

Jin-Soo Kim
(jinsoo.kim@snu.ac.kr)

Systems Software &
Architecture Lab.

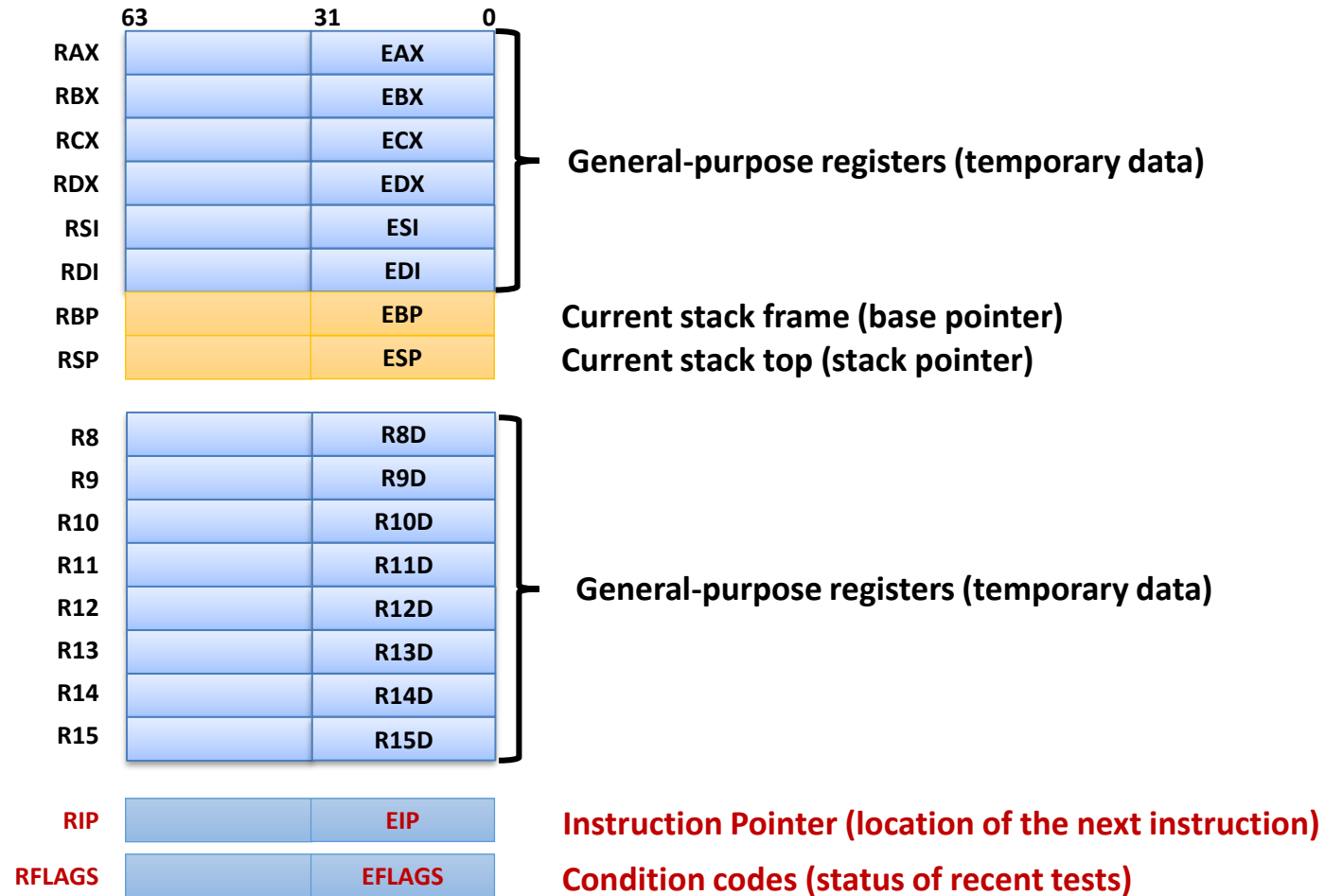
Seoul National University

Spring 2019

Assembly II: Control Flow



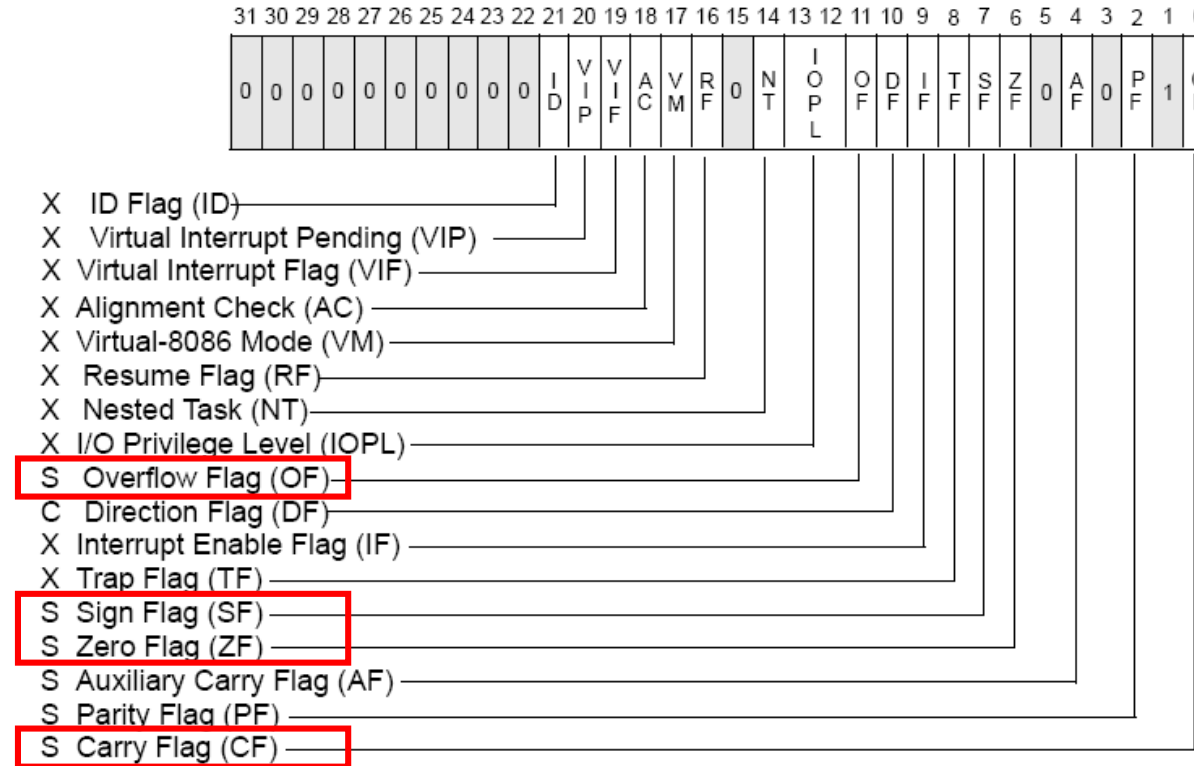
Processor State (x86-64)



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



- 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
- **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

Condition Codes: Implicit Setting

- 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

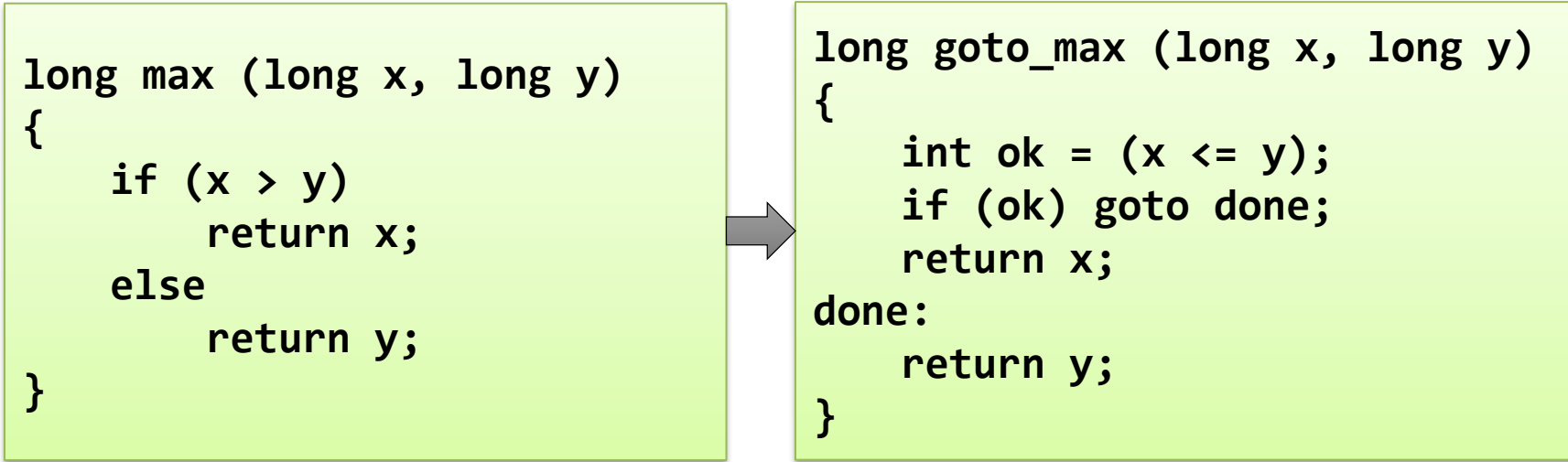
Conditional Branch

▪ jX instructions

- Jump to different part of code depending on condition codes

jX	Condition	Description
jmp	1	Unconditional
je	ZF	Equal / Zero
jne	\sim ZF	Not Equal / Not Zero
js	SF	Negative
jns	\sim SF	Nonnegative
jg	\sim (SF ^ OF) & \sim ZF	Greater (Signed >)
jge	\sim (SF ^ OF)	Greater or Equal (Signed >=)
j1	(SF ^ OF)	Less (Signed <)
jle	(SF ^ OF) ZF	Less or Equal (Signed <=)
ja	\sim CF & \sim ZF	Above (Unsigned >)
jae	\sim CF	Above or Equal (Unsigned >=)
jb	CF	Below (Unsigned <)
jbe	CF ZF	Below or Equal (Unsigned <=)

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

Conditional Branch Example (2)

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

x in %rdi
y in %rsi

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

Conditional Moves

■ Conditional move instructions

- if (Test) Dest \leftarrow 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

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

x in %rdi
y in %rsi

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

“Do-While” Loop (I)

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

C Code

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

Goto Version

```
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

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

Registers

%rdi	x
%rax	result

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
```

“Do-While” Loop (3)

- General “Do-While” translation

C Code

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

Goto Version

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

- *Body* can be any C statement
 - Typically compound statement:
- *Test* is expression returning integer:
 - = 0 interpreted as false, $\neq 0$ interpreted as true

```
{  
  Statement1;  
  Statement2;  
  ...  
  Statementn;  
}
```

“While” Loop (I)

C Code

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

First Goto Version

```
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

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

- Historically used by GCC
- Uses same inner loop as do-while version
- Guards loop entry with extra test

Second Goto Version

```
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;
}
```

“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 (I)

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

Example:

$$3^{10} = 3^2 * 3^8 = 3^2 * ((3^2)^2)^2$$

```
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;
}
```

“For” Loop (2)

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

General Form

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

Init

```
result = 1
```

Test

```
p != 0
```

Update

```
p = p >> 1
```

Body

```
{  
    if (p & 0x1)  
        result *= x;  
    x = x*x;  
}
```

“For” Loop (3)

For Version

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

While Version

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

Do-While Version

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

Goto Version

```
Init;  
if (!Test)  
  goto done;  
loop:  
  Body  
  Update;  
  if (Test)  
    goto loop;  
done:
```

“For” Loop (4)

Goto Version

```
Init;  
if (!Test)  
    goto done;  
Loop:  
    Body  
    Update;  
    if (Test)  
        goto Loop;  
done:
```



```
result = 1;  
if (p == 0)  
    goto done;  
Loop:  
    if (p & 0x1)  
        result *= x;  
    x = x*x;  
    p = p >> 1;  
    if (p != 0)  
        goto Loop;  
done:
```

Init

```
result = 1
```

Test

```
p != 0
```

Update

```
p = p >> 1
```

Body

```
{  
    if (p & 0x1)  
        result *= x;  
    x = x*x;  
}
```

“Switch” Implementation

- Series of conditionals
 - Good if few cases
 - Slow if many
- Jump table (see textbook)
 - Lookup branch target and perform indirect jump
 - Avoids conditionals
 - Possible when cases are small integer constants
- Binary search tree
 - For sparse cases
 - Logarithmic performance

```
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