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

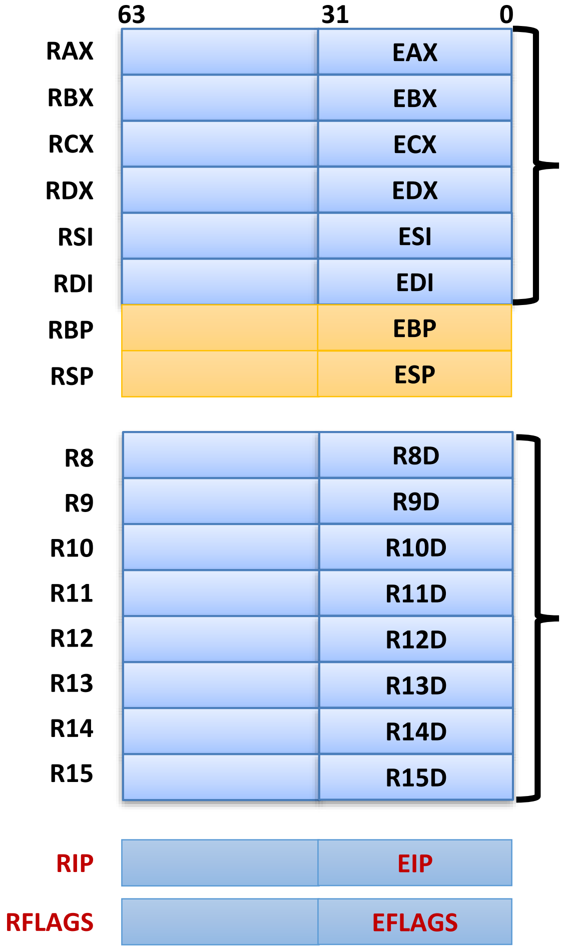
Systems Software
Research Lab.

Seoul National University

Spring 2018

Assembly II: Control Flow

Processor State (x86-64)



General-purpose registers (temporary data)

Current stack frame (base pointer)

Current stack top (stack pointer)

General-purpose registers (temporary data)

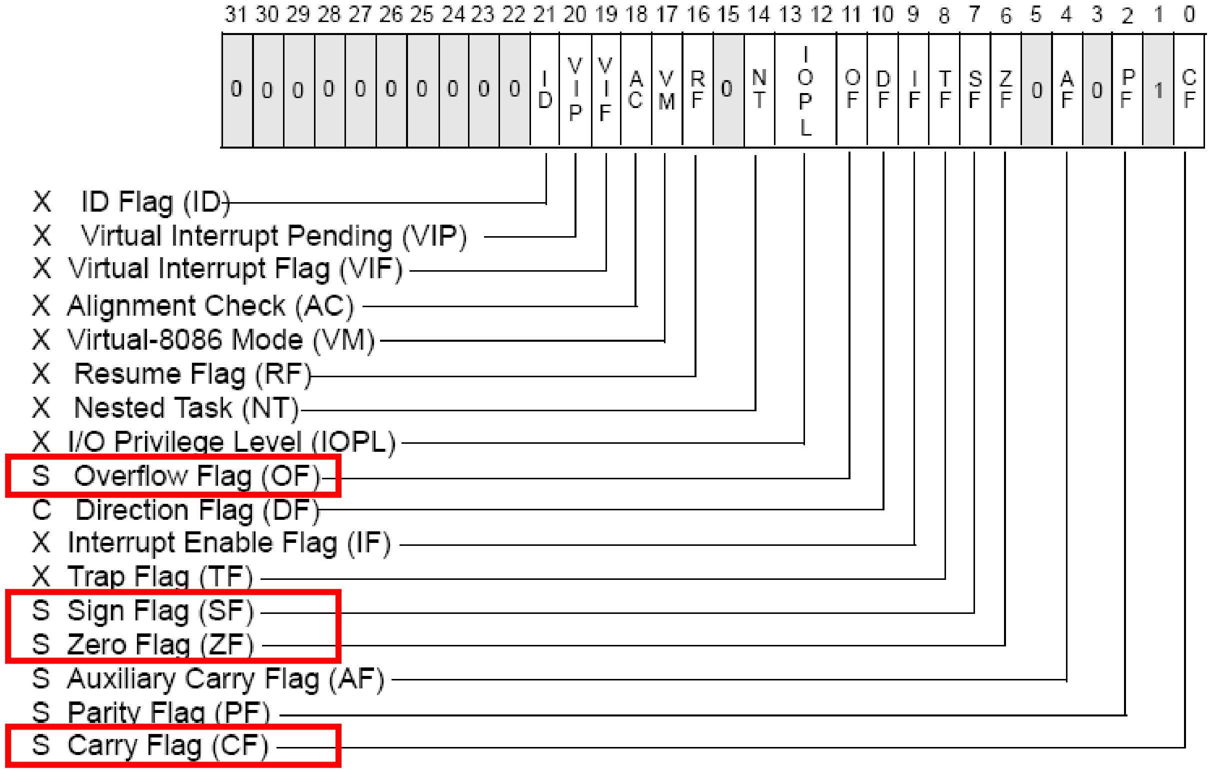
Instruction Pointer (location of the next instruction)

Condition codes (status of recent tests)

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



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	ID	VIP	VIF	AC	VM	RF	0	NT	IOPL		OF	DF	IF	TF	SF	ZF	0	AF	0	PF	1	CF

X ID Flag (ID)
X Virtual Interrupt Pending (VIP)
X Virtual Interrupt Flag (VIF)
X Alignment Check (AC)
X Virtual-8086 Mode (VM)
X Resume Flag (RF)
X Nested Task (NT)
X I/O Privilege Level (IOPL)
S Overflow Flag (OF)
C Direction Flag (DF)
X Interrupt Enable Flag (IF)
X Trap Flag (TF)
S Sign Flag (SF)
S Zero Flag (ZF)
S Auxiliary Carry Flag (AF)
S Parity Flag (PF)
S 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
- 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`	~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 >)
`jae`	~CF	Above or Equal (Unsigned >=)
`jb`	CF	Below (Unsigned <)
`jbe`	CF | ZF	Below or Equal (Unsigned <=)

Conditional Branch Example (1)

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

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

- 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;
}
```

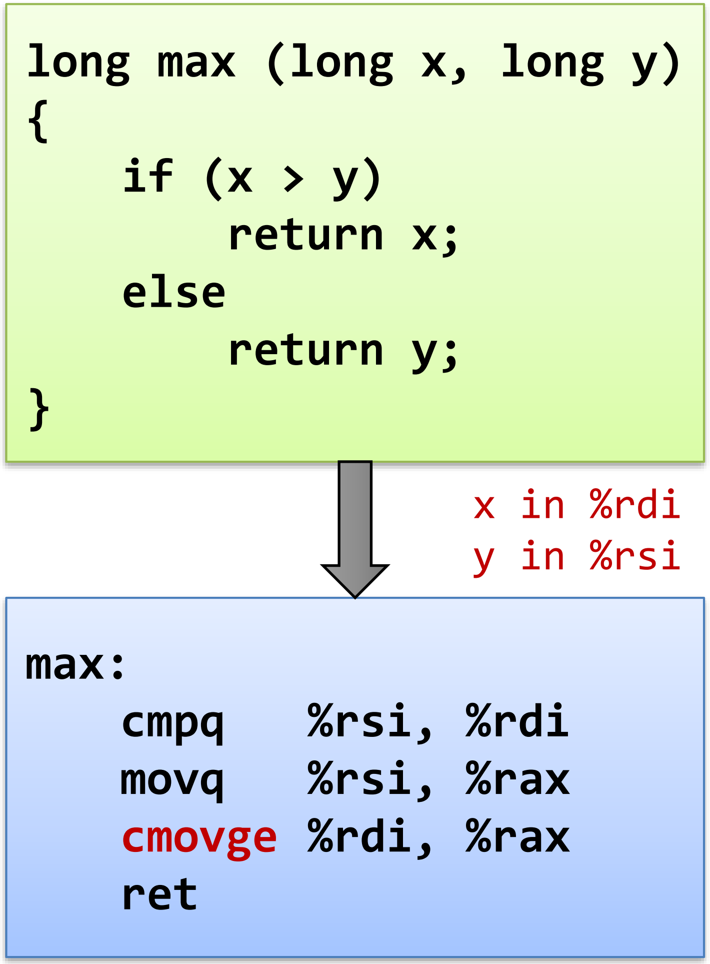
```
x in %rdi
y in %rsi
```

```
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 ← 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
```

Bad Cases for Conditional Moves

- Expensive computations

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

  - Only makes sense when computations are very simple

- Risky computations

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

  - May have undesirable effects

- Computations with side effects

```
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

```
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:

```
{
    $Statement_1$;
    $Statement_2$;
      …
    $Statement_n$;
}
```

- ***Test*** is expression returning integer:

  = 0 interpreted as false, ≠ 0 interpreted as true

“While” Loop (1)

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;
}
```

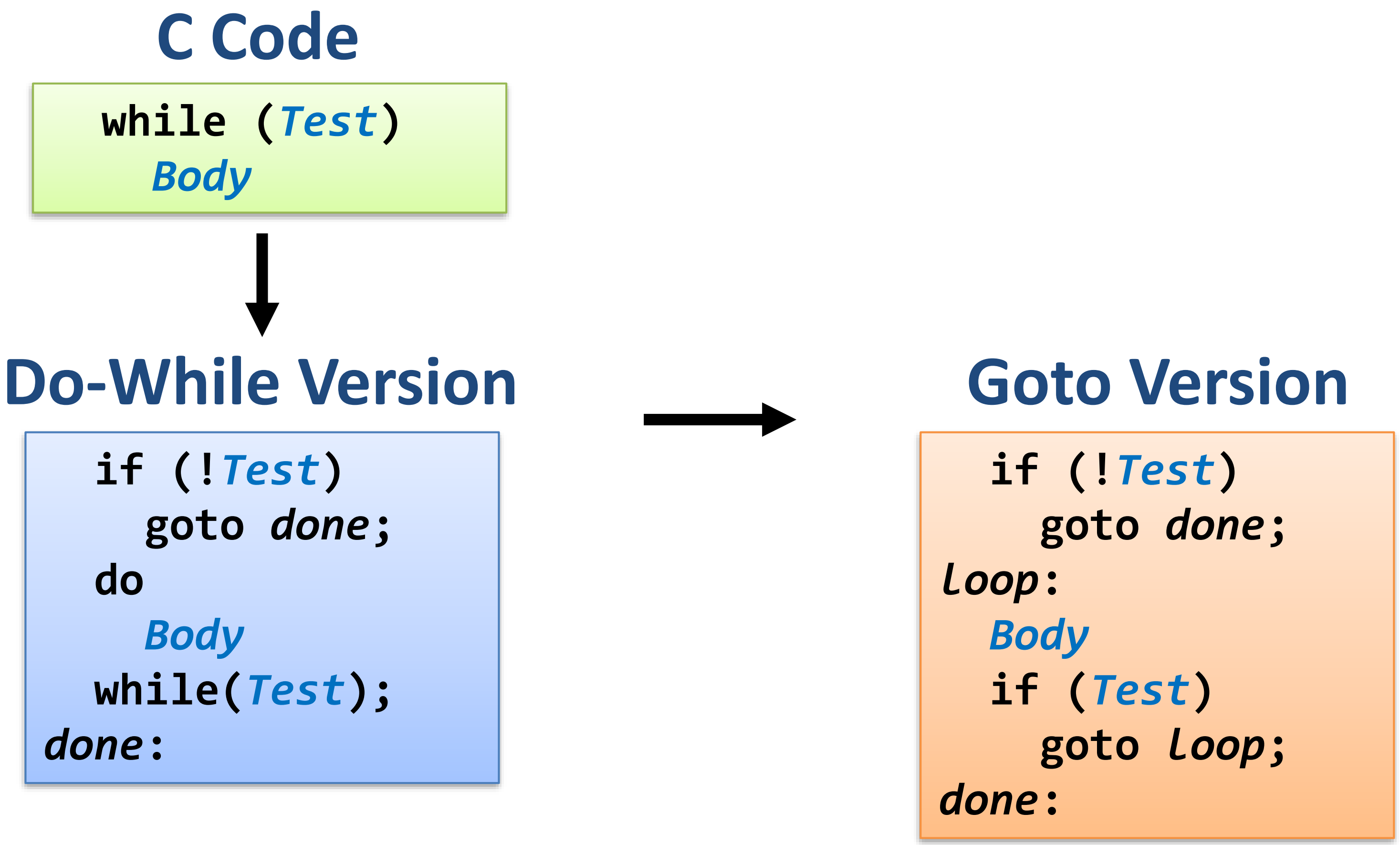
Second Goto Version

```
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;
}
```

- 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 $p = p_0 + 2p_1 + 4p_2 + \ldots + 2^{n-1}p_{n-1}$
  - Gives: $x^p = z_0 \cdot z_1{}^2 \cdot (z_2{}^2)^2 \cdot \ldots \cdot (\ldots((z_{n-1}{}^2)^2)\ldots)^2$
    - $z_i = 1$ 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
```

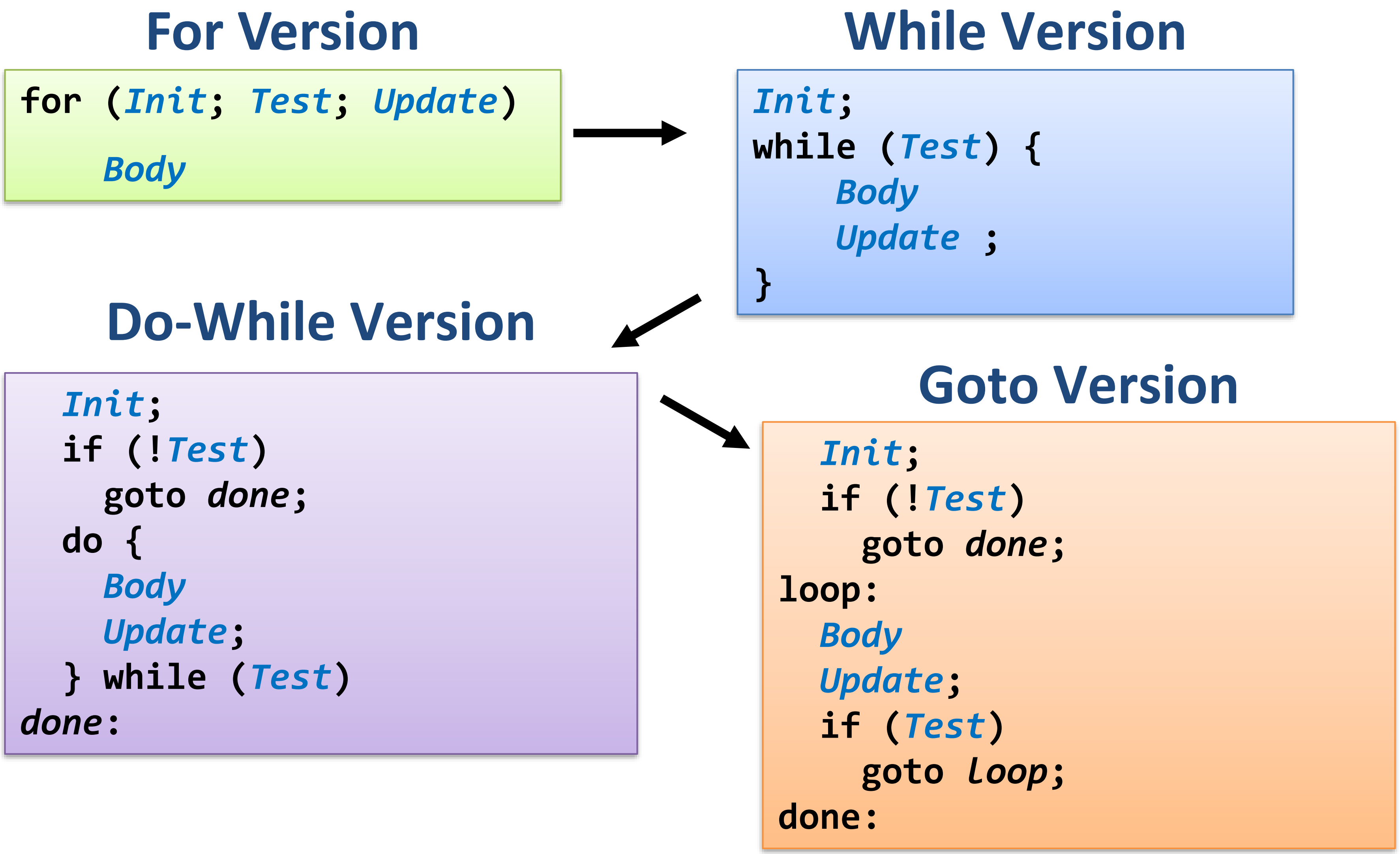
Update

```
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
  - 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