movq Operand Combinations

	Source	Dest	Src,Dest	C code
movq	*Imm*	*Reg*	`movq $0x4,%rax`	`temp = 0x4;`
		Mem	`movq $-147,(%rax)`	`*p = -147;`
	Reg	*Reg*	`movq %rax,%rdx`	`temp2 = temp1;`
		Mem	`movq %rax,(%rdx)`	`*p = temp;`
	Mem	*Reg*	`movq (%rax),%rdx`	`temp = *p;`

Cannot do memory-memory transfer with a single instruction!

Type	Form	Operand value	Name
Immediate	$\$Imm$	Imm	Immediate
Register	r_a	$R[r_a]$	Register
Memory	Imm	$M[Imm]$	Absolute
Memory	(r_a)	$M[R[r_a]]$	Indirect
Memory	$Imm(r_b)$	$M[Imm + R[r_b]]$	Base + displacement
Memory	(r_b,r_i)	$M[R[r_b] + R[r_i]]$	Indexed
Memory	$Imm(r_b,r_i)$	$M[Imm + R[r_b] + R[r_i]]$	Indexed
Memory	$(,r_i,s)$	$M[R[r_i] \cdot s]$	Scaled indexed
Memory	$Imm(,r_i,s)$	$M[Imm + R[r_i] \cdot s]$	Scaled indexed
Memory	(r_b,r_i,s)	$M[R[r_b] + R[r_i] \cdot s]$	Scaled indexed
Memory	$Imm(r_b,r_i,s)$	$M[Imm + R[r_b] + R[r_i] \cdot s]$	Scaled indexed

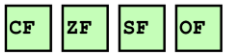
Instruction		Effect	Description
`leaq`	S, D	$D \leftarrow \&S$	Load effective address
INC	D	$D \leftarrow D+1$	Increment
DEC	D	$D \leftarrow D-1$	Decrement
NEG	D	$D \leftarrow -D$	Negate
NOT	D	$D \leftarrow \sim D$	Complement
ADD	S, D	$D \leftarrow D + S$	Add
SUB	S, D	$D \leftarrow D - S$	Subtract
IMUL	S, D	$D \leftarrow D * S$	Multiply
XOR	S, D	$D \leftarrow D \hat{} S$	Exclusive-or
OR	S, D	$D \leftarrow D \mid S$	Or
AND	S, D	$D \leftarrow D \& S$	And
SAL	k, D	$D \leftarrow D << k$	Left shift
SHL	k, D	$D \leftarrow D << k$	Left shift (same as SAL)
SAR	k, D	$D \leftarrow D >>_A k$	Arithmetic right shift
SHR	k, D	$D \leftarrow D >>_L k$	Logical right shift

63	31	15	7	0
%rax	%eax	%ax	%al	
%rbx	%ebx	%bx	%bl	
%rcx	%ecx	%cx	%cl	
%rdx	%edx	%dx	%dl	
%rsi	%esi	%si	%sil	
%rdi	%edi	%di	%dil	
%rbp	%ebp	%bp	%bpl	
%rsp	%esp	%sp	%spl	

63	31	15	7	0
%r8	%r8d	%r8w	%r8b	
%r9	%r9d	%r9w	%r9b	
%r10	%r10d	%r10w	%r10b	
%r11	%r11d	%r11w	%r11b	
%r12	%r12d	%r12w	%r12b	
%r13	%r13d	%r13w	%r13b	
%r14	%r14d	%r14w	%r14b	
%r15	%r15d	%r15w	%r15b	

B = byte = 1 bytes
W = word = 2 bytes
L = long = 4 bytes
Q = quadword = 8 bytes

- **Single bit registers**



  - **CF** Carry Flag (for *unsigned*) **SF** Sign Flag (for *signed*)
  - **ZF** Zero Flag **OF** Overflow Flag (for *signed*)
- **Implicitly** set (as side effect) by **arithmetic** operations

  Example: **addq** *Src,Dest* ↔ **t = a+b**

  CF set if carry out from most significant bit (unsigned overflow)

  ZF set if **t == 0**

  SF set if **t < 0** (as signed)

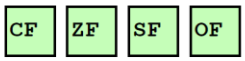
  OF set if two's-complement (signed) overflow

```
(a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)
```

- Note: **leaq** is not considered as arithmetic instruction

- **Explicit Setting by Compare Instruction**
  - **cmpq** *Src1, Src2*
  - **cmpq b,a** like computing **a-b** without setting destination

  - **CF set** if carry out from most significant bit (used for unsigned comparisons)
  - **ZF set** if **a == b**
  - **SF set** if **(a-b) < 0** (as signed)



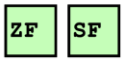
  - **OF set** if two's-complement (signed) overflow

```
    (a>0 && b<0 && (a-b)<0)
||  (a<0 && b>0 && (a-b)>0)
```

- **Explicit Setting by Test instruction**
  - **testq** *Src1, Src2*
    - **testq b,a** like computing **a&b** without setting destination

  - Sets condition codes based on value of *Src1* & *Src2*
  - Useful to have one of the operands be a mask

  - **ZF set** when **a&b == 0**
  - **SF set** when **a&b < 0**



SetX	Condition	Description
sete	ZF	Equal / Zero
setne	~ZF	Not Equal / Not Zero
sets	SF	Negative
setns	~SF	Nonnegative
setg	~(SF^OF)&~ZF	Greater (Signed)
setge	~(SF^OF)	Greater or Equal (Signed)
setl	(SF^OF)	Less (Signed)
setle	(SF^OF)|ZF	Less or Equal (Signed)
seta	~CF&~ZF	Above (unsigned)
setb	CF	Below (unsigned)

type is Unsigned

- **SetX Instructions:**
  - Set single byte based on combination of condition codes
- **One of addressable byte registers**
  - Does not alter remaining bytes
  - Typically use movzbl to finish job
    - 32-bit instructions also set upper 32 bits to 0

```
int gt (long x, long y)
{
  return x > y;
}
```

Register	Use(s)
%rdi	Argument **x**
%rsi	Argument **y**
%rax	Return value

```
cmpq   %rsi, %rdi   # Compare x:y
setg   %al          # Set when >
movzbl %al, %eax    # Zero rest of %rax
ret
```

jX	Condition	Description
jmp	1	Unconditional
je	ZF	Equal / Zero
jne	~ZF	Not Equal / Not Zero
js	SF	Negative
jns	~SF	Nonnegative
jg	~(SF^OF)&~ZF	Greater (Signed)
jge	~(SF^OF)	Greater or Equal (Signed)
jl	(SF^OF)	Less (Signed)
jle	(SF^OF)|ZF	Less or Equal (Signed)
ja	~CF&~ZF	Above (unsigned)
jb	CF	Below (unsigned)

Conditional Move Example

```
long absdiff
  (long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	Return value

```
absdiff:
   movq    %rdi, %rax  # x
   subq    %rsi, %rax  # result = x-y
   movq    %rsi, %rdx
   subq    %rdi, %rdx  # eval = y-x
   cmpq    %rsi, %rdi  # x:y
   cmovle  %rdx, %rax  # if <=, result = eval
   ret
```

leaq 7(%rdi), %rax ==> t = x + 7, t in %rax

testq %rdi, %rdi ==> x & x, set conditional codes

cmovns %rdi, %rax ==> if (x&x >= 0), t = x

sarq $3, %rax ==> t = t >> 3

一种得到准确的负数除法结果的方法

（直接算术右移，所得商（整数）比正确的永远小 1，通过加 7 使其进入上一组“轮回”可以使其商增加 1，同时可以辨别正数）

```
long fun(long a, long b) {
    long result = b;
    while (b>0) {
        result = result * a;
        b = b - a;
    }
    return result;
}
```

GOTO Version (do-while)

```
long result = b;
if (!(b>0)) goto DONE;
LOOP:
    result = result * a;
    b = b - a;
TEST:
    if (b>0) goto LOOP;
DONE:
    return result;
```

GOTO Version (jump-to-middle)

```
long result = b;
goto TEST;
LOOP:
    result = result * a;
    b = b - a;
TEST:
    if (b>0) goto LOOP;
DONE:
    return result;
```

GOTO Version (do-while)

```
long result = b;
if (!(b>0)) goto DONE;
LOOP:
    result = result * a;
    b = b - a;
TEST:
    if (b>0) goto LOOP;
DONE:
    return result;
```

```
Func:
    movq    %rsi, %rax      # result = b
    cmpq    $0, %rsi        # compare b and 0
    jng     DONE            # if (!(b>0)) goto DONE
LOOP:
    imulq   %rdi, %rax      # result = result * a
    subq    %rdi, %rsi      # b = b - a
TEST:
    cmpq    $0, %rsi        # compare b and 0
    jg      LOOP            # goto LOOP
DONE:
    ret
```

Variable	register
a	%rdi
b	%rsi
result	%rax

```
fun:
        movq    %rsi, %rax      # result = b
L2:
        cmpq    $0, %rsi        # compare b and 0
        jg      L3              # if (b>0) goto L3
        ret                     # return result
L3:
        imulq   %rdi, %rax      # result = result * a
        subq    %rdi, %rsi      # b = b - a
        jmp     L2              # goto L2
```

```
long fun(long a, long b) {
    long result = b;
    while (b>0) {
        result = result * a;
        b = b - a;
    }
    return result;
}
```

Variable	register
a	%rdi
b	%rsi
result	%rax

An example of while loop

“For” Loop → While Loop

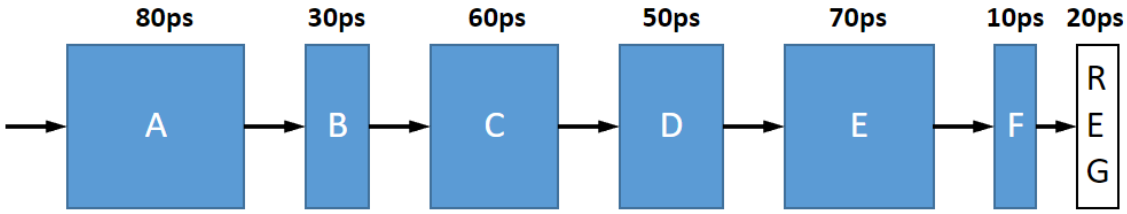
For Version

```
for (Init; Test; Update )
    Body
```

While Version

```
Init;
while (Test ) {
    Body
    Update;
}
```

Instruction	Description
CMOVA *r16, r/m16*	Move if above (CF=0 and ZF=0)
CMOVA *r32, r/m32*	Move if above (CF=0 and ZF=0)
CMOVAE *r16, r/m16*	Move if above or equal (CF=0)
CMOVAE *r32, r/m32*	Move if above or equal (CF=0)
CMOVB *r16, r/m16*	Move if below (CF=1)
CMOVB *r32, r/m32*	Move if below (CF=1)
CMOVBE *r16, r/m16*	Move if below or equal (CF=1 or ZF=1)
CMOVBE *r32, r/m32*	Move if below or equal (CF=1 or ZF=1)
CMOVC *r16, r/m16*	Move if carry (CF=1)
CMOVC *r32, r/m32*	Move if carry (CF=1)
CMOVE *r16, r/m16*	Move if equal (ZF=1)
CMOVE *r32, r/m32*	Move if equal (ZF=1)
CMOVG *r16, r/m16*	Move if greater (ZF=0 and SF=OF)
CMOVG *r32, r/m32*	Move if greater (ZF=0 and SF=OF)
CMOVGE *r16, r/m16*	Move if greater or equal (SF=OF)
CMOVGE *r32, r/m32*	Move if greater or equal (SF=OF)
CMOVL *r16, r/m16*	Move if less (SF<>OF)
CMOVL *r32, r/m32*	Move if less (SF<>OF)
CMOVLE *r16, r/m16*	Move if less or equal (ZF=1 or SF<>OF)
CMOVLE *r32, r/m32*	Move if less or equal (ZF=1 or SF<>OF)
CMOVNA *r16, r/m16*	Move if not above (CF=1 or ZF=1)
CMOVNA *r32, r/m32*	Move if not above (CF=1 or ZF=1)
CMOVNAE *r16, r/m16*	Move if not above or equal (CF=1)
CMOVNAE *r32, r/m32*	Move if not above or equal (CF=1)
CMOVNB *r16, r/m16*	Move if not below (CF=0)
CMOVNB *r32, r/m32*	Move if not below (CF=0)
CMOVNBE *r16, r/m16*	Move if not below or equal (CF=0 and ZF=0)
CMOVNBE *r32, r/m32*	Move if not below or equal (CF=0 and ZF=0)
CMOVNC *r16, r/m16*	Move if not carry (CF=0)
CMOVNC *r32, r/m32*	Move if not carry (CF=0)
CMOVNE *r16, r/m16*	Move if not equal (ZF=0)
CMOVNE *r32, r/m32*	Move if not equal (ZF=0)
CMOVNG *r16, r/m16*	Move if not greater (ZF=1 or SF<>OF)
CMOVNG *r32, r/m32*	Move if not greater (ZF=1 or SF<>OF)
CMOVNGE *r16, r/m16*	Move if not greater or equal (SF<>OF)
CMOVNGE *r32, r/m32*	Move if not greater or equal (SF<>OF)
CMOVNL *r16, r/m16*	Move if not less (SF=OF)
CMOVNL *r32, r/m32*	Move if not less (SF=OF)
CMOVNLE *r16, r/m16*	Move if not less or equal (ZF=0 and SF=OF)
CMOVNLE *r32, r/m32*	Move if not less or equal (ZF=0 and SF=OF)
CMOVNO *r16, r/m16*	Move if not overflow (OF=0)
CMOVNO *r32, r/m32*	Move if not overflow (OF=0)
CMOVNP *r16, r/m16*	Move if not parity (PF=0)
CMOVNP *r32, r/m32*	Move if not parity (PF=0)
CMOVNS *r16, r/m16*	Move if not sign (SF=0)
CMOVNS *r32, r/m32*	Move if not sign (SF=0)
CMOVNZ *r16, r/m16*	Move if not zero (ZF=0)
CMOVNZ *r32, r/m32*	Move if not zero (ZF=0)
CMOVO *r16, r/m16*	Move if overflow (OF=0)
CMOVO *r32, r/m32*	Move if overflow (OF=0)
CMOVP *r16, r/m16*	Move if parity (PF=1)
CMOVP *r32, r/m32*	Move if parity (PF=1)
CMOVPE *r16, r/m16*	Move if parity even (PF=1)
CMOVPE *r32, r/m32*	Move if parity even (PF=1)
CMOVPO *r16, r/m16*	Move if parity odd (PF=0)
CMOVPO *r32, r/m32*	Move if parity odd (PF=0)
CMOVS *r16, r/m16*	Move if sign (SF=1)
CMOVS *r32, r/m32*	Move if sign (SF=1)
CMOVZ *r16, r/m16*	Move if zero (ZF=1)
CMOVZ *r32, r/m32*	Move if zero (ZF=1)



A. Inserting **a single register** to produce a two-stage pipeline. Where should the register be inserted to maximize throughput? What would be the throughput and latency?

B. Where should **two registers** be inserted to maximize the throughput of a three-stage pipeline? What would be the throughput and latency?

? stages	Where?	Throughput and latency
2	A | B, 80 | 220 B | C, 110 | 190 **C | D, 170 | 130** D | E, 220 | 80 E | F, 290 | 10	Th = 1 / ((170 + 20) * 10^{-12}) = 5.26 * 10^{9} IPS Latency = (170 + 20) * 2 = 380 ps
3	10 possibilities AB | CD | EF, 110 | 110 | 80	Th = 1 / ((110 + 20) * 10^{-12}) = 7.69 * 10^{9} IPS Latency = (110 + 20) * 3 = 390 ps