CH4-Processor Architecture

4.1 Introducing to Y86

Y86-64 Processor State

- **program registers**

  Each register has 4-bit ID

%rax	0	%rsi	6	%r8	8	%r12	C
%rcx	1	%rdi	7	%r9	9	%r13	D
%rdx	2	%rsp	4	%r10	A	%r14	E
%rbx	3	%rbp	5	%r11	B	None	F

  - Same encoding as in x86-64, except %r15

  其中一个为空，这样就可以使用4个bits来表示寄存器
- **program counter**(也就是PC)
- **condition codes**(也就是CC:OF,ZF,SF)
- **status code**

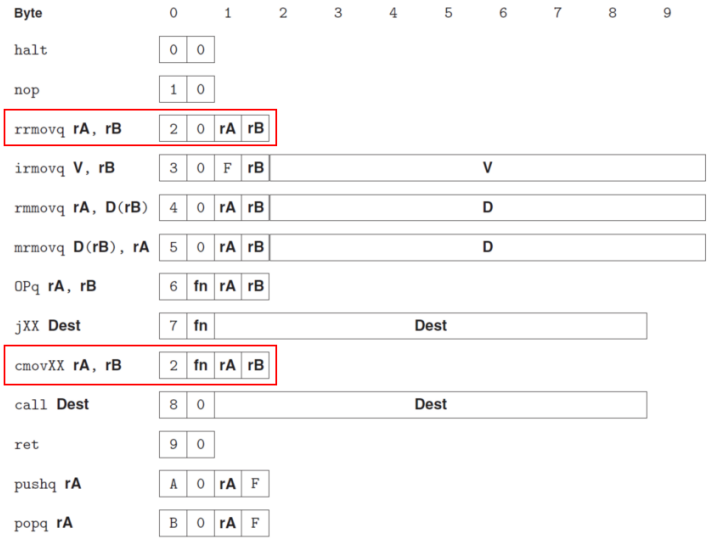
  程序正常执行或发生事件

  分为：
  - 1(AOK):正常执行
  - 2(HLT):halt指令
  - 3(ADR):非法地址（取指或内存）
  - 4(INS):非法指令
- **memory**

Y86-64 Instructions

一共有12条变长指令，通过其编号就可以判断其长度

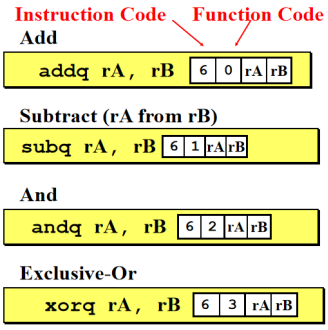
icode+ifun+rA+rB+valC



其中cmovXX是rrmovq的子集

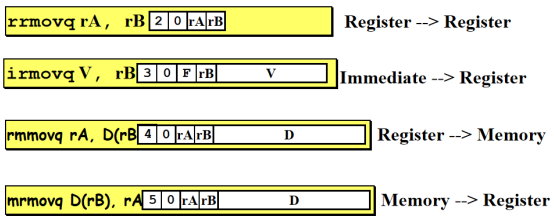
- **Arithmetic and Logical Operations**

  这会设置CC

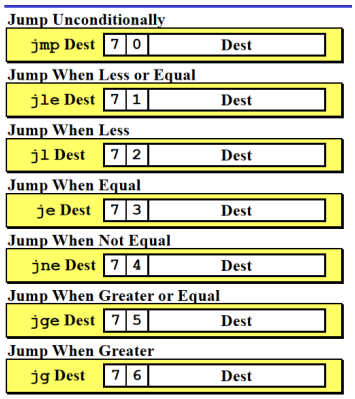
  

- **Move Operations**

  注意到这些指令都要通过寄存器作为媒介，所以会有一些x86没有的指令

- **Conditional Move Operations**
- **Jump Instructions**



- **Stack Operations**

  y86的栈基本类似于x86

  一些trick:

  pushq %rsp -> save old %rsp

  popq %rsp -> movq (%rsp) %rsp
- **Subroutine Call and Return**
- **Miscellaneous Instructions**

  nop不做任何事情

  halt中止执行指令

Y86-64 Programs

```
1   # Execution begins at address 0
2       .pos 0                          # assembler directives,告诉汇编器从地址0产生代码
3       irmovq      stack, %rsp                 # Set up stack pointer,可以认为这里的stack类似于宏
4       call                main                # Execute main program
5       halt                            # Terminate program
6
7   # Array of 4 elements
8       .align 8                        # 同为伪指令，指出8字节对齐
9   array:                              # 声明一个数组
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,相当于置零
27      andq        %rsi,%rsi           # Set CC,判断count是否为零，同时不改变其值
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:                             # 指明栈从0x200开始
```

通过观察每条指令执行后的状态码来debug

Assembling Y86-64 Program

```
unix> yas eg.ys
```

小端序

```
                        | # Execution begins at address 0
0x000:                  |   .pos 0
0x000: 30f400200000000000 |   irmovq stack, %rsp      # Set up stack pointer
0x00a: 8038000000000000 |   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: 30f718000000000000 |   irmovq array,%rdi
0x042: 30f604000000000000 |   irmovq $4,%rsi
0x04c: 8056000000000000 |   call sum                # sum(array, 4)
0x055: 90               |   ret
```

Simulating Y86-64 Program

```
unix> yis eg.yo
```

ISA(Instruction Set Architecture)

ISA提供了软件与硬件之间的概念抽象层

CISC vs.RISC

RISC:ARM,CISC:X86

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。

现在的ISA综合了CISC和RISC的优点

4.2 Logical Design & HCL

Combinational Circuits

- **Bit Equal**
- **Bit-level Multiplexor(Bit MUX)**

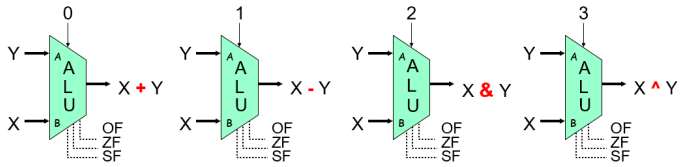
HCL Representation

使用case expression表示,由多个select:expr组合而成，输出为第一个select为1的expr

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

这里的1可以认为是default

- **Arithmetic Logic Unit**

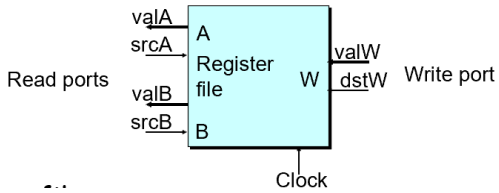


Storage(Sequential Circuits)

Clocked Registers

在clock呈现上升沿时，才根据input改变output

- **Register File**

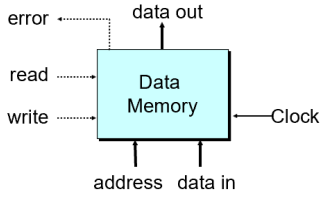


可同时支持读两个程序寄存器的值，同时更新第三个寄存器的状态

  - 读：根据src的寄存器ID，一段延迟后输出对应的val
  - 写：输入val和dst，在clock上升沿时写入

- **Memory**

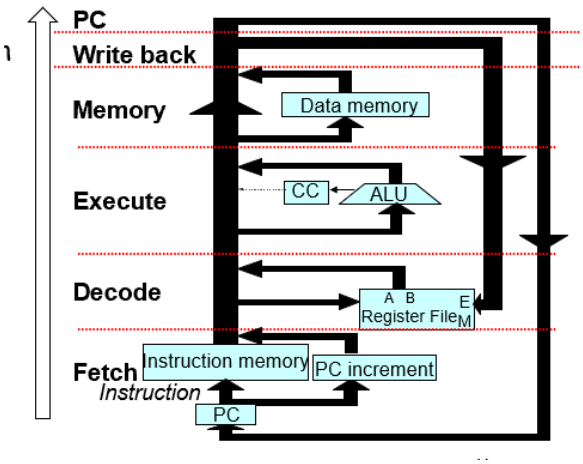
数据处理器



类似的读与写，设置write为0或1，非法地址时，error设置为1

4.3 Sequential CPU Implementation

Instruction Execution Stages



- **Fetch**
  读取指令
  因为PC是clock register，所以在上升沿的时候增加PC
- **Decode**
  读取register，使用register file
- **Execute**
  执行指令，ALU用于算术/逻辑单元，可能修改或使用CC
- **Memory**
  读写内存
- **Write Back**
  对寄存器进行写操作
- **PC**
  更新PC

当出现异常时（halt/非法指令/非法地址），processor loop停止

Computation Steps

		OPq rA, rB	call Dest
Fetch	icode,ifun	icode:ifun ← M_1[PC]	icode:ifun ← M_1[PC]
	rA,rB	rA:rB ← M_1[PC+1]	
	valC		valC ← M_8[PC+1]
	valP	valP ← PC+2	valP ← PC+9
Decode	valA, srcA	valA ← R[rA]	
	valB, srcB	valB ← R[rB]	valB ← R[%rsp]
Execute	valE,aluA,aluB	valE ← valB OP valA	valE ← valB + –8
	Cond code	Set CC	
Memory	valM,addr,data		M_8[valE] ← valP
Write Back	valE,dstE	R[rB] ← valE	R[%rsp] ← valE
	valM,dstM		
PC update	PC, newPC	PC ← valP	PC ← valC

上述步骤同时在上升沿发生，status同时更新

SEQ CPU Implementation

后面再补,拉下共同点就可以

4.4 Principles of Pipeline