Datawhale

Datawhale 开源社区

Datawhale open source community

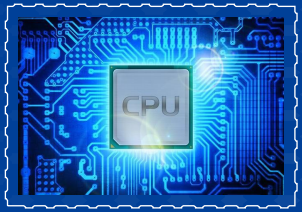
深入理解计算机系统（8）

Computer Systems A Programmer's Perspective CSAPP

李岳昆、易远哲

realyurk@gmail.com、yuanzhe.yi@outlook.com

2021 年 11 月 18 日



第 I 部分

处理器体系结构-I

Y86-64 指令集体系结构

Y86-64 的顺序实现

为什么要学习处理器体系结构

Datawhale

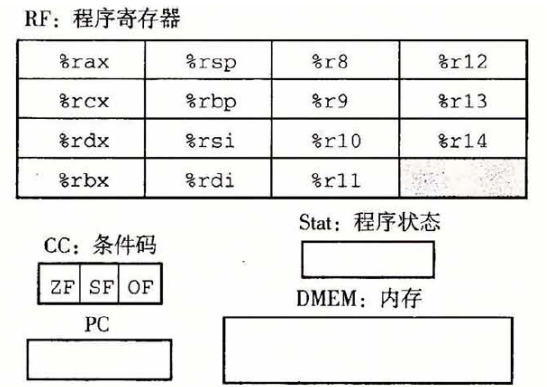
虽然，我们不太可能真的去自己设计一款处理器，但处理器作为一个结构和规则相对简单但又能完成复杂工作的结构，学习它的原理无论是对科学家还是工程师来说都是十分重要的。同时，理解处理器是如何工作的能够帮助我们理解整个计算机系统的工作方式。

在该部分内容中，我们将定义一个简单的指令集，称为“Y86-64”。相对于“x86-64”指令集，Y86 做了部分精简以便于我们学习。

程序员可见的状态

Datawhale

在这里我们说的程序员不仅指真的程序员，还包括产生机器级代码的编译器。



程序员可见的状态

在上图中，我们定义了 15 各 64 位的程序寄存器。相比于 x86，省略了%r15 以简化指令的编码。和 x86 一样，寄存器%rsp 被定义为栈指针，其他 14 个寄存器无固定含义。

此外，我们只保留了 x86 中的零标志（ZF）、符号标志（SF）和溢出标志（OF）。程序计数器 PC 用来存放当前正在执行的指令的地址，状态码 Stat 用来表示程序的执行状态。

为了简化 x86-64 的指令集，我们将 x86-64 中的 movq 指令分成了 rrmovq、irmovq、rmmovq 和 mrmovq 四种。movq 前面的两个祖母分别表示了源和目的的格式，例如 irmovq 的源操作数是立即数（**I**mmediate），目的操作数是寄存器（**R**egister）。

指令编码

Datawhale

接下来我们对上面声明的数据传送指令进行编码

指令编码

上图中每条指令的第一个字节表明指令的类型。这个字节分为两部分，每部分占 4 个比特位，高四位表示指令代码，低四位表示指令功能。

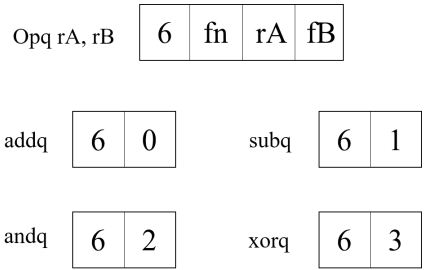
为了方便在指令编码中使用寄存器，我们需要给寄存器进行编号：

编号	寄存器	编号	寄存器
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

再看之前定义的 irmovq，该指令的指令代码为 3，指令功能为 0. 由于源操作数是立即数，因此用 0xF 进行填充。

指令编码

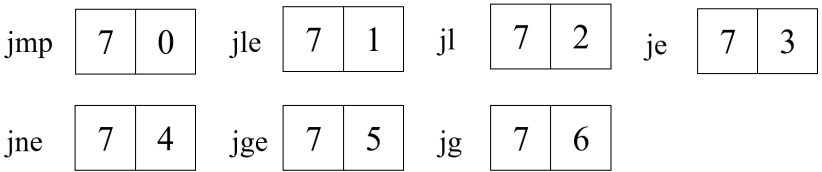
对于整数的操作，我们定义了四条整数操作指令。和 x86-64 不同，我们定义的指令只允许对寄存器中的数据进行操作。



以上四条指令属于同一类型，因此指令代码是一样的，不同的是指令的功能。

指令编码

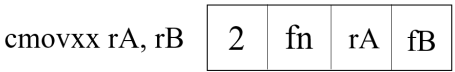
对于跳转，我们定义了 7 条跳转指令，它们的跳转条件和 x86-64 中的跳转指令相同，都是根据条件码的某种组合来判断是否进行跳转:

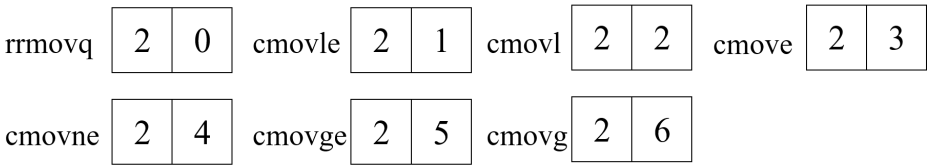


数据编码

Datawhale

我们定义了 6 条传送指令，它们与数据传送指令 rrmovq 有相同的指令格式，只有条件码满足条件时才会更新目的寄存器的值。

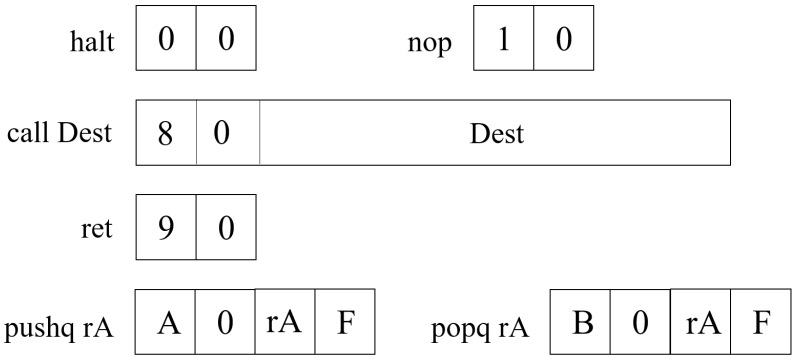




指令编码

Datawhale

接下来是一些其他的系统操作指令：



指令编码

Datawhale

在上面的操作指令中：

1. halt 指令停止指令的执行，执行该指令会导致处理器停止，并将状态码设为 HLT。
2. nop 指令表示一个空操作。
3. call 和 ret 指令分别实现函数的调用和返回。
4. push 和 pop 指令分别实现入栈和出栈的操作。

指令编码

Datawhale

有了如上的指令编码，我们就可以将 Y86-64 的汇编代码翻译成二进制表示了，例如：

rmmovq %rsp, 0x123456789abcd(%rdx)

根据前面 rmmovq 指令的编码定义，该指令二进制表示中的第一个字节为 0x40；

指令操作数部分的寄存器%rsp 对应的寄存器编号为 0x4，基址寄存器 rdx 对应的编号为 0x2，因此该指令二进制表示中的第二个字节为 0x42；

指令编码中偏移量占 8 个字节，因此我们需要在该偏移量前面补 0 来凑齐 8 个字节。又由于我们采用小端法存储，因此还要对偏移量进行字节反序操作。最终我们得到该指令的二进制表示如下：

40 42 cd ab 89 67 45 23 01 00

最后，我们看看 Y86-64 中的状态码：

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

Y86-64 中程序的底层表示

Datawhale

我们通过一段代码来展示 Y86-64 中 C 语言代码的汇编翻译

```c
long sum(long *start, long count) {
    long sum = 0;
    while (count) {
        sum += *start;
        start ++;
        count --;
    }
    return sum;
}
```

```
sum:
    irmovq $8, %r8
    irmovq $1, %r9
    xorq %rax, %rax
    andq %rsi, %rsi
    jmp test
loop:
    mrmovq (%rdi), %r10
    addq %r10, %rax
```

这段 C 语言代码的作用是计算一个数组的元素之和，其中指针 start 指向数组的起始位置，count 用于表示数组的长度。

可以看到，除数据传送指令外，Y86-64 的指令与 x86-64 的指令相差不大。

Y86-64 中程序的底层表示

Datawhale

我们再看看上面的汇编代码是如何翻译成二进制指令的:

```
sum:                                0x056:
    irmovq $8, %r8                  0x056: 30f80800000000000000
    irmovq $1, %r9                  0x060: 30f90100000000000000
    xorq %rax, %rax                 0x06a: 6300
    andq %rsi, %rsi                 0x06c: 6266
    jmp test                        0x06e: 708700000000000000
loop:                               0x077:
    mrmovq (%rdi), %r10             0x077: 50a70000000000000000
    addq %r10, %rax                 0x081: 60a0
```

在本节，我们的主要内容就是设计一个能够执行上面二进制指令的 Y86-64 处理器。

阶段

Datawhale

处理器在执行单条指令时，往往也包含着很多操作。我们将 Y86-64 执行指令的过程组织成如下 6 个阶段：

1. 取址：处理器执行所有的指令都需要取址。在 Y86-64 指令系统中，指令的长度不是固定的，因此取址阶段需要根据指令代码判断指令是否含有寄存器指示符、是否含有常数来计算当前的指令长度。
2. 译码：在译码阶段中，处理器从寄存器文件中读取数据。寄存器文件有两个读端口，可以支持同时进行两个读操作。
3. 执行：指令被正式执行的阶段。在该阶段中，算术逻辑单元（ALU）主要执行三类操作：执行算术逻辑运算、计算内存引用的有效地址、针对 push 和 pop 指令的运算。
4. 访存：顾名思义，对内存进行读写操作的阶段。
5. 写回：将执行结果写回到寄存器文件中。
6. 更新：将 PC 更新为下一条指令的地址。

subq 指令的各个阶段

Datawhale

subq %rdx, %rbx

6	1	2	3

我们来看看上面这条 subq 指令各阶段包含的内容：

阶段	subq %rdx, %rbx
取址	icode=6, ifun=1 rA=2, rB=3　valP=PC+2
译码	valA = R[%rdx] valB = R[%rbx]
执行	valE = valB - valA Set CC
访存	
写回	R[%rbx] = valE
更新	PC = valP

subq 指令的各个阶段

Datawhale

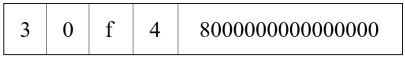
如上图所示，

1. 取址阶段，根据指令代码来计算指令长度。
2. 译码阶段，根据寄存器指示符来读取寄存器的值。
3. 执行阶段，ALU 根据译码阶段读取到的操作数以及指令来执行具体的运算，并设置条件码寄存器。
4. 访存阶段，由于减法指令不需要读写内存，因此该阶段无操作。
5. 写回阶段，将 ALU 的运算结果写回寄存器。
6. 更新阶段，更新程序计数器。

irmovq 指令的各个阶段

Datawhale

irmovq $8, %rsp



上图这条 irmovq 指令将一个立即数传送给寄存器，它的各阶段内容如下：

阶段	irmovq $8, %rsp
取址	icode=3, ifun=0 rA=0xF, rB=4, valC=8 valP=PC+10
译码	
执行	valE = 0 + 8
访存	
写回	R[%rsp] = 8
更新	PC = valP

irmovq 指令的各个阶段

Datawhale

如前图所示

1. 取址阶段，该指令既含有寄存器指示符字节，也含有常数字段。
2. 译码阶段，该指令不需要从寄存器中读取数据，译码阶段无操作。
3. 执行阶段，虽然该指令仅仅传送数据，看似不需要 ALU，但由于 ALU 的输出端与寄存器的写入端相连，数据的传送还是需要经过 ALU，因此该指令将立即数加 0.
4. 访存阶段，该指令不需要读写内存，因此该阶段无操作。
5. 写回阶段，将 ALU 的运算结果写回寄存器。
6. 更新阶段，更新程序计数器。

rmmovq 指令的各个阶段

Datawhale

rmmovq %rsp, 100(%rbx)	4	0	4	3	6400000000000000

上图这条 rmmovq 指令各个阶段的内容如下：

阶段	rmmovq %rsp, 100(%rbx)
取址	icode=4, ifun=0 rA=4, rB=3, valC=100 valP=PC+10
译码	valA = R[%rsp] valB = R[%rbx]
执行	valE = valB + valC
访存	M[valE] = valA
写回	
更新	PC = valP

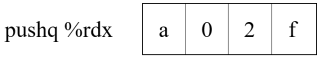
rrmovq 指令的各个阶段

Datawhale

如前图所示

1. 取址阶段，该指令既含有寄存器指示符字节，也含有常数字段。
2. 译码阶段，从寄存器中读取数据。
3. 执行阶段，ALU 根据偏移量和基址寄存器来计算访存地址。
4. 访存阶段，将寄存器 rsp 的数值写入内存中。
5. 写回阶段，由于内存地址由执行阶段得出并写入寄存器，因此写回阶段不需要进行操作。
6. 更新阶段，更新程序计数器。

pushq 指令的各个阶段



上图这条 pushq 指令各阶段的内容如下：

阶段	pushq %rdx
取址	icode=a, ifun=0 rA=2, rB=0xF, valP=PC+2
译码	valA = R[%rdx] valB = R[%rsp]
执行	valE = valB + (-8)
访存	M[valE] = valA
写回	R[%rsp] = valE
更新	PC = valP

pushq 指令的各个阶段

Datawhale

1. 取址阶段，该指令含有寄存器指示符，不含常数，因此指令长度为 2 字节。
2. 译码阶段，由于 pushq 指令要将寄存器 rdx 的值保存到栈上，因此该指令不仅需要读取寄存器 rdx 的值，还需要读取寄存器 rsp 的值。
3. 执行阶段，ALU 计算内存地址。
4. 访存阶段，将寄存器 rdx 的值写到栈上。
5. 写回阶段，由于寄存器 rsp 指向的内存地址发生了变化，因此更新寄存器 rsp 的值。
6. 更新阶段，更新程序计数器。

je 指令的各个阶段

Datawhale

je 0x040	7	3	4000000000000000

上图这条 je 指令各阶段的内容如下：

阶段	je 0x040
取址	icode=7, ifun=3 valC = 0x040 valP = PC + 9
译码	
执行	cnd = Cond(CC, ifun)
访存	
写回	
更新	PC = Cnd ? 0x040 : valP

je 指令的各个阶段

Datawhale

1. 取址阶段，该指令含有常数字段，不含寄存器指示符字节，因此指令长度为 9 字节。
2. 译码阶段，不需要读取寄存器，无操作。
3. 执行阶段，标号为 Cond 的硬件单元根据条件码和指令功能来判断是否执行跳转，该模块产生一个信号 Cnd，若 Cnd = 1，则执行跳转；Cnd = 0 则不执行跳转。
4. 访存阶段，无操作。
5. 写回阶段，无操作。
6. 更新阶段，若 Cnd = 1，将 PC 的值设为 0x040；若 Cnd = 0，则将 PC 的值设为当前值加 9。