Machine-Level Programming II Arithmetic and logical operations and control flow

Data movement

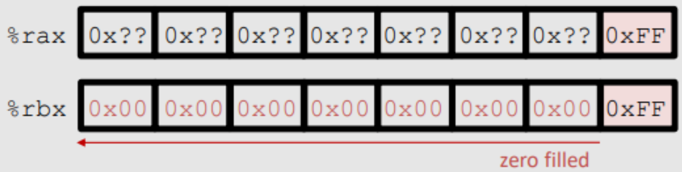
Of interest: `movz` and `movs`

`movz__` *src, regDest* *Move with zero extension*
`movs__` *src, regDest* *Move with sign extension*

- Copy from a *smaller* source value to a *larger* destination
- Source can be memory or register; Destination *must* be a register
- Fill remaining bits of dest with **zero**(mov**z**) or **sign bit** (mov**s**)

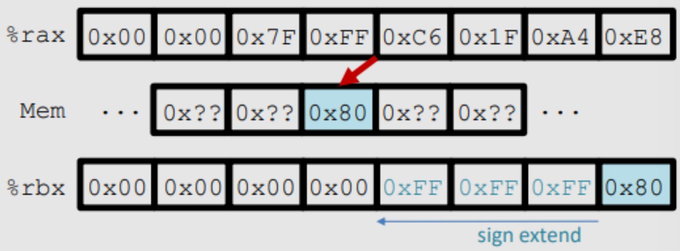
movz*SD* / **movs*SD***
S – size of source (**b**=1 byte, **w**=2)
D – size of dest (**w**=2 bytes, **l**=4, **q**=8)

Example: `movzbq %al, %rbx`



movz*SD* / **movs*SD***
S – size of source (**b**=1 byte, **w**=2)
D – size of dest (**w**=2 bytes, **l**=4, **q**=8)

Example: `movsbl (%rax), %ebx`



Arithmetic and Logic Operations

Address computation instruction

- **`lea` src, dest**
  - "lea" stands for ***load effective address***
  - **src** is memory address mode expression
  - set **dest** to address computed by expression
  - **dest** is a register
- **Used when:**
  - computing addresses **without** a memory reference
  - computing arithmetic expressions of the form $x + k * y$, where k = 1, 2, 4 or 8

Code example:

```
long mult_12(long x) {
    return x*12;
}
```

Generated assembly with optimization:

```
mult_12:
leaq (%rdi,%rdi,2), %rax
salq $2, %rax
ret
```

Register	Use
`%rdi`	Argument `x`
`%rax`	Return value

Example: `lea` vs `mov`

Registers

`%rax`	
`%rbx`	
`%rcx`	`0x4`
`%rdx`	`0x100`
`%rdi`	
`%rsi`	

Memory

`0x400`	`0x120`
`0xF`	`0x118`
`0x8`	`0x110`
`0x10`	`0x108`
`0x1`	`0x100`

```
leaq (%rdx,%rcx,4), %rax
movq (%rdx,%rcx,4), %rbx
leaq (%rdx), %rdi
movq (%rdx), %rsi
```

Registers

%rax	0x110
%rbx	0x8
%rcx	0x4
%rdx	0x100
%rdi	0x100
%rsi	0x1

Memory

0x400	0x120
0xF	0x118
0x8	0x110
0x10	0x108
0x1	0x100

```
leaq (%rdx,%rcx,4), %rax
movq (%rdx,%rcx,4), %rbx
leaq (%rdx), %rdi
movq (%rdx), %rsi
```

Example x86 Arithmetic Operations

- Two-operand instructions (longword variants)
- Watch out for argument order!

Instruction	Operation	Notes
`addl` *src,dest*	dest = dest + src	Addition
`subl` *src,dest*	dest = dest - src	Subtraction
`imull` *src,dest*	dest = dest * src	Multiplication
`sall` *src,dest*	dest = dest << src	Shift arithmetic left
`sarl` *src,dest*	dest = dest >> src	Shift arithmetic right
`xorl` *src,dest*	dest = dest ^ src	Bitwise xor
`andl` *src,dest*	dest = dest & src	Bitwise and
`orl` *src,dest*	dest = dest | src	Bitwise or

Quick way to multiply and divide by powers of 2

- One-operand instructions (longword variants)

Instruction	Operation	Notes
`incl` *dest*	dest = dest + 1	Increment by 1
`decl` *dest*	dest = dest - 1	Decrement by 1
`negl` *dest*	dest = -dest	Negate
`notl` *dest*	dest = ~dest	Bitwise not

Special Arithmetic Operations

- These operations provide 128-bits

Instruction	Operation	Notes
`imulq` *src*	R[%rdx]:R[%rax] ← src X R[%rax]	Signed multiplication
`mulq` *src*	R[%rdx]:R[%rax] ← src X R[%rax]	Unsigned multiplication
`idivq` *src*	R[%rdx]← R[%rdx]:R[%rax] mod src; R[%rax] ← R[%rdx]:R[%rax] ÷ src	Signed divide
`divq` *src*	R[%rdx]← R[%rdx]:R[%rax] mod src; R[%rax] ← R[%rdx]:R[%rax] ÷ src	Unsigned divide
`cqto`	R[%rdx]:R[%rax] ← signExtend(R[%rax])	Convert to octal word

Arithmetic expression example

Arithmetic expression example

Register	Use(s)
%rdi	x
%rsi	y
%rdx	z, t4
%rax	t1, t2, rval
%rcx	t5

Code example:

```
int arithmetic
(int x, int y, int z) {
    int t1 = x + y;
    int t2 = z + t1;
    int t3 = x + 4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

Generated assembly with optimization:

```
arithmetic:

    leal    (%rdi,%rsi),%eax     # eax = x+y
    addl    %edx, %eax           # eax = edx+eax
    leal    (%rsi,%rsi,2),%edx   # edx = y*3
    sall    $4, %edx             # edx = edx*16
    leal    4(%rdi,%rdx),%ecx    # ecx = x+4+edx
    imull   %ecx, %eax           # eax = eax*ecx
    ret
```

Boolean expression example

Boolean expression example

Code example:

```c
int logical
(int x, int y) {
    int t1 = x ^ y;
    int t2 = t1 >> 17;
    int t3 = (1<<13) - 7;
    int rval = t2 & t3;
    return rval;
}
```

Generated assembly with optimization:

```
logical:

  xorl %esi, %edi        # edi = x ^ y
  sarl $17, %edi         # edi = edi >> 17
  movl %edi, %eax        #
  andl $8185, %eax       # eax = t2 & 8185
  ret
```

Register	Use
%edi	x, t1, t2
%esi	y
%eax	t3, rval

- Generating the mask t3
  $2^{13} = 8192, 2^{13} - 7 = 8185$

Control Flow: Condition Codes

Control Flow

Code example:

```c
long max(long x, long y)
{
  long max;
  if (x > y){
    max = x;
  } else {
    max = y;
  }
  return max;
}
```

Assembly

```
max:
   ???
   movq  %rdi, %rax
   ???
   ???
   movq  %rsi, %rax
   ???
   ret
```

Register	Use
%rdi	Argument x
%rsi	Argument y
%rax	Return value

Code example:

```c
long max(long x, long y)
{
  long max;
  if (x > y){
    max = x;
  } else {
    max = y;
  }
  return max;
}
```

Assembly

Conditional jump

```
max:
   if x<=y then jump to else
   movq  %rdi, %rax
   jump to done
else:
   movq  %rsi, %rax
done:
   ret
```

Unconditional jump

Register	Use
%rdi	Argument x
%rsi	Argument y
%rax	Return value

Conditionals and Control Flow

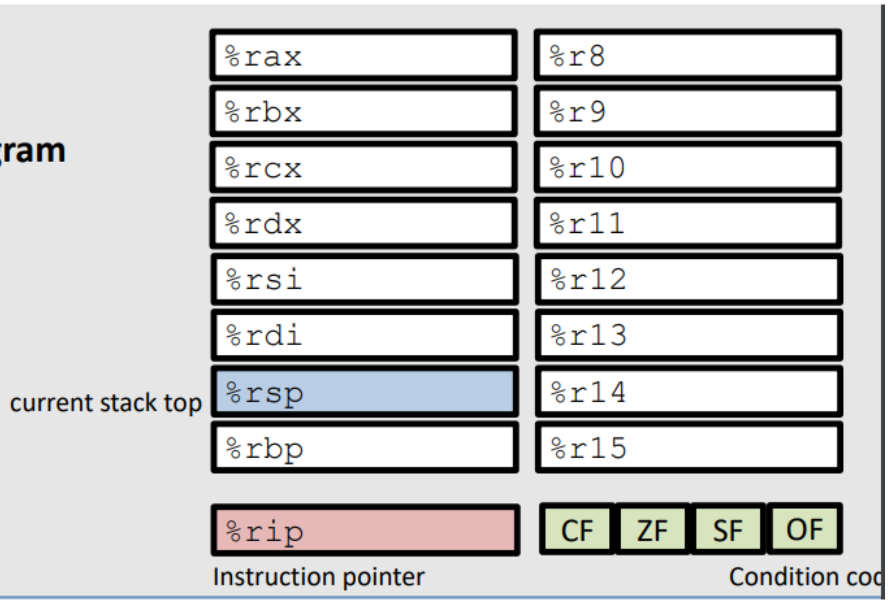
- **Conditional branch/jump**
  - Jump to somewhere else if some *condition* is true otherwise execute the next instruction
- **Unconditional branch/jump**
  - *Always* jump when you get to this instruction
  - For example: `break`, `continue`

- They can implement most control flow constructs in high-level languages:
  - **`if`** (*condition*) **`then`** {…} **`else`** {…}
  - **`while`** (*condition*) {…}
  - **`do`** {…} **`while`** (*condition*)
  - **`for`** (*initialization*; *condition*; *iterative*) {…}
  - **`switch`** {…}

Processor State (x86-64, partial)

Information about

- **currently executing program**
- Temporary data
  - (`%rax`, …)
  - Location of runtime stack
  - (**`%rsp`**)
  - Location of current code
  - control point
  - (**`%rip`**, …)
  - Status of recent tests
  - (**CF**, **ZF**, **SF**, **OF**)



Condition codes (implicit setting)

- **Single bit registers**

  CF – Carry Flag (for unsigned)
  ZF – Zero Flag
  SF – Sign Flag (for signed)
  OF – Overflow Flag (for signed)

- **Implicitly set** (think of it as a side effect) **by arithmetic operations** (not by `lea`)

  Example: `addl/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)`

Condition codes (explicit setting: compare)

- **Explicit setting** by a `compare` **instruction**

  `cmpl/cmpq` *Src2,Src1*

  Example: `cmpl 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)`

Condition codes (explicit setting: test)

- **Explicit setting by a** `test` **instruction**

  `testl/testq` *Src2,Src1*

  Example: `testl 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`
- `testl %eax, %eax`
  - Sets SF and ZF, check if `eax` is +,0,-

Reading Condition codes (CC)

- `set*` **instructions**: set low order byte to 0 or 1 based on computation of CC.

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

As a programmer, you never access the condition flags directly

Reading condition codes (cont.)

- `set*` **instructions:**
- set single byte based on combination of condition codes
- **One of 16 addressable byte registers**
  - Does not alter remaining 3-7 bytes
  - Typically use `movzbl` to finish job

%rax	%eax	%ah	%al

Register	Use
`%edi`	x
`%esi`	y

Code example:

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

Body

```
cmpl %esi, %edi      # compare x : y
setg %al             # al = x > y
movzbl %al, %eax     # zero rest of %rax
```

Jumping

Instruction	Condition	Description
jmp *target*	1	Unconditional (direct jump)
jmp *Operand	1	Unconditional (indirect jump)
je *target*	`ZF`	Equal / Zero
jne *target*	`~ZF`	Not Equal / Not Zero
js *target*	`SF`	Negative
jns *target*	`~SF`	Nonnegative
jg *target*	`~(SF^OF)&~ZF`	Greater (signed)
jge *target*	`~(SF^OF)`	Greater or equal (signed)
jl *target*	`(SF^OF)`	Less (signed)
jle *target*	`(SF^OF)|ZF`	Less or equal (signed)
ja *target*	`~CF&~ZF`	Above (unsigned)
jb *target*	`CF`	Below (unsigned)

- `j*` **Instructions**: Jump to different part of the code indicated by **target** argument. Conditional jump depends on ***condition code registers***

Choosing instructions for conditionals

`compare` or `test`

		cmp b,a	**test b,a**
je	"Equal"	`a == b`	`a&b == 0`
jne	"Not equal"	`a != b`	`a&b != 0`
js	"Sign" (negative)		`a&b < 0`
jns	(non-negative)		`a&b >= 0`
jg	"Greater"	`a > b`	`a&b > 0`
jge	"Greater or equal"	`a >= b`	`a&b >= 0`
jl	"Less"	`a < b`	`a&b < 0`
jle	"Less or equal"	`a <= b`	`a&b <= 0`
ja	"Above" (unsigned >)	`a > b`	
jb	"Below" (unsigned <)	`a < b`	

`jump` or `set`

Examples:

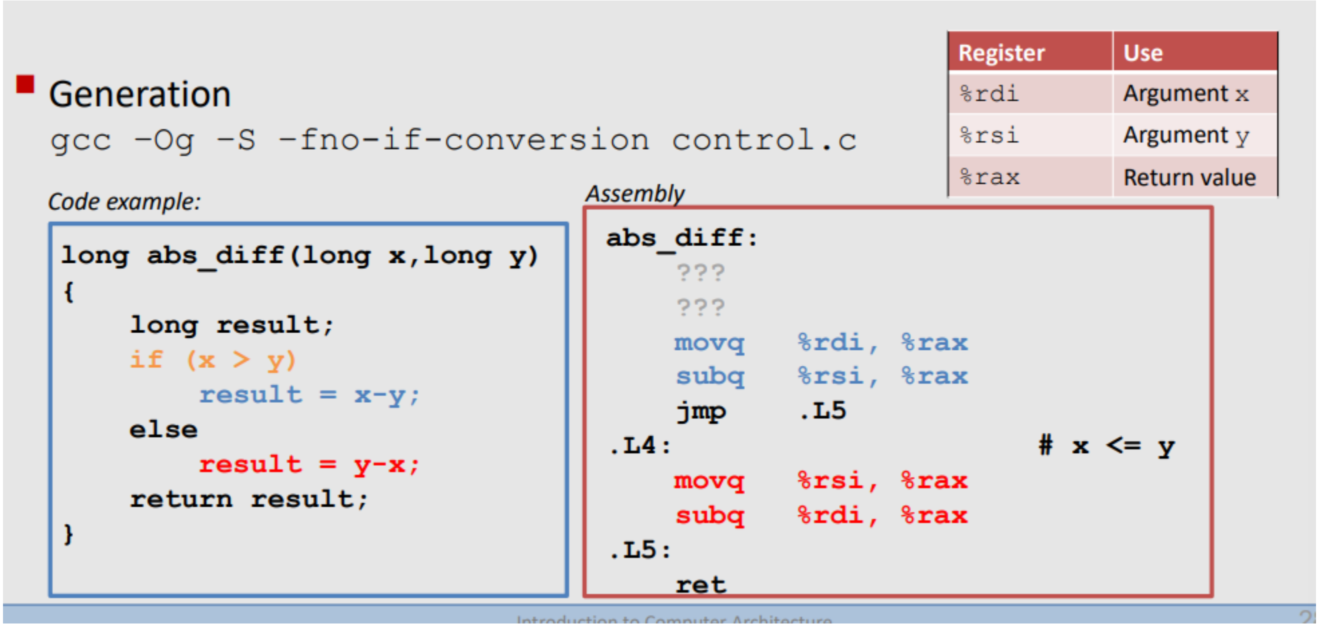
```
        cmp 5, (%rax)
je:   (%rax) == 5
jne:  (%rax) != 5
jg:   (%rax) >  5
jl:   (%rax) <  5
```

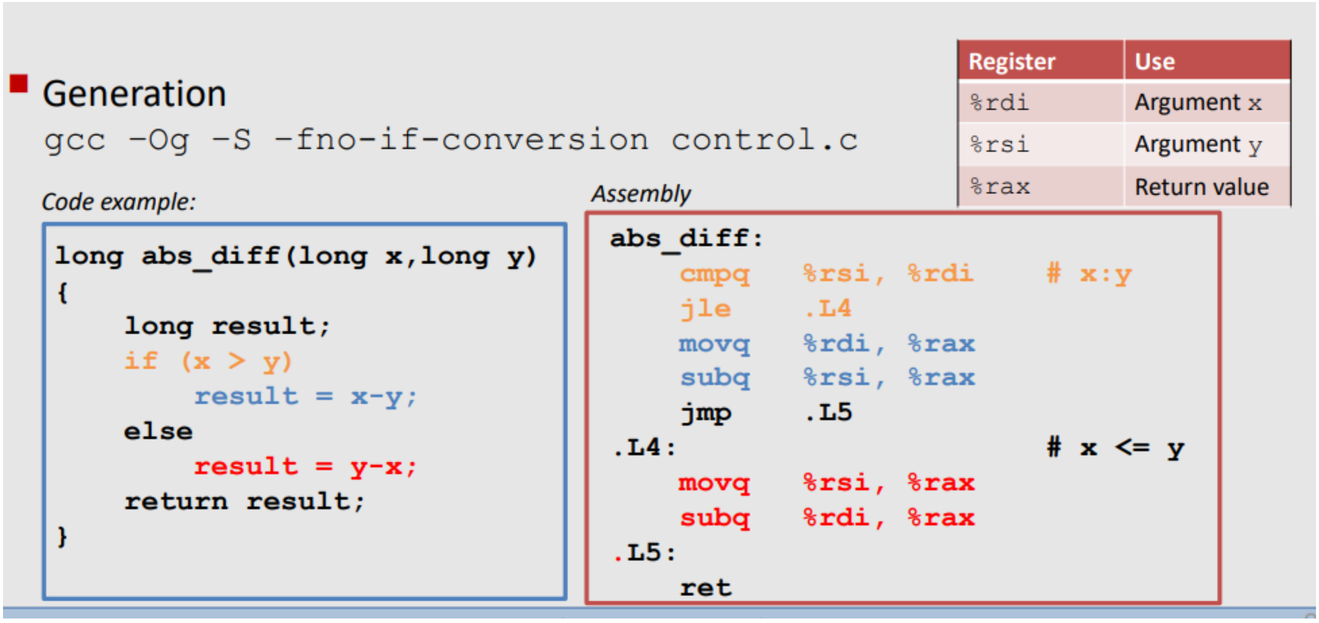
```
        test %rdi, %rdi
je:    %rdi == 0
jne:   %rdi != 0
jg:    %rdi >  0
jl:    %rdi <  0
```

```
        test %rax, 0x1
je:    %rax_LSB == 0
jne:   %rax_LSB == 1
```

Conditional control and data movement

Conditional branch example





General conditional expression translation with jump (`goto`)

- val = *Test* ? *Then_Expr* : *Else_Expr*;

C/Java code example

```
val = x>y ? x-y; y-x;
```

C allows `goto` as means of transferring control (`jump`):
- Closer to assembly programming style
- Generally considered **bad** coding style

Code example with `goto`:

```
    ntest = !Test;
    if (ntest) goto else;
    val = Then_Expr;
    goto done;
else:
    val = Else_Expr;
done:
    ...
```

- Create separate code regions for **then** and **else** expressions
- Execute appropriate one
- Can it be made more efficient?

(Spring'24) Introduction to Computer Architecture

Using conditional moves (in x86-64)

- **Conditional move instructions (`cmov*`)**
  - Instruction supports:

    `if (Test) Dest ← Src`
  - Move value from **`src`** to **`dest`** if condition **`Test`** holds
- **Why do we use it?**
  - More efficient than conditional branching (simple control flow)
  - But, there is overhead, as both branches are evaluated.

Conditional move example

Conditional move instructions

- do the move in case a condition has been satisfied.

Register	Use
`%rdi`	Argument x
`%rsi`	Argument y
`%rax`	Return value

Code example:

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

Assembly

```
absdiff:
   movq   %rdi, %rdx   # x
   subq   %rsi, %rdx   # res = x-y
   movq   %rsi, %rax
   subq   %rdi, %rax   # eval = y-x
   cmpq   %rsi, %rdi   # x:y
   ???
   ret
```

Conditional move instructions

- do the move in case a condition has been satisfied.

Register	Use
%rdi	Argument x
%rsi	Argument y
%rax	Return value

Code example:

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

Assembly

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

Bad cases for conditional move

- Expensive computations
  - both values get computed
  - makes sense when computations are simple
- Risky computations
  - both values get computed
  - may have undesirable effects
- Computations with side-effects
  - both values get computed
  - must be side-effect free

Example 1

```
val = Test(x)
    ? Hard1(x)
    : Hard2(x);
```

Example 2

```
val = p ? *p : 0;
```

Example 3

```
val = x > 0
    ? x*=7
    : x+=3
```

Summary

- **`lea` is address calculation instruction**
  - Does NOT actually go to memory
  - Used to compute address or some arithmetic expression

- **Control flow in x86 is determined by status of Condition Codes**
  - Showed **C**arry, **Z**ero, **S**ign, and **O**verflow, though others exist as well
  - Set flags with arithmetic operations (implicit) or `compare` and `test` (explicit)
  - `set*` instructions read out flag values
  - `j*` instructions use flag values to determine next instruction to execute
  - `cmov*` instructions use flag values to execute a move instruction