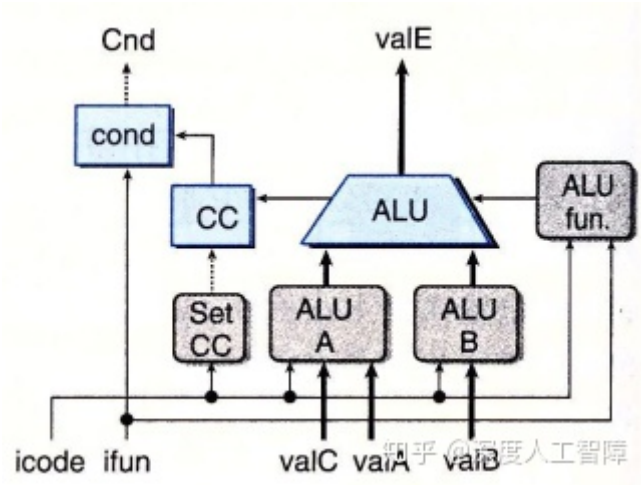
```
word dstE = [
  icode in {IRRMOVQ} && Cnd             : rB; #cmovXX指令，可以将其看成是rrmovq和条件信号Cnd
的组合
  icode in {IIRMOVQ, IOPQ}              : rB;
  icode in {IPUSHQ, IPOPQ, ICALL, IRET} : RRSP; #获取%rsp
  1                                     : RNONE;
];
word dstM = [
  icode in {IMRMOVQ, IPOPQ} : rA;
  1                         : RNONE;
];
```

- 读取的源`srcA`和`srcB`

```
word srcA = [
  icode in {IRRMOVQ, IRMMOVQ, IOPQ, IPUSHQ} : rA;
  icode in {IPOPQ, IRET}                    : RRSP;
  1                                         : RNONE;
];
word srcB = [
  icode in {IOPQ, IRMMOVQ, IMRMOVQ}     : rB;
  icode in {IPUSHQ, IPOPQ, ICALL, IRET} : RRSP;
  1                                     : RNONE;
];
```

2.3.3 执行阶段

该部分包括**ALU**。该部分逻辑主要根据`icode`来设置进入ALU进行计算的两个数字`aluA`和`aluB`，会根据`ifun`来设置需要ALU进行的计算。其中根据`ifun`和条件码计算得到的条件信息`Cnd`是通过cond硬件模块。



- 进入ALU进行计算的两个值`aluA`和`aluB`

```
word aluA = [
  icode in {IRRMOVQ, IOPQ}              : valA; #包含两个寄存器时，aluA为寄存器的值valA
  icode in {IIRMOVQ, IRMMOVQ, IMRMOVQ} : valC; #当出现立即数、偏移量时，aluA为常数值
  icode in {ICALL, IPUSHQ}              : -8; #入栈需要将栈顶地址下移8字节
  icode in {IRET, IPOPQ}                : 8; #出栈需要将栈顶地址上移8字节
];
word aluB = [
  icode in {IRMMOVQ, IMRMOVQ, IOPQ, ICALL, IPUSHQ, IRET, IPOPQ} : valB;
  icode in {IRRMOVQ, IIRMOVQ}                                   : 0;
];
```

- 设置ALU进行的函数`alufun`

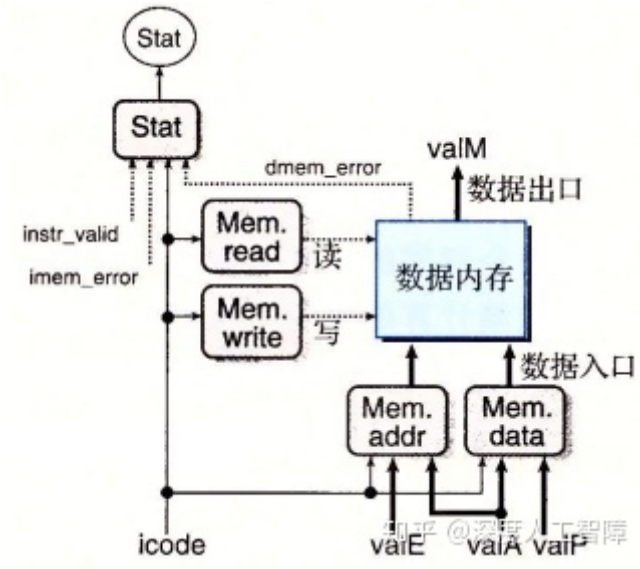
```
word alufun = [
  icode == IOPQ : ifun;
  1             : ALUADD;
];
```

- 获得是否设置条件码`set_cc`

```
bool set_cc = icode in {IOPQ};
```

2.3.4 访存阶段

这部分包含**数据内存**，设计从从数据内存读写程序数据。



- 确定是从内存中读取数据还是写入数据

```
bool mem_read = icode in {IMRMOVQ, IPOPQ, IRET};
bool mem_write = icode in {IRMMOVQ, IPUSHQ, ICALL};
```

- 获得内存地址`mem_addr`

```
mem_addr = [
  icode in {IRMMOVQ, IPUSHQ, ICALL, IMRMOVQ} : valE; #IPUSHQ和ICALL设计栈地址计算，IRMMOVQ和IMRMOVQ设计内存引用计算，所以都是ALU的计算结果
  icode in {IPOPQ, IRET}                     : valA; #这部分没涉及ALU计算
];
```

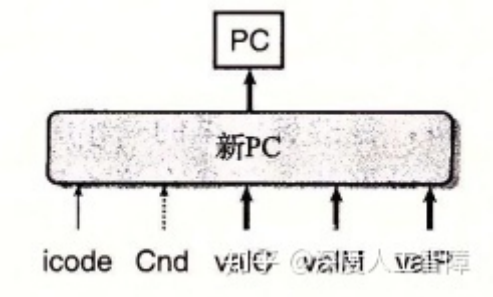
- 获得输入内存的数据`mem_data`

```
word mem_data = [
  icode in {IRMMOVQ, IPUSHQ} : valA; #从寄存器获得值
  icode == ICALL             : valP; #当调用函数时，会将返回地址写入内存中
  #默认不写入任何信息
];
```

- **确定程序状态**`Stat`

```
word Stat = [
  imem_error || dmem_error: ASDR;
  !instr_valid            : SINS;
  icode == IHALT          : SHLT;
  1                       : SAOK;
];
```

2.3.5 更新PC阶段




```
word new_pc = [
  icode == ICALL       : valC; #调用函数时，会直接将PC更新为目标函数的地址
  icode == IJXX && Cnd : valC; #当条件跳转指令满足时，会跳转到目标地址
  icode == IRET        : valM; #ret会从内存中读取返回地址，所以是valM
  1                    : valP; #默认为valP
];
```

2.4 SEQ性能

我们通过将指令执行过程划分成了若干个阶段，使得我们能通过统一框架来描述各个指令执行的过程，也能进一步减少需要的硬件。但是由于每次时钟变为高电平时需要写入数据，使得需要在一个时间周期内完成所有步骤，所以我们要求时钟周期特别慢。

比如执行`ret`时，当前PC指向`ret`指令的地址，当时钟变为高电平时，我们需要在下一次时钟变为高电平之前，完成：两次从寄存器文件读取`%rsp`内容，通过ALU计算`%rsp`上移8字节的地址，根据`%rsp`从内存中获得返回地址，然后将新的`%rsp`值写回寄存器文件中（此时由于时钟还是低电平，所以还没有真实写入，只是设置为了值）。由此能够保证在下一个时钟变为高电平时，能够把正确的`%rsp`值写回寄存器文件中。

而且我们可以发现，指令执行的不同阶段是在处理器的不同硬件部分，所以完全可以让不同指令同时运行，只要求他们处于不同阶段，这也是下一章中流水线的主要思想。

想要更进一步了解的话，可以看ECE 741课程。