# Machine-Level Programming II: Arithmetic & Control

15-213 / 18-213: Introduction to Computer Systems
6th Lecture, Feb. 2, 2012

**Instructors:**
Todd C. Mowry & Anthony Rowe





## Today

- **Complete addressing mode, address computation (leal)**
- Arithmetic operations
- Control: Condition codes
- Conditional branches
- While loops





## Complete Memory Addressing Modes

- **Most General Form**
- **D(Rb,Ri,S)  Mem[Reg[Rb]+S*Reg[Ri]+ D]**
  - D: Constant "displacement" 1, 2, or 4 bytes
  - Rb: Base register: Any of 8 integer registers
  - Ri: Index register: Any, except for **%esp**
    - Unlikely you'd use **%ebp**, either
  - S: Scale: 1, 2, 4, or 8 (***why these numbers?***)

- **Special Cases**
- **(Rb,Ri)  Mem[Reg[Rb]+Reg[Ri]]**
- **D(Rb,Ri)  Mem[Reg[Rb]+Reg[Ri]+D]**
- **(Rb,Ri,S)  Mem[Reg[Rb]+S*Reg[Ri]]**





## Address Computation Examples

| | |
|---|---|
| `%edx` | `0xf000` |
| `%ecx` | `0x0100` |

| Expression | Address Computation | Address |
|---|---|---|
| `0x8(%edx)` | | |
| `(%edx,%ecx)` | | |
| `(%edx,%ecx,4)` | | |
| `0x80(,%edx,2)` | | |

## Address Computation Instruction

- **`leal` *Src,Dest***
  - *Src* is address mode expression
  - Set *Dest* to address denoted by expression
- **Uses**
  - Computing addresses without a 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**

```
int mul12(int x)
{
  return x*12;
}
```

Converted to ASM by compiler:

```
leal (%eax,%eax,2), %eax  ;t <- x+x*2
sall $2, %eax             ;return t<<2
```





## Today

- Complete addressing mode, address computation (leal)
- **Arithmetic operations**
- Control: Condition codes
- Conditional branches
- While loops





## 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 | |
| `sall` | *Src,Dest* | Dest = Dest << Src | ***Also called shll*** |
| `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!**
- **No distinction between signed and unsigned int (why?)**





## Some Arithmetic Operations

- **One Operand Instructions**

| | | |
|---|---|---|
| `incl` | *Dest* | *Dest = Dest + 1* |
| `decl` | *Dest* | *Dest = Dest – 1* |
| `negl` | *Dest* | *Dest = – Dest* |
| `notl` | *Dest* | *Dest = ~Dest* |

- **See book for more instructions**

## 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:
  pushl  %ebp
  movl   %esp, %ebp

  movl   8(%ebp), %ecx
  movl   12(%ebp), %edx
  leal   (%edx,%edx,2), %eax
  sall   $4, %eax
  leal   4(%ecx,%eax), %eax
  addl   %ecx, %edx
  addl   16(%ebp), %edx
  imull  %edx, %eax

  popl   %ebp
  ret
```

Set Up

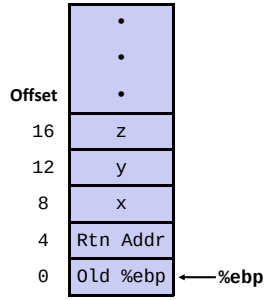
Body

Finish



## 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;
}
```

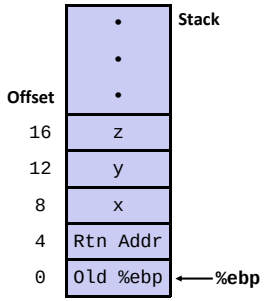



```
  movl   8(%ebp), %ecx
  movl   12(%ebp), %edx
  leal   (%edx,%edx,2), %eax
  sall   $4, %eax
  leal   4(%ecx,%eax), %eax
  addl   %ecx, %edx
  addl   16(%ebp), %edx
  imull  %edx, %eax
```



## 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;
}
```




```
  movl   8(%ebp), %ecx        # ecx = x
  movl   12(%ebp), %edx       # edx = y
  leal   (%edx,%edx,2), %eax  # eax = y*3
  sall   $4, %eax             # eax *= 16 (t4)
  leal   4(%ecx,%eax), %eax   # eax = t4 +x+4 (t5)
  addl   %ecx, %edx           # edx = x+y (t1)
  addl   16(%ebp), %edx       # edx += z (t2)
  imull  %edx, %eax           # eax = t2 * t5 (rval)
```



## 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 require multiple instructions
- Some instructions cover multiple expressions
- Get exact same code when compile:
- **`(x+y+z)*(x+4+48*y)`**

```
  movl   8(%ebp), %ecx        # ecx = x
  movl   12(%ebp), %edx       # edx = y
  leal   (%edx,%edx,2), %eax  # eax = y*3
  sall   $4, %eax             # eax *= 16 (t4)
  leal   4(%ecx,%eax), %eax   # eax = t4 +x+4 (t5)
  addl   %ecx, %edx           # edx = x+y (t1)
  addl   16(%ebp), %edx       # edx += z (t2)
  imull  %edx, %eax           # eax = t2 * t5 (rval)
```

## 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;
}
```

```
logical:
    pushl %ebp
    movl %esp,%ebp

    movl 12(%ebp),%eax
    xorl 8(%ebp),%eax
    sarl $17,%eax
    andl $8185,%eax

    popl %ebp
    ret
```

Set Up

Body

Finish

```
    movl 12(%ebp),%eax    # eax = y
    xorl 8(%ebp),%eax     # eax = x^y       (t1)
    sarl $17,%eax         # eax = t1>>17    (t2)
    andl $8185,%eax       # eax = t2 & mask (rval)
```



## 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;
}
```

```
logical:
    pushl %ebp
    movl %esp,%ebp

    movl 12(%ebp),%eax
    xorl 8(%ebp),%eax
    sarl $17,%eax
    andl $8185,%eax

    popl %ebp
    ret
```

Set Up

Body

Finish

```
    movl 12(%ebp),%eax    # eax = y
    xorl 8(%ebp),%eax     # eax = x^y       (t1)
    sarl $17,%eax         # eax = t1>>17    (t2)
    andl $8185,%eax       # eax = t2 & mask (rval)
```



## 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;
}
```

```
logical:
    pushl %ebp
    movl %esp,%ebp

    movl 12(%ebp),%eax
    xorl 8(%ebp),%eax
    sarl $17,%eax
    andl $8185,%eax

    popl %ebp
    ret
```

Set Up

Body

Finish

```
    movl 12(%ebp),%eax    # eax = y
    xorl 8(%ebp),%eax     # eax = x^y       (t1)
    sarl $17,%eax         # eax = t1>>17    (t2)
    andl $8185,%eax       # eax = t2 & mask (rval)
```



## 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;
}
```

```
logical:
    pushl %ebp
    movl %esp,%ebp

    movl 12(%ebp),%eax
    xorl 8(%ebp),%eax
    sarl $17,%eax
    andl $8185,%eax

    popl %ebp
    ret
```

Set Up

Body

Finish

$2^{13} = 8192$, $2^{13} - 7 = 8185$

```
    movl 12(%ebp),%eax    # eax = y
    xorl 8(%ebp),%eax     # eax = x^y       (t1)
    sarl $17,%eax         # eax = t1>>17    (t2)
    andl $8185,%eax       # eax = t2 & mask (rval)
```

## Today

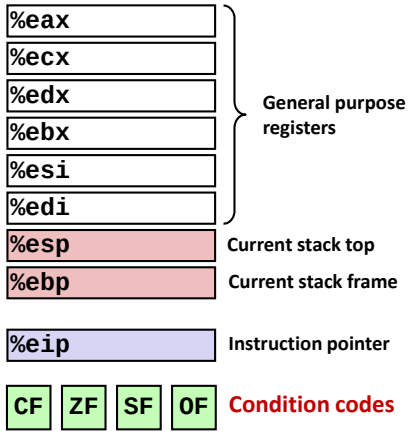
- Complete addressing mode, address computation (leal)
- Arithmetic operations
- **Control: Condition codes**
- Conditional branches
- Loops





## Processor State (IA32, Partial)

- **Information about currently executing program**
  - Temporary data ( **%eax**, … )
  - Location of runtime stack ( **%ebp**,**%esp** )
  - Location of current code control point ( **%eip**, … )
  - Status of recent tests ( **CF, ZF, SF, OF** )








## 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 (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**
- **Full documentation (IA32), link on course website**





## Condition Codes (Explicit Setting: Compare)

- **Explicit Setting by Compare Instruction**
  - **`cmpl/cmpq`** *Src2, Src1*
  - **`cmpl b,a`** like computing **`a-b`** without setting destination

  **CF set** if carry out from most significant bit (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`**





# Reading Condition Codes

- **SetX Instructions**
  - Set single byte 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
- **One of 8 addressable byte registers**
  - Does not alter remaining 3 bytes
  - Typically use **`movzbl`** to finish job

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

**Body**

```
movl 12(%ebp),%eax   # eax = y
cmpl %eax,8(%ebp)    # Compare x : y
setg %al             # al = x > y
movzbl %al,%eax      # Zero rest of %eax
```

| | | |
|---|---|---|
| `%eax` | `%ah` | `%al` |
| `%ecx` | `%ch` | `%cl` |
| `%edx` | `%dh` | `%dl` |
| `%ebx` | `%bh` | `%bl` |
| `%esi` | | |
| `%edi` | | |
| `%esp` | | |
| `%ebp` | | |





# Reading Condition Codes: x86-64

- **SetX Instructions:**
  - Set single byte based on combination of condition codes
  - Does not alter remaining 3 bytes

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

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

**Bodies**

```
cmpl %esi, %edi
setg %al
movzbl %al, %eax
```

```
cmpq %rsi, %rdi
setg %al
movzbl %al, %eax
```

Is **`%rax`** zero?
Yes: 32-bit instructions set high order 32 bits to 0!

## Today

- Complete addressing mode, address computation (leal)
- Arithmetic operations
- x86-64
- Control: Condition codes
- **Conditional branches & Moves**
- Loops





## Jumping

- **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) |
| jb | CF | Below (unsigned) |





## Conditional Branch Example

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

```
absdiff:
    pushl  %ebp              } Setup
    movl   %esp, %ebp        }
    movl   8(%ebp), %edx     } Body1
    movl   12(%ebp), %eax    }
    cmpl   %eax, %edx        }
    jle    .L6               }
    subl   %eax, %edx        } Body2a
    movl   %edx, %eax        }
    jmp .L7                  }
.L6:
    subl %edx, %eax          } Body2b
.L7:
    popl %ebp                } Finish
    ret                      }
```





## 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;
}
```

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

```
absdiff:
    pushl  %ebp              } Setup
    movl   %esp, %ebp        }
    movl   8(%ebp), %edx     } Body1
    movl   12(%ebp), %eax    }
    cmpl   %eax, %edx        }
    jle    .L6               }
    subl   %eax, %edx        } Body2a
    movl   %edx, %eax        }
    jmp .L7                  }
.L6:
    subl %edx, %eax          } Body2b
.L7:
    popl %ebp                } Finish
    ret                      }
```

## 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:
    pushl  %ebp              } Setup
    movl   %esp, %ebp        }
    movl   8(%ebp), %edx     } Body1
    movl   12(%ebp), %eax    }
    cmpl   %eax, %edx        }
    jle    .L6               }
    subl   %eax, %edx        } Body2a
    movl   %edx, %eax        }
    jmp .L7                  }
.L6:
    subl %edx, %eax          } Body2b
.L7:
    popl %ebp                } Finish
    ret                      }
```





## 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:
    pushl  %ebp              } Setup
    movl   %esp, %ebp        }
    movl   8(%ebp), %edx     } Body1
    movl   12(%ebp), %eax    }
    cmpl   %eax, %edx        }
    jle    .L6               }
    subl   %eax, %edx        } Body2a
    movl   %edx, %eax        }
    jmp .L7                  }
.L6:
    subl %edx, %eax          } Body2b
.L7:
    popl %ebp                } Finish
    ret                      }
```





## 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:
    pushl  %ebp              } Setup
    movl   %esp, %ebp        }
    movl   8(%ebp), %edx     } Body1
    movl   12(%ebp), %eax    }
    cmpl   %eax, %edx        }
    jle    .L6               }
    subl   %eax, %edx        } Body2a
    movl   %edx, %eax        }
    jmp .L7                  }
.L6:
    subl %edx, %eax          } Body2b
.L7:
    popl %ebp                } Finish
    ret                      }
```





## General Conditional Expression Translation

**C Code**

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

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

**Goto Version**

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

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

## Using Conditional Moves

- **Conditional Move Instructions**
  - Instruction supports:
    if (Test) Dest ← Src
  - Supported in post-1995 x86 processors
  - GCC does not always use them
    - Wants to preserve compatibility with ancient processors
    - Enabled for x86-64
    - Use switch `–march=686` for IA32
- **Why?**
  - Branches are very disruptive to instruction flow through pipelines
  - Conditional move do not require control transfer

**C Code**

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

**Goto Version**

```
  tval = Then_Expr;
  result = Else_Expr;
  t = Test;
  if (t) result = tval;
  return result;
```





## Conditional Move Example: x86-64

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

x in %edi

y in %esi

```
absdiff:
  movl   %edi, %edx
  subl   %esi, %edx   # tval = x-y
  movl   %esi, %eax
  subl   %edi, %eax   # result = y-x
  cmpl   %esi, %edi   # Compare x:y
  cmovg  %edx, %eax   # If >, result = tval
  ret
```





## 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**





## Today

- Complete addressing mode, address computation (leal)
- Arithmetic operations
- x86-64
- Control: Condition codes
- Conditional branches and moves
- **Loops**

## "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 to either 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:
  - %edx x
  - %ecx result

```
  movl   $0, %ecx     #   result = 0
.L2:                  # loop:
  movl   %edx, %eax
  andl   $1, %eax     #   t = x & 1
  addl   %eax, %ecx   #   result += t
  shrl   %edx         #   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?

## 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;
}
```





## Summary

- **Today**
  - Complete addressing mode, address computation (leal)
  - Arithmetic operations
  - Control: Condition codes
  - Conditional branches & conditional moves
  - Loops
- **Next Time**
  - Switch statements
  - Stack
  - Call / return
  - Procedure call discipline