Assembly II: Control Flow

Jin-Soo Kim (jinsookim@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
Processor State (x86-64)

General-purpose registers (temporary data)
- RAX
- RBX
- RCX
- RDX
- RSI
- RDI
- RBP
- RSP

Current stack frame (base pointer)
- RBP
- ESP

General-purpose registers (temporary data)
- R8
- R9
- R10
- R11
- R12
- R13
- R14
- R15

Instruction Pointer (location of the next instruction)
- RIP

Condition codes (status of recent tests)
- EFLAGS
Instruction Pointer

- **RIP register**
  - Contains the offset in the current code segment for the next instruction to be executed
    - Advanced from one instruction boundary to the next in straightline code, or
    - Moved ahead or backwards by instructions such as JMP, Jcc, CALL, RET, and IRET
  - Cannot be accessed directly by software
    - RIP is controlled implicitly by control transfer operations, interrupts, and exceptions
  - Because of instruction prefetching, an instruction address read from the bus does not match the value in the RIP register
EFLAGS Register

- ID Flag (ID)
- Virtual Interrupt Pending (VIP)
- Virtual Interrupt Flag (VIF)
- Alignment Check (AC)
- Virtual-8086 Mode (VM)
- Resume Flag (RF)
- Nested Task (NT)
- I/O Privilege Level (IOPL)

- Overflow Flag (OF)
- Direction Flag (DF)
- Interrupt Enable Flag (IF)
- Trap Flag (TF)

- Sign Flag (SF)
- Zero Flag (ZF)
- Auxiliary Carry Flag (AF)
- Parity Flag (PF)
- Carry Flag (CF)

S Indicates a Status Flag
C Indicates a Control Flag
X Indicates a System Flag

Reserved bit positions. DO NOT USE.
Always set to values previously read.
Status Flags

CF (Carry): set if an arithmetic operation generates a carry or a borrow; indicates an overflow condition for unsigned-integer arithmetic.

PF (Parity): set if the least-significant byte of the result contains an even number of 1 bits

AF (Adjust): set if an arithmetic operation generates a carry or a borrow out of bit 3 of the result; used in binary-coded decimal (BCD) arithmetic

ZF (Zero): set if the result is zero

SF (Sign): set equal to the most-significant bit of the result

OF (Overflow): set if the integer result is too large a positive number or too small a negative number to fit in the destination operand; indicates an overflow condition for signed-integer arithmetic.

DF (Direction): setting the DF causes the string instructions to auto-decrement; set and cleared by STD/CLD instructions
Condition Codes: Implicit Setting

- Single bit registers
  - CF (Carry), SF (Sign), ZF (Zero), OF (Overflow)

- Implicitly set by arithmetic operations
  - Example: `addq Src, Dest` (t = a + b)
  - CF set if carry out from most significant bit
    - Used to detect unsigned overflow
  - ZF set if t == 0
  - SF set if t < 0
  - OF set if two’s complement (signed) overflow:
    (a > 0 && b > 0 && t < 0) || (a < 0 && b < 0 && t > 0)

- Not set by `leaq`, `incq`, or `decq` instruction
Condition Codes: Compare

- Explicitly setting by Compare instruction
  - Example: `cmpq b, a`
  - Computes \((a - b)\) without saving the result
  - CF set if carry out from most significant bit
    - Used for unsigned comparison
  - ZF set if \(a == b\)
  - SF set if \((a - b) < 0\) (as signed)
  - OF set if two’s complement overflow:
    \((a > 0 && b < 0 && (a - b) < 0) || (a < 0 && b > 0 && (a - b) > 0)\)
Condition Codes: Test

- Explicitly setting by Test instruction
  - Example: `testq b, a`
  - Computes \((a \& b)\) without saving the result
    - Useful to have one of the operations be a mask
  - ZF set when \(a \& b == 0\)
  - SF set when \(a \& b < 0\)
  - CF and OF are cleared to 0
Reading Condition Codes

- **setX instructions**
  - Set single byte based on combination of condition codes

<table>
<thead>
<tr>
<th>setX</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sete</td>
<td>R₈ ← ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>setne</td>
<td>R₈ ← ~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>sets</td>
<td>R₈ ← SF</td>
<td>Negative</td>
</tr>
<tr>
<td>setns</td>
<td>R₈ ← ~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>setg</td>
<td>R₈ ← ~(SF ^ OF) &amp; ~ZF</td>
<td>Greater (Signed &gt;)</td>
</tr>
<tr>
<td>setge</td>
<td>R₈ ← ~(SF ^ OF)</td>
<td>Greater or Equal (Signed &gt;=)</td>
</tr>
<tr>
<td>setl</td>
<td>R₈ ← (SF ^ OF)</td>
<td>Less (Signed &lt;)</td>
</tr>
<tr>
<td>setle</td>
<td>R₈ ← (SF ^ OF)</td>
<td>ZF</td>
</tr>
<tr>
<td>seta</td>
<td>R₈ ← ~CF &amp; ~ZF</td>
<td>Above (Unsigned &gt;)</td>
</tr>
<tr>
<td>setae</td>
<td>R₈ ← ~CF</td>
<td>Above or Equal (Unsigned &gt;=)</td>
</tr>
<tr>
<td>setb</td>
<td>R₈ ← CF</td>
<td>Below (Unsigned &lt;)</td>
</tr>
<tr>
<td>setbe</td>
<td>R₈ ← CF</td>
<td>ZF</td>
</tr>
</tbody>
</table>
# 8-bit Registers

<table>
<thead>
<tr>
<th>8-bit Registers</th>
<th>32-bit Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>%al</td>
</tr>
<tr>
<td>%rbx</td>
<td>%bl</td>
</tr>
<tr>
<td>%rcx</td>
<td>%cl</td>
</tr>
<tr>
<td>%rdx</td>
<td>%dl</td>
</tr>
<tr>
<td>%rsi</td>
<td>%sil</td>
</tr>
<tr>
<td>%rdi</td>
<td>%sil</td>
</tr>
<tr>
<td>%rsp</td>
<td>%spl</td>
</tr>
<tr>
<td>%rbp</td>
<td>%bpl</td>
</tr>
<tr>
<td>%r8</td>
<td>%r8b</td>
</tr>
<tr>
<td>%r9</td>
<td>%r9b</td>
</tr>
<tr>
<td>%r10</td>
<td>%r10b</td>
</tr>
<tr>
<td>%r11</td>
<td>%r11b</td>
</tr>
<tr>
<td>%r12</td>
<td>%r12b</td>
</tr>
<tr>
<td>%r13</td>
<td>%r13b</td>
</tr>
<tr>
<td>%r14</td>
<td>%r14b</td>
</tr>
<tr>
<td>%r15</td>
<td>%r15b</td>
</tr>
</tbody>
</table>
Condition Codes: Example

- **setX instructions**
  - Set the 8-bit register to 0 or 1 based on condition codes
  - Does not alter remaining 7 bytes
  - Typically use `movz`** to finish job
    - `dest ← ZeroExtend (src);`

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

```assembly
cmpq  %rsi, %rdi # Compare x : y
setg  %al  # set %al when x > y
movzb %al,%eax # Zero rest of %eax
ret
```

Note inverted ordering!
Conditional Branch

- **jX** instructions
  - Jump to different part of code depending on condition codes

<table>
<thead>
<tr>
<th>jX</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp</td>
<td>1</td>
<td>Unconditional</td>
</tr>
<tr>
<td>je</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>jne</td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>js</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>jns</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>jg</td>
<td>~(SF ^ OF) &amp; ~ZF</td>
<td>Greater (Signed &gt;)</td>
</tr>
<tr>
<td>jge</td>
<td>~(SF ^ OF)</td>
<td>Greater or Equal (Signed &gt;=)</td>
</tr>
<tr>
<td>jl</td>
<td>(SF ^ OF)</td>
<td>Less (Signed &lt;)</td>
</tr>
<tr>
<td>jle</td>
<td>(SF ^ OF)</td>
<td>ZF</td>
</tr>
<tr>
<td>ja</td>
<td>~CF &amp; ~ZF</td>
<td>Above (Unsigned &gt;)</td>
</tr>
<tr>
<td>jae</td>
<td>~CF</td>
<td>Above or Equal (Unsigned &gt;=)</td>
</tr>
<tr>
<td>jb</td>
<td>CF</td>
<td>Below (Unsigned &lt;)</td>
</tr>
<tr>
<td>jbe</td>
<td>CF</td>
<td>ZF</td>
</tr>
</tbody>
</table>
Conditional Branch Example (I)

C allows “goto” as means of transferring control
  • Jump to position designated by label
  • Closer to machine-level programming style

Generally considered bad coding style

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

```c
long goto_max (long x, long y)
{
    int ok = (x <= y);
    if (ok) goto done;
    return x;
}
```

```c
done:
    return y;
}
```
Conditional Branch Example (2)

```c
long goto_max (long x, long y) {
    int ok = (x <= y);
    if (ok) goto done;
    return x;

done:
    return y;
}
```

```
max:
    cmpq %rsi, %rdi          # x - y?
    jle .L3                 # if <= goto .L3
    movq %rdi, %rax         # rax = x
    ret
 .L3:
    movq %rsi, %rax         # rax = y
    ret
```

x in %rdi
y in %rsi
Conditional Moves

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

- Why?
  - Branches are very disruptive to instruction flow through pipelines
  - Conditional moves do not require control transfer

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

```
max:  
    cmpq  %rsi, %rdi
    movq  %rsi, %rax
    cmovge %rdi, %rax
    ret  
```

x in %rdi
y in %rsi
Bad Cases for Conditional Moves

- Expensive computations

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

  - Only makes sense when computations are very simple

- Risky computations

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

  - May have undesirable effects

- Computations with side effects

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

  - Must be side-effect free
“Do-While” Loop (1)

- Example: compute factorial $x!$
  - Use backward branch to continue looping
  - Only take branch when “while” condition holds

**C Code**

```c
long fact_do (long x)
{
    long result = 1;
    do {
        result *= x;
        x = x - 1;
    } while (x > 1);
    return result;
}
```

**Goto Version**

```c
long fact_goto (long x)
{
    long result = 1;
    Loop:
    result *= x;
    x = x - 1;
    if (x > 1)
        goto loop;
    return result;
}
```
“Do-While” Loop (2)

Goto Version

```c
long fact_goto (long x) {
    long result = 1;
    loop:
        result *= x;
        x = x - 1;
        if (x > 1)
            goto loop;
    return result;
}
```

Assembly

```assembly
fact_goto:
    movl $1, %eax  # result = 1
    .L2:
        imulq %rdi, %rax  # result *= x
        subq $1, %rdi   # x--
        cmpq $1, %rdi   # compare x : 1
        jg .L2          # if > goto Loop
    ret
```

Registers

<table>
<thead>
<tr>
<th>%rdi</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>result</td>
</tr>
</tbody>
</table>
“Do-While” Loop (3)

- General “Do-While” translation

**C Code**

```
    do
      Body
    while (Test);
```

**Goto Version**

```
Loop:
  Body
  if (Test)
    goto loop

{
  Statement_1;
  Statement_2;
  ...
  Statement_n;
}
```

- **Body** can be any C statement
  - Typically compound statement:

- **Test** is expression returning integer:
  - = 0 interpreted as false, ≠ 0 interpreted as true
“While” Loop (1)

C Code

```c
long fact_while (long x) {
    long result = 1;
    while (x > 1) {
        result *= x;
        x = x-1;
    }
    return result;
}
```

First Goto Version

```c
long fact_while_goto (long x) {
    long result = 1;

    Loop:
    if (!(x > 1))
        goto done;
    result *= x;
    x = x-1;
    goto Loop;

done:
    return result;
}
```

- Is this code equivalent to the do-while version?
- Must jump out of loop if test fails
“While” Loop (2)

**C Code**

```c
long fact_while (long x) {
    long result = 1;
    while (x > 1) {
        result *= x;
        x = x - 1;
    }
    return result;
}
```

**Second Goto Version**

```c
long fact_while_goto2 (long x) {
    long result = 1;
    if (!(x > 1))
        goto done;

    loop:
    result *= x;
    x = x - 1;
    if (x > 1)
        goto loop;

    done:
    return result;
}
```

- Historically used by GCC
- Uses same inner loop as do-while version
- Guards loop entry with extra test
“While” Loop (3)

- General “While” translation

C Code

```
while (Test)
  Body
```

Do-While Version

```
if (!Test)
  goto done;
do
  Body
while (Test);
done:
```

Goto Version

```
if (!Test)
  goto done;
Loop:
  Body
if (Test)
  goto Loop;
done:
```
“For” Loop (1)

- Example: compute $x^p$
  - Exploit property that $p = p_0 + 2p_1 + 4p_2 + ... + 2^{n-1}p_{n-1}$
  - Gives: $x^p = z_0 \cdot z_1^2 \cdot (z_2^2)^2 \cdot ... \cdot (\ldots((z_{n-1}^2)^2)^2\ldots)^2$
    - $z_i = 1$ when $p_i = 0$
    - $z_i = x$ when $p_i = 1$
  - Complexity $O(\log p)$

```c
long ipwr_for(long x, unsigned long p) {
    long result;
    for (result = 1; p != 0; p = p>>1) {
        if (p & 0x1) result *= x;
        x = x*x;
    }
    return result;
}
```

Example:
$3^{10} = 3^2 * 3^8 = 3^2 * ((3^2)^2)^2$
“For” Loop (2)

General Form

\[
\text{for (Init; Test; Update)} \\
\text{Body}
\]

Init
\[\text{result} = 1\]

Test
\[p \neq 0\]

Update
\[p = p \gg 1\]

Body
\[
\{ \\
\text{if (p & 0x1)} \\
\text{result} *= x; \\
\text{x} = x*x; \\
\}
\]

Example

```c
long result;
for (result = 1; 
    p != 0; 
    p = p>>1) {
    if (p & 0x1)
        result *= x;
    x = x*x;
}  
```
“For” Loop (3)

For Version

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

While Version

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

Do-While Version

```plaintext
Init; 
if (!Test) 
  goto done; 
do { 
  Body
  Update; 
} while (Test) 
done:
```

Goto Version

```plaintext
Init; 
if (!Test) 
  goto done; 
loop: 
  Body
  Update; 
if (Test) 
  goto loop; 
done:
```
“For” Loop (4)

Goto Version

\[\begin{align*}
\text{Init} &; \\
& \text{if (!Test)} \\
& \quad \text{goto done;} \\
\text{Loop:} \\
& \quad \text{Body} \\
& \quad \text{Update;} \\
& \quad \text{if (Test)} \\
& \quad \quad \text{goto loop;} \\
\text{done:}
\end{align*}\]

\[\begin{align*}
\text{result} &= 1; \\
\text{if (p == 0)} \\
& \quad \text{goto done;} \\
\text{Loop:} \\
& \quad \text{if (p & 0x1)} \\
& \quad \quad \text{result} *= x; \\
& \quad \quad x = x*x; \\
& \quad \quad p = p >> 1; \\
& \quad \quad \text{if (p != 0)} \\
& \quad \quad \quad \text{goto loop;} \\
\text{done:}
\end{align*}\]
“Switch” Implementation

- **Series of conditionals**
  - Good if few cases
  - Slow if many

- **Jump table**
  - Lookup branch target and perform indirect jump
  - Avoids conditionals
  - Possible when cases are small integer constants

- **Binary search tree**
  - For sparse cases
  - Logarithmic performance

```c
typedef enum {  
    ADD, MULT, MINUS, DIV,  
    MOD, BAD  
} op_type;

char unparse_symbol(op_type op) {  
    switch (op) {  
    case ADD :  return '+';  
    case MULT:  return '*';  
    case MINUS: return '-';  
    case DIV:   return '/';  
    case MOD:   return '%';  
    case BAD:   return '?';  
    }  
}
```
Summary

- **C control**
  - if-then-else
  - do-while, while, for
  - switch

- **Assembler control**
  - Conditional jump
  - Conditional move
  - Indirect jump (via jump tables)
  - Compiler generates code sequence to implement more complex control