CS 33

Machine Programming (3)

Swapxy for Ints

```
struct xy {
  int x;
  int y;
}
void swapxy(struct xy *p){
  int temp = p->x;
  p->x = p->y;
  p->y = temp;
}
```

```
swap:
  movl (%rdi), %eax
  movl 4(%rdi), %edx
  movl %edx, (%rdi)
  movl %eax, 4(%rdi)
  ret
```

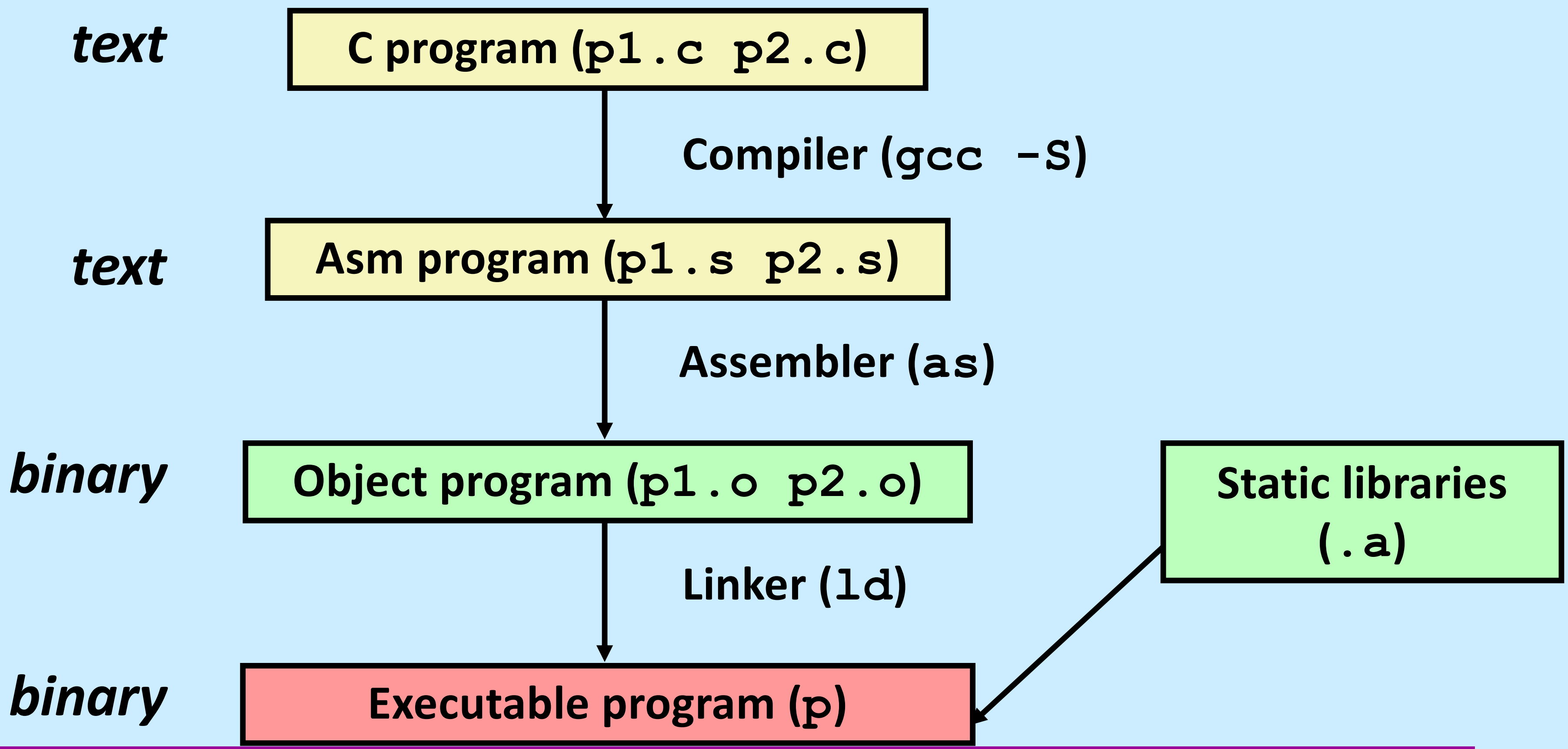
- **Pointers are 64 bits**
- **What they point to are 32 bits**

Bytes

- **Each register has a byte version**
  - e.g., %r10: %r10b; see earlier slide for x86 registers
- **Needed for byte instructions**
  - movb (%rax, %rsi), %r10b
  - sets *only* the low byte in %r10
    - » other seven bytes are unchanged
- **Alternatives**
  - movzbq (%rax, %rsi), %r10
    - » copies byte to low byte of %r10
    - » zeroes go to higher bytes
  - movsbq (%rax, %rsi), %r10
    - » copies byte to low byte of %r10
    - » sign is extended to all higher bits

Turning C into Object Code

- Code in files `p1.c p2.c`
- Compile with command: `gcc -O1 p1.c p2.c -o p`
  - » use basic optimizations (`-O1`)
  - » put resulting binary in file `p`



Example

```
long ASum(long *a, unsigned long size) {
    long i, sum = 0;
    for (i=0; i<size; i++)
        sum += a[i];
    return sum;
}

int main() {
    long array[3] = {2,117,-6};
    long sum = ASum(array, 3);
    return sum;
}
```

Assembler Code

```
ASum:
        testq   %rsi, %rsi
        je      .L4
        movq    %rdi, %rax
        leaq    (%rdi,%rsi,8), %rcx
        movl    $0, %edx
.L3:
        addq    (%rax), %rdx
        addq    $8, %rax
        cmpq    %rcx, %rax
        jne     .L3
.L1:
        movq    %rdx, %rax
        ret
.L4:
        movl    $0, %edx
        jmp     .L1
```

```
main:
        subq    $32, %rsp
        movq    $2, (%rsp)
        movq    $117, 8(%rsp)
        movq    $-6, 16(%rsp)
        movq    %rsp, %rdi
        movl    $3, %esi
        call    ASum
        addq    $32, %rsp
        ret
```

Object Code

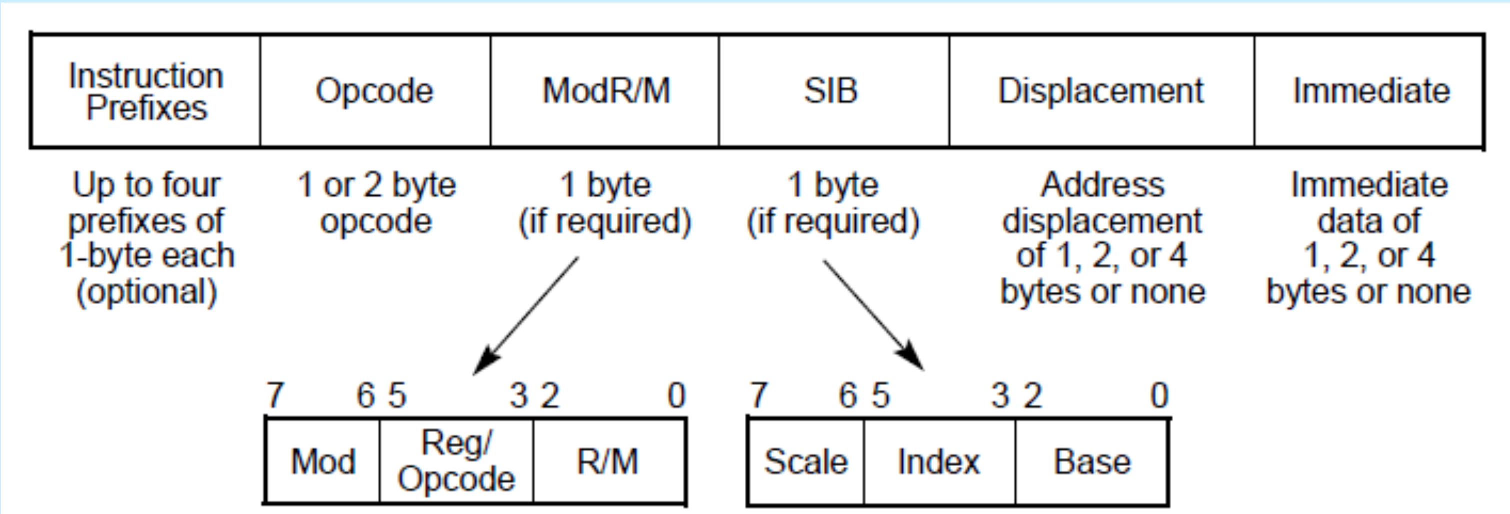
Code for `ASum`

```
0x1125 <ASum>:
   0x48
   0x85
   0xf6
   0x74
   0x1c
   0x48
   0x89
   0xf8
   0x48
   0x8d
   0x0c
   0xf7
    .
    .
    .
```

- Total of 39 bytes
- Each instruction: 1, 2, or 3 bytes
- Starts at address `0x1125`

- **Assembler**
  - translates `.s` into `.o`
  - binary encoding of each instruction
  - nearly complete image of executable code
  - missing linkages between code in different files
- **Linker**
  - resolves references between files
  - combines with static run-time libraries
    - » e.g., code for `printf`
  - some libraries are *dynamically linked*
    - » linking occurs when program begins execution

Instruction Format



Disassembling Object Code

Disassembled

```
0000000000001125 <ASum>:
    1125:       48 85 f6                test   %rsi,%rsi
    1128:       74 1c                   je     1146 <ASum+0x21>
    112a:       48 89 f8                mov    %rdi,%rax
    112d:       48 8d 0c f7             lea    (%rdi,%rsi,8),%rcx
    1131:       ba 00 00 00 00          mov    $0x0,%edx
    1136:       48 03 10                add    (%rax),%rdx
    1139:       48 83 c0 08             add    $0x8,%rax
    113d:       48 39 c8                cmp    %rcx,%rax
    1140:       75 f4                   jne    1136 <ASum+0x11>
    1142:       48 89 d0                mov    %rdx,%rax
    1145:       c3                      retq
    1146:       ba 00 00 00 00          mov    $0x0,%edx
    114b:       eb f5                   jmp    1142 <ASum+0x1d>
```

- **Disassembler**

  `objdump -d <file>`

  – useful tool for examining object code

  – produces approximate rendition of assembly code

Alternate Disassembly

Object

```
0x1125:
    0x48
    0x85
    0xf6
    0x74
    0x1c
    0x48
    0x89
    0xf8
    0x48
    0x8d
    0x0c
    0xf7
      .
      .
      .
```

Disassembled

```
Dump of assembler code for function ASum:
   0x1125 <+0>:     test   %rsi,%rsi
   0x1128 <+3>:     je     0x1146 <ASum+33>
   0x112a <+5>:     mov    %rdi,%rax
   0x112d <+8>:     lea    (%rdi,%rsi,8),%rcx
   0x1131 <+12>:    mov    $0x0,%edx
     ...
```

- **Within gdb debugger**

  `gdb <file>`

  `disassemble ASum`

  – disassemble the ASum object code

  `x/39xb ASum`

  – examine the 39 bytes starting at `ASum`

How Many Instructions are There?

- We cover ~30
- Implemented by Intel:
  - 80 in original 8086 architecture
  - 7 added with 80186
  - 17 added with 80286
  - 33 added with 386
  - 6 added with 486
  - 6 added with Pentium
  - 1 added with Pentium MMX
  - 4 added with Pentium Pro
  - 8 added with SSE
  - 8 added with SSE2
  - 2 added with SSE3
  - 14 added with x86-64
  - 10 added with VT-x
  - 2 added with SSE4a
- Total: 198
- Doesn't count:
  - floating-point instructions
    - ~100
  - SIMD instructions
    - lots
  - AMD-added instructions
  - undocumented instructions

Some Arithmetic Operations

- **Two-operand instructions:**

Format		Computation	
`addl`	Src,Dest	Dest = Dest + Src	
`subl`	Src,Dest	Dest = Dest – Src	
`imull`	Src,Dest	Dest = Dest * Src	
`shll`	Src,Dest	Dest = Dest << Src	Also called sall
`sarl`	Src,Dest	Dest = Dest >> Src	Arithmetic
`shrl`	Src,Dest	Dest = Dest >> Src	Logical
`xorl`	Src,Dest	Dest = Dest ^ Src	
`andl`	Src,Dest	Dest = Dest & Src	
`orl`	Src,Dest	Dest = Dest | Src	

  – watch out for argument order!

Some Arithmetic Operations

- One-operand Instructions

`incl`	Dest	= Dest + 1
`decl`	Dest	= Dest – 1
`negl`	Dest	= – Dest
`notl`	Dest	= ~Dest

- See textbook for more instructions
- See Intel documentation for even more

Quiz 1

- **What is the value stored in %r9 after the following code is execute?**

```
movq $8, %r8
movq $9, %r9
addq %r9, %r8
addq %r8, %r9
addq %r9, %r8
```

a) 17
b) 26
c) 42
d) 43

Arithmetic Expression Example

```
int arith(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;
}
```

```
arith:
    leal   (%rdi,%rsi), %eax
    addl   %edx, %eax
    leal   (%rsi,%rsi,2), %edx
    shll   $4, %edx
    leal   4(%rdi,%rdx), %ecx
    imull  %ecx, %eax
    ret
```

Understanding `arith`

```
int arith(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;
}
```

%rdx	z
%rsi	y
%rdi	x

```
leal   (%rdi,%rsi), %eax
addl   %edx, %eax
leal   (%rsi,%rsi,2), %edx
shll   $4, %edx
leal   4(%rdi,%rdx), %ecx
imull  %ecx, %eax
ret
```

Understanding arith

```
int arith(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;
}
```

%rdx	z
%rsi	y
%rdi	x

```
leal   (%rdi,%rsi), %eax     # eax = x+y      (t1)
addl   %edx, %eax            # eax = t1+z     (t2)
leal   (%rsi,%rsi,2), %edx   # edx = 3*y      (t4)
shll   $4, %edx              # edx = t4*16    (t4)
leal   4(%rdi,%rdx), %ecx    # ecx = x+4+t4   (t5)
imull  %ecx, %eax            # eax *= t5      (rval)
ret
```

Observations about `arith`

```
int arith(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;
}
```

- Instructions in different order from C code
- Some expressions might require multiple instructions
- Some instructions might cover multiple expressions

```
leal   (%rdi,%rsi), %eax     # eax = x+y     (t1)
addl   %edx, %eax            # eax = t1+z    (t2)
leal   (%rsi,%rsi,2), %edx   # edx = 3*y     (t4)
shll   $4, %edx              # edx = t4*16   (t4)
leal   4(%rdi,%rdx), %ecx    # ecx = x+4+t4  (t5)
imull  %ecx, %eax            # eax *= t5     (rval)
ret
```

Another Example

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

$2^{13} = 8192$, $2^{13} – 7 = 8185$

```
xorl %esi, %edi        # edi = x^y          (t1)
sarl $17, %edi         # edi = t1>>17       (t2)
movl %edi, %eax        # eax = edi
andl $8185, %eax       # eax = t2 & mask (rval)
ret
```

Processor State (x86-64, Partial)

	%rax	%eax	%r8	%r8d	a5
	%rbx	%ebx	%r9	%r9d	a6
a4	%rcx	%ecx	%r10	%r10d	
a3	%rdx	%edx	%r11	%r11d	
a2	%rsi	%esi	%r12	%r12d	
a1	%rdi	%edi	%r13	%r13d	
	%rsp	%esp	%r14	%r14d	
	%rbp	%ebp	%r15	%r15d	

%rip

CF	ZF	SF	OF

condition codes

Condition Codes (Implicit Setting)

- **Single-bit registers**

CF	**carry flag (for unsigned)**	**SF**	**sign flag (for signed)**
ZF	**zero flag**	**OF**	**overflow flag (for signed)**

- **Implicitly set (think of it as side effect) by arithmetic operations**

  example: *addl/addq* Src,Dest ↔ `t = a+b`

  CF set if carry out from most significant bit or borrow (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)`

- **Not set by `lea` instruction**

Condition Codes (Explicit Setting: Compare)

- **Explicit setting by compare instruction**

  `cmpl/cmpq` **src2, src1**

  compares src1:src2

  `cmpl b,a` **like computing** `a-b` **without setting destination**

  CF set if carry out from most significant bit or borrow (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 test instruction**

  `testl/testq` src2, src1

  `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`

Quiz 2

- The following code is executed:

```
movq $10, %r8
movq $-11, %r9
cmpq %r8, %r9
```

Which of the condition codes will be 1?

a) CF, SF, ZF, OF
b) SF, OF
c) SF
d) CF, OF

Reading Condition Codes

- **SetX instructions**
  - **set single byte (to 1 or 0) based on combinations of condition codes**

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

Reading Condition Codes (Cont.)

- **SetX instructions:**
  - set single byte based on combination of condition codes
- **Uses byte registers**
  - does not alter remaining 7 bytes
  - typically use `movzbl` to finish job

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

%rax	%eax	%ah	%al

Body

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

Jumping

- **jX instructions**
  - Jump to different part of program 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)
`jb`	`CF`	Below (unsigned)

Conditional-Branch Example

```c
int absdiff(int x, int y)
{
    int result;
    if (x > y) {
      result = x-y;
    } else {
      result = y-x;
    }
    return result;
}
```

```
absdiff:
    movl    %esi, %eax
    cmpl    %esi, %edi          Body1
    jle     .L6
    subl    %eax, %edi
    movl    %edi, %eax          Body2a
    jmp     .L7
.L6:
    subl %edi, %eax             Body2b
.L7:
    ret
```

x in %edi

y in %esi

Conditional-Branch Example (Cont.)

```
int goto_ad(int x, int y)
{
  int result;
  if (x <= y) goto Else;
  result = x-y;
  goto Exit;
Else:
  result = y-x;
Exit:
  return result;
}
```

```
absdiff:
    movl    %esi, %eax
    cmpl    %esi, %edi
    jle     .L6
    subl    %eax, %edi
    movl    %edi, %eax
    jmp .L7
.L6:
    subl %edi, %eax
.L7:
    ret
```

Body1

Body2a

Body2b

- C allows "goto" as means of transferring control
  - closer to machine-level programming style
- Generally considered bad coding style

General Conditional-Expression Translation

C Code

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

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

- <u>Test</u> is expression returning integer
  - == 0 interpreted as false
  - ≠ 0 interpreted as true
- Create separate code regions for <u>then</u> and <u>else</u> expressions
- Execute appropriate one

Goto Version

```
  nt = !Test;
  if (nt) goto Else;
  val = Then_Expr;
  goto Done;
Else:
  val = Else_Expr;
Done:
  . . .
```

"Do-While" Loop Example

C Code

```
int pcount_do(unsigned x)
{
  int result = 0;
  do {
    result += x & 0x1;
    x >>= 1;
  } while (x);
  return result;
}
```

Goto Version

```
int pcount_do(unsigned x)
{
  int result = 0;
loop:
  result += x & 0x1;
  x >>= 1;
  if (x)
    goto loop;
  return result;
}
```

- **Count number of 1's in argument x ("popcount")**
- **Use conditional branch either to continue looping or to exit loop**

“Do-While” Loop Compilation

Goto Version

```
int pcount_do(unsigned x)  {
  int result = 0;
loop:
  result += x & 0x1;
  x >>= 1;
  if (x)
    goto loop;
  return result;
}
```

Registers:

%edi	x
%eax	result

```
  movl   $0, %eax      #   result = 0
.L2:     # loop:
  movl   %edi, %ecx
  andl   $1, %ecx      #   t = x & 1
  addl   %ecx, %eax    #   result += t
  shrl   $1, %edi      #   x >>= 1
  jne    .L2           #   if !0, goto loop
```

General "Do-While" Translation

C Code

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

Goto Version

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

- **Body:**

```
{
   Statement1;
   Statement2;
     …
   Statementn;
}
```

- **Test returns integer**
  - **= 0 interpreted as false**
  - **≠ 0 interpreted as true**

"While" Loop Example

C Code

```
int pcount_while(unsigned x) {
  int result = 0;
  while (x) {
    result += x & 0x1;
    x >>= 1;
  }
  return result;
}
```

Goto Version

```
int pcount_do(unsigned x)  {
  int result = 0;
  if (!x) goto done;
loop:
  result += x & 0x1;
  x >>= 1;
  if (x)
    goto loop;
done:
  return result;
}
```

- **Is this code equivalent to the do-while version?**
  - **must jump out of loop if test fails**

General "While" Translation

While version

```
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 Example

C Code

```
#define WSIZE 8*sizeof(int)
int pcount_for(unsigned x) {
  int i;
  int result = 0;
  for (i = 0; i < WSIZE; i++) {
    unsigned mask = 1 << i;
    result += (x & mask) != 0;
  }
  return result;
}
```

- **Is this code equivalent to other versions?**

"For" Loop Form

General Form

```
for (Init; Test; Update )

    Body
```

```
for (i = 0; i < WSIZE; i++) {
    unsigned mask = 1 << i;
    result += (x & mask) != 0;
}
```

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{
  unsigned mask = 1 << i;
  result += (x & mask) != 0;
}
```

"For" Loop → While Loop

For Version

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

While Version

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

"For" Loop → … → Goto

For Version

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

While Version

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

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

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

"For" Loop Conversion Example

C Code

```
#define WSIZE 8*sizeof(int)
int pcount_for(unsigned x) {
  int i;
  int result = 0;
  for (i = 0; i < WSIZE; i++) {
    unsigned mask = 1 << i;
    result += (x & mask) != 0;
  }
  return result;
}
```

- Initial test can be optimized away

Goto Version

```
int pcount_for_gt(unsigned x) {
  int i;
  int result = 0;
  i = 0;                          Init
  if (!(i < WSIZE))               !Test
    goto done;
 loop:
  {                               Body
    unsigned mask = 1 << i;
    result += (x & mask) != 0;
  }
  i++;  Update
  if (i < WSIZE)  Test
    goto loop;
 done:
  return result;
}
```