Machine-Level Programming: Arithmetic and Logical Operations Condition Codes

CMPU 224 – Computer Organization
Jason Waterman

Address Computation Instruction

- **`leaq`** ***Src*, *Dst***
  - *Src* is address mode expression
  - Set *Dst* to address denoted by expression

- Uses
  - Computing addresses without the memory reference
    - E.g., translation of **`p = &x[i];`**
  - Computing arithmetic expressions of the form x + k*y
    - k = 1, 2, 4, or 8

- Example

```
long m12(long x)
{
  return x*12;
}
```

Converted to ASM by compiler:

```
leaq (%rdi,%rdi,2), %rax # t <- x+x*2
salq $2, %rax            # return t<<2
```

Some Arithmetic Operations

- Two Operand Instructions:

Format		***Computation***	
`addq`	*Src, Dest*	Dest = Dest + Src	
`subq`	*Src, Dest*	Dest = Dest – Src	
`imulq`	*Src, Dest*	Dest = Dest * Src	
`sarq`	*Src, Dest*	Dest = Dest >> Src	***Arithmetic shift***
`shrq`	*Src, Dest*	Dest = Dest >> Src	***Logical shift***
`salq`	*Src, Dest*	Dest = Dest << Src	***Also called shlq***
`xorq`	*Src, Dest*	Dest = Dest ^ Src	
`andq`	*Src, Dest*	Dest = Dest & Src	
`orq`	*Src, Dest*	Dest = Dest | Src	

- Watch out for argument order, `subq` in particular
- No distinction between signed and unsigned int (why?)

Some Arithmetic Operations

- One Operand Instructions

`incq`	*Dest*	*Dest = Dest + 1*
`decq`	*Dest*	*Dest = Dest – 1*
`negq`	*Dest*	*Dest = – Dest*
`notq`	*Dest*	*Dest = ~Dest*

- See book for more instructions

x86-64 Processor State

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

Registers

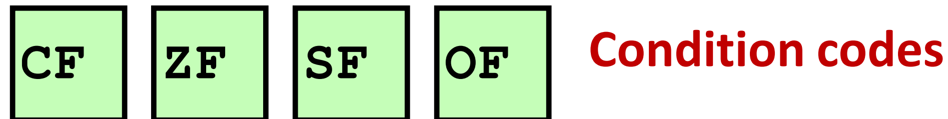
`%rax`	`%r8`
`%rbx`	`%r9`
`%rcx`	`%r10`
`%rdx`	`%r11`
`%rsi`	`%r12`
`%rdi`	`%r13`
`%rsp`	`%r14`
`%rbp`	`%r15`

`%rip` **Instruction pointer**

`CF` `ZF` `SF` `OF` **Condition codes**

Condition Codes

- Single bit registers set by arithmetic and logic operations
  - **ZF** Zero Flag – The most recent operation yielded zero
  - **SF** Sign Flag – The most recent operation yielded a negative value (signed)
  - **CF** Carry Flag – The most recent operation generated a carry out of the MSB
    - Designates overflow (unsigned)
  - **OF** Overflow Flag – The most recent operation caused a two's-complement overflow, either positive or negative (signed)



Condition codes

Setting Condition

- They are implicitly set (think of it as side effect) by arithmetic/logic operations based on the result of the operation
  - For logical operations, the carry and overflow flags are set to zero
  - For shift operations, `CF` is set to the last bit shifted out, `OF` is set to zero
  - `INC` and `DEC` set `OF` and `ZF`, but leave the carry flag unchanged

- **Not** set by **`leaq`** instruction

- Condition Codes are not accessed directly, but some instructions alter their behavior based on the value of the Condition Codes

Setting Condition Codes Explicitly with Compare

- Compare Instruction: `cmp S1, S2`
  - Similar to the `sub` (subtract) instruction
  - Sets the condition codes according to the differences of their two operands ($S_2 - S_1$) but **without setting the destination operand**
  - Used to compare two numbers
  - Example: **`cmp b,a`**

    Read as: *a compare b* (also as `a:b`)
- Operands are reversed for a compare
  - Why? AT&T vs Intel assembler syntax
  - In Intel syntax operands are reversed compared to AT&T syntax
  - We use AT&T style syntax, so remember to switch the order of operands for compare

Comparing Two Numbers Using Subtraction

- By subtracting two numbers you can compare them!
  - Example: A – B

- **Equality**: when A and B are equal, A – B == 0 (ZF)
- **Not Equal**: When A – B != 0 (~ZF)
- **Greater than**: when A > B, A – B == Positive number and not zero (~SF & ~ZF)
- **Greater than or equal:** when A >= B, A – B == Positive number or zero (~SF | ZF)
- **Less than**: when A < B, A – B == Negative number (SF)
- **Less than or equal:** when A <= B, A – B == Negative number or zero (SF | ZF)

Test instruction

- Like the `cmp` instruction, `test` is used to set condition codes
- Test Instruction: `test` S_1, S_2
  - Similar to the `and` (bitwise and) instruction
  - Sets the ZF and the SF based on (S_2 & S_1) but **without setting the destination operand**
  - Often the same operand repeated (`testq %rax, %rax`) to check if the value is zero, positive, or negative

Reading Condition Codes (SetX instrutions)

- SetX Instructions
  - Set destination to 0 or 1 based on combinations of condition codes
  - Destination must be a low-order byte register or single byte memory location
  - Does not alter remaining 7 bytes for register destinations

Instruction	Synonym	Effect	Set condition
set**e** D	setz	D ← ZF	Equal / zero
set**ne** D	setnz	D ← ~ZF	Not equal / not zero
set**s** D		D ← SF	Negative
set**ns** D		D ← ~SF	Nonnegative
set**g** D	setnle	D ← ~(SF ^ OF) & ~ZF	Greater (signed >)
set**ge** D	setnl	D ← ~(SF ^ OF)	Greater or equal (signed >=)
set**l** D	setnge	D ← SF ^ OF	Less (signed <)
set**le** D	setng	D ← (SF ^ OF) | ZF	Less or equal (signed <=)
set**a** D	setnbe	D ← ~CF & ~ZF	Above (unsigned >)
set**ae** D	setnb	D ← ~CF	Above or equal (unsigned >=)
set**b** D	setnae	D ← CF	Below (unsigned <)
set**be** D	setna	D ← CF | ZF	Below or equal (unsigned <=)

Reading Condition Codes (Cont.)

- SetX Instructions:
  - Set single byte based on combination of condition codes; descriptions apply after a cmpq instruction – **remember to reverse your operands!**
- 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

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

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

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

Conditional Moves

Instruction	Synonym	Move Condition	Description
`cmove S,R`	`cmovz`	`ZF`	Equal / zero
`cmovne S,R`	`cmovnz`	`~ZF`	Not equal / not zero
`cmovs S,R`		`SF`	Negative
`cmovns S,R`		`~SF`	Nonnegative
`cmovg S,R`	`cmovnle`	`~(SF ^ OF) & ~ZF`	Greater (signed >)
`cmovge S,R`	`cmovnl`	`~(SF ^ OF)`	Greater or equal (signed >=)
`cmovl S,R`	`cmovnge`	`SF ^ OF`	Less (signed <)
`cmovle S,R`	`cmovng`	`(SF ^ OF) | ZF`	Less or equal (signed <=)
`cmova S,R`	`cmovnbe`	`~CF & ~ZF`	Above (unsigned >)
`cmovae S,R`	`cmovnb`	`~CF`	Above or equal (Unsigned >=)
`cmovb S,R`	`cmovnae`	`CF`	Below (unsigned <)
`cmovbe S,R`	`cmovna`	`CF | ZF`	Below or equal (unsigned <=)

- Destination must be a register
- Source can be register or memory location
- Length is determined by the name of the destination register

Using Conditional Moves

- Conditional Move Instructions
  - Instruction supports:

    if (Test) Dest ← Src
  - Supported in post-1995 x86 processors
  - GCC tries to use them
    - But only when known to be safe

C Code

```
val = Test ? Then_Expr : Else_Expr;
```

Conditional Move Pseudo Code

```
val = Else_Expr;
eval = Then_Expr;
if (Test) val = eval;
```

Conditional Move Example

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

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

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

Bad Cases for Conditional Move

Expensive Computations

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

- Both values get computed
- Only makes sense when computations are very simple

Risky Computations

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

- Both values get computed
- May have undesirable effects

Computations with side effects

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

- Both values get computed
- Must be side-effect free