Controlling Program Flow

- **Conditionals** (If-statement)
- **Loops**
  (while, do-while, for-loops)
- **Switch Statements**
- **New Instructions**
  JMP
  CMP
  Conditional jumps (branches)
  Conditional MOV instruction

Conditional statements

Normally: Sequential Execution of instructions

Changing the Flow of Control

- JUMP and CALL instructions

Some jumps and calls are *conditional*

- Flow of control constructs in “C”:

```
if (x) {…} else {…}
while (x) {…}
do {…} while (x)
for (i=0; i<max; i++) {…}
switch (x) {
  case 1: …
  case 2: …
}
```

Condition Code Register

A register in the processor

eflags *(extended flags)*

Each bit is a flag, or *condition code*

`CF` Carry Flag

`SF` Sign Flag

`ZF` Zero Flag

`OF` Overflow Flag

Not like general purpose register

Do not read/write directly

Flags are modified by hardware

Depending on the result of an instruction

"set" (=1) *"cleared" (=0)*

Condition Codes

Automatically Set/Cleared by Arithmetic and Logical Operations

Example: `addl Src,Dest`

C analog: `t = a + b`

CF (carry flag)

- set if unsigned overflow (carry out from MSB)

  `(unsigned t) < (unsigned a)`

ZF (zero flag)

- set if `t == 0`

SF (sign flag)

- set if `t < 0`

OF (overflow flag)

- set if signed (two's complement) overflow

  `(a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)`

Set/Cleared by compare and test operations as well.

Not modified by `lea`, `push`, `pop`, `mov` instructions.

Condition Codes

The compare instruction sets the condition codes

`cmpq` *b,a*

Compute `a-b` without altering the 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`
- **OF** set if two's complement overflow
  - `(a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)`

<u>Variations</u>

`cmpb` *b,a*

`cmpw` *b,a*

`cmpl` *b,a*

`cmpq` *b,a*

Condition Codes

The test instruction also sets the condition codes

`testq  b,a`

- Computes a&b without setting destination
  - Sets condition codes based on result
  - Useful to have one of the operands be a mask
- Often used to test zero, positive

  `testq  %rax,%rax`

  - ZF set when `a&b == 0`
  - SF set when `a&b < 0`
- Variations

```
testb  b,a
testw  b,a
testl  b,a
testq  b,a
```

Jump Instructions

Change sequential flow of execution

- One operand: the jump target
- Variations: conditional or unconditional

Unconditional Jumps

- Direct jump

```
jmp MyLoopLabel
jmp .L1
jmp 0x0040C0
```

- Indirect Jump

```
jmp *Operand
jmp *%rax
```

  - Jump target is specified by a register or memory location

Conditional Jumps

- Based on the condition codes!

Jump Instructions

Jump depending on condition codes

jX	Condition	Description
`jmp`	1	Unconditional
`je, jz`	ZF	Equal / Zero
`jne,jnz`	~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)

Overflow flips result

Jump Instructions

What’s the difference between jg and ja ?

Which one would you use to compare two pointers?

Conditional Branch Example

```
gcc –Og -S –fno-if-conversion control.c
```

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

Register	Use(s)
`%rdi`	Argument **`x`**
`%rsi`	Argument **`y`**
`%rax`	Return value

```
absdiff:
   cmpq    %rsi, %rdi   # x:y
   jle     .L4
   movq    %rdi, %rax
   subq    %rsi, %rax
   ret
.L4:            # x <= y
   movq    %rsi, %rax
   subq    %rdi, %rax
   ret
```

Expressing with Goto Code

```
if (condition) {
    Then Statements
    Then Statements
    Then Statements
} else {
    Else Statements
    Else Statements
    Else Statements
}
```

```
if (not condition) goto Else;
    Then Statements
    Then Statements
    Then Statements
    goto Done;
Else:
    Else Statements
    Else Statements
    Else Statements
Done:
```

Expressing with Goto Code

C allows `goto` statement

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

```
long absdiff_j
  (long x, long y)
{
    long result;
    int ntest = x <= y;
    if (ntest) goto Else;
    result = x-y;
    goto Done;
 Else:
    result = y-x;
 Done:
    return result;
}
```

The SetXX Instructions

- Set low-order byte of destination to 0x00 or 0x01 based on combinations of condition codes
- Does not alter remaining 7 bytes.

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)

The SetXX Instructions

- Set low-order byte of destination to 0x00 or 0x01 based on combinations of condition codes
- Does not alter remaining 7 bytes.

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

Register	Use(s)
`%rdi`	Argument `x`
`%rsi`	Argument `y`
`%rax`	Return value

- Typically use `movzbl` to finish job
  - (The 32-bit instructions also zero out the upper 32-bits.)

```
gt:
   cmpq   %rsi, %rdi    # Compare x:y
   setg   %al           # Set when >
   movzbl %al, %eax     # Zero rest of %rax
   ret
```

Conditional Expressions

An expression operator in "C"

(Test ? Then_Expr : Else_Expr)

Example

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

Translation, using goto code:

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

Create separate code regions for then & else expressions

Execute the appropriate one

The Conditional Move Instructions

Problem:

- Branches are very disruptive to instruction flow through pipelines!

A group of instructions:

- What they do:
  - if (Test) Dest ← Src
- Example:

```
cmovge   %rax,%rbx
```

Benefits:

- Conditional moves do not require control transfer!
- GCC tries to use them
  - (Not always possible)

C Code

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

Goto Version

```
result = Then_Expr;
temp = Else_Expr;
nt = !Test;
if (nt) result = temp;
```

The Conditional Move Instructions

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

Register	Use(s)
`%rdi`	Argument **x**
`%rsi`	Argument **y**
`%rax`	Return value

```
absdiff:
   movq    %rdi, %rax  # x
   subq    %rsi, %rax  # result = x-y
   movq    %rsi, %rdx
   subq    %rdi, %rdx  # eval = y-x
   cmpq    %rsi, %rdi  # x:y
   cmovle  %rdx, %rax  # if <=, result = eval
   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

Loops

Implemented in assembly via tests and jumps

- Compilers implement most loops as `do-while`
  - Add additional check at beginning to get "`while-do`"
- Convenient to write using "`goto`" in order to understand assembly implementation

do-while

```
do {
      body-statements
  } while (test-expr);
```

goto version

```
loop:
    body-statements
    t = test-expr
    if (t) goto loop
```

while-do

```
while (test-expr) {
      body-statements
  }
```

goto version

```
    t = test-expr
    if (not t) goto exit
loop:
    body-statements
    t = test-expr
    if (t) goto loop
exit:
```

Loops

Implemented in assembly via tests and jumps

Compilers implement most loops as `do-while`

- Add additional check at beginning to get "`while-do`"

Convenient to write using "`goto`" in order to understand assembly implementation

do-while

```
do {
      body-statements
  } while (test-expr);
```

goto version

```
loop:
    body-statements
    t = test-expr
    if (t) goto loop
```

while-do

```
while (test-expr) {
      body-statements
  }
```

goto version

```
    goto test
loop:
    body-statements
test:
    t = test-expr
    if (t) goto loop
```

C examples

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

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

```
factorial_goto:
        movl    $1, %eax        ; eax = result = 1
.L2:
        imull   %edi, %eax      ; result = result*x
        subl    $1,%edi         ; x--
        cmpl    $1,%edi         ; if x > 1
        jg      .L2             ;    goto .L2
        rep ret                 ; return
```

"do-while" example revisited

C code: *do-while* *while-do*

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

```
int factorial_while(int x)
{
  int result = 1;
  while (x > 1) {
     result *= x;
     x = x-1;
  }
  return result;
}
```

Are these equivalent?

"do-while" example revisited

Assembly: *do-while* *while-do*

```
factorial_do:
        movl    $1,%eax


.L2:
        imull   %edi, %eax
        subl    $1,%edi
        cmpl    $1,%edi
        jg      .L2

        rep ret
```

```
factorial_while:
        movl    $1,%eax
        cmpl    $1,%edi
        jle     .L6
.L2:
        imull   %edi, %eax
        subl    $1,%edi
        cmpl    $1,%edi
        jg      .L2
.L6:
        rep ret
```

"For" Loop Example

```
int factorial_for(int x)
{
  int result;
  for (result=1; x > 1; x=x-1) {
      result *= x;
  }
  return result;
}
```

Init

```
result = 1
```

Test

```
x > 1
```

Update

```
x = x - 1
```

Body

```
{
  result *= x;
}
```

Is this code equivalent to the do-while version or the while-do version?

"For" Loop Example

```
int factorial_for(int x)
{
  int result;
  for (result=1; x > 1; x=x-1) {
      result *= x;
  }
  return result;
}
```

General Form

```
for (Init; Test; Update )

      Body
```

```
    Init;
    if (not Test) goto exit;
loop:
    Body;
    Update;
    if (Test) goto loop;
exit:
```

Init	*Test*	*Update*
`result = 1`	`x > 1`	`x = x - 1`

Body

```
{
  result *= x;
}
```

Is this code equivalent to the do-while version or the while-do version?

“For” Loop Example

```
factorial_for:
        movl    $1,%eax
        cmpl    $1,%edi
        jle     .L6
.L2:
        imull   %edi, %eax
        subl    $1,%edi
        cmpl    $1,%edi
        jg      .L2
.L6:
        rep ret
```

```
   Init;
   if (not Test) goto exit;
loop:
   Body;
   Update;
   if (Test) goto loop;
exit:
```

"For" Loop Example

```
factorial_for:
        movl    $1,%eax
        cmpl    $1,%edi
        jle     .L6
.L2:
        imull   %edi, %eax
        subl    $1,%edi
        cmpl    $1,%edi
        jg      .L2
.L6:
        rep ret
```

```
factorial_while:
        movl    $1,%eax
        cmpl    $1,%edi
        jle     .L6
.L2:
        imull   %edi, %eax
        subl    $1,%edi
        cmpl    $1,%edi
        jg      .L2
.L6:
        rep ret
```

Reverse Engineer This!

```
loop:
    subl    $1, %edx
    js      .L18
    imull   %edi, %esi
    movl    $0, %eax
.L17:
    addl    %esi, %eax
    subl    %edi, %edx
    jns     .L17
    rep ret
.L18:
    movl    $0, %eax
    ret
```

%edi %esi %edx

```
int loop(int x, int y, int z)
{
  int result=0;
  int i;
  for (i = ____ ; i ____ ; i = ____ )
  {
     result += ___ ;
  }
  return result;
}
```

What registers hold result and `i`?
What is the initial value of `i`?
What is the test condition on `i`?
How is `i` updated?
What instructions increment result?

Reverse Engineer This!

```
loop:
    subl    $1, %edx
    js      .L18
    imull   %edi, %esi
    movl    $0, %eax
.L17:
    addl    %esi, %eax
    subl    %edi, %edx
    jns     .L17
    rep ret
.L18:
    movl    $0, %eax
    ret
```

%edi %esi %edx

```
int loop(int x, int y, int z)
{
  int result=0;
  int i;
  for (i = z-1  ; i >= 0 ; i = i-x  )
  {
     result += y*x ;
  }
  return result;
}
```

What registers hold result and `i`? %eax = result, %edx = i
What is the initial value of `i`? i = z-1
What is the test condition on `i`? i >= 0
How is `i` updated? i = i - x
What instructions increment result? addl (x*y)

C Switch Statements

Test whether an expression matches one of a number of constant integer values.

Branch accordingly.

Missing “break”?

Fall through to the next case.

No matching case?

Execute “default” case.

```
int switch_eg (int x) {
    int result = x;
    switch (x) {
        case 100:
            result *= 13;
            break;

        case 102:
            result += 10;
            /* Fall through */

        case 103:
            result += 11;
            break;

        case 104:
        case 106:
            result *= result;
            break;

        default:
            result = 0;
    }
    return result;
}
```

C Switch Statements

Implementation options

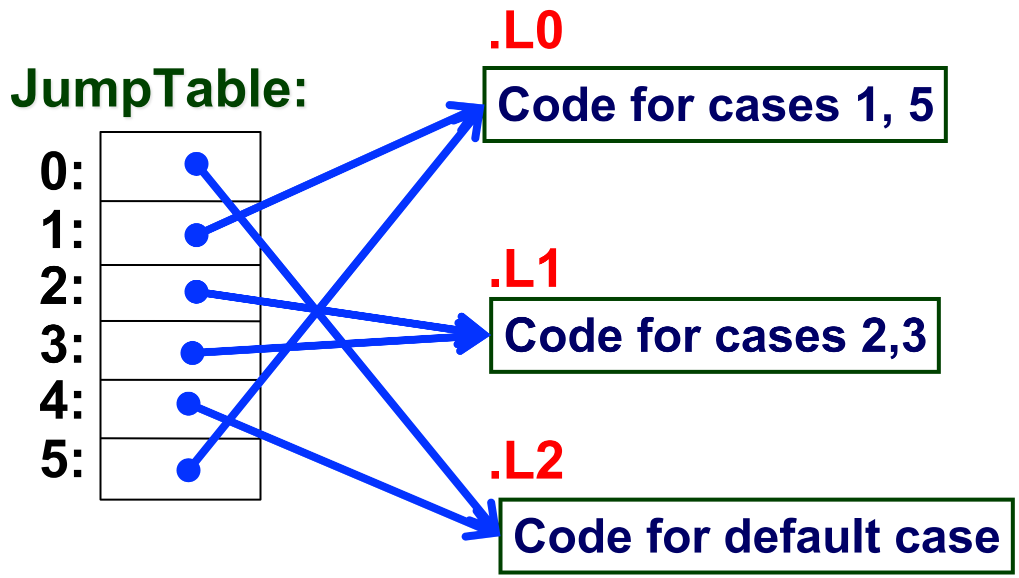
- **Series of conditionals**
  - `testl/cmpl` followed by `je`
  - Good, if only a few cases
  - Slow, if many cases
- **Jump table (example below)**
  - Build a table of addresses
  - Use the index value as an offset into this table
  - Each table entry points to the right chunk of code
  - Do an “indirect jump” through the table
  - Possible with a small range of integer constants

GCC picks implementation based on the actual switch values.

C Switch Statements

Example:

```
switch (x) {
  case 1:
  case 5:
       code at L0
  case 2:
  case 3:
       code at L1
  default:
       code at L2
}
```




```
     Check that 0 ≤ x ≤ 5
     if not, goto .L2
     %rax = .L3 + (4 *x)
     jmp      * %rax
.L3:
     .quad    .L2
     .quad    .L0
     .quad    .L1
     .quad    .L1
     .quad    .L2
     .quad    .L0
```

```c
int switch_eg (int x) {
    int result = x;
    switch (x) {
      case 100:
        result *= 13;
        break;

      case 102:
        result += 10;
        /* Fall through */

      case 103:
        result += 11;
        break;

      case 104:
      case 106:
        result *= result;
        break;

      default:
        result = 0;
    }
    return result;
}
```

```
switch_eg:
    leal    -100(%rdi), %eax
    cmpl    $6, %eax
    ja      .L23
    movl    %eax, %eax
    jmp     *.L19(,%rax,8)
.L18:
    leal    (%rdi,%rdi,2), %eax
    leal    (%rdi,%rax,4), %eax
    ret
.L20:
    addl    $10, %edi
.L21:
    leal    11(%rdi), %eax
    ret
.L22:
    movl    %edi, %eax
    imull   %edi, %eax
    ret
.L23:
    movl    $0, %eax
    ret
```

100	.L18
101	.L23
102	.L20
103	.L21
104	.L22
105	.L23
106	.L22

```c
int switch_eg (int x) {
    int result = x;
    switch (x) {
        case 100:
            result *= 13;
            break;

        case 102:
            result += 10;
            /* Fall through */

        case 103:
            result += 11;
            break;

        case 104:
        case 106:
            result *= result;
            break;

        default:
            result = 0;
    }
    return result;
}
```

```
switch_eg:
    leal    -100(%rdi), %eax
    cmpl    $6, %eax
    ja      .L23
    movl    %eax, %eax
    jmp      *.L19(,%rax,8)
.L18:
    leal    (%rdi,%rdi,2), %eax
    leal    (%rdi,%rax,4), %eax
    ret
.L20:
    addl    $10, %edi
.L21:
    leal    11(%rdi), %eax
    ret
.L22:
    movl    %edi, %eax
    imull   %edi, %eax
    ret
.L23:
    movl    $0, %eax
    ret
```

```
    .section   .rodata
    .align 8
.L19:
    .quad   .L18      100
    .quad   .L23      101
    .quad   .L20      102
    .quad   .L21      103
    .quad   .L22      104
    .quad   .L23      105
    .quad   .L22      106
```

Reverse Engineering Challenge

```
int switch2(int x) {
     int result = 0;
     switch (x) {
        ...???...
     }
     return result;
}
```

```
        addl    $2, $edi
        cmpl    $6, %edi
        ja      .L25
        movl    %edi,%edi
        jmp     *.L27(,%rdi,8)
        .align  8
.L27:
        .quad   .L26
        .quad   .L25
        .quad   .L32
        .quad   .L29
        .quad   .L30
        .quad   .L30
        .quad   .L31
```

The body of the switch statement has been omitted in the above C program. The code has case labels that did not span a contiguous range, and some cases had multiple labels. GCC generates the code shown when compiled. Variable x is initially at offset 8 relative to register %ebp.

a) What were the values of the case labels in the switch statement body?
b) What cases had multiple labels in the C code?

Reverse Engineering Challenge

```
int switch2(int x) {
     int result = 0;
     switch (x) {
        ...???...
     }
     return result;
}
```

Sets start range to -2

Top range is 4

```
        addl    $2, %edi
        cmpl    $6, %edi
        ja      .L25
        movl    %edi,%edi
        jmp     *.L27(,%rdi,8)
        .align  8
.L27:
        .quad   .L26    -2
        .quad   .L25    -1
        .quad   .L32    0
        .quad   .L29    1
        .quad   .L30    2
        .quad   .L30    3
        .quad   .L31    4
```

case –2:
/* Code at .L26 */
case 0:
/* Code at .L32 */
case 1:
/* Code at .L29 */
case 2,3:
/* Code at .L30 */
case 4:
/* Code at .L31 */
case –1:
default:
/* Code at .L25 */

"For" Loop Example: `ipwr`

$$\boxed{3^{11} = 3 \times \underbrace{3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3}_{\textbf{10 multiplications}}}$$

$$\boxed{3^{47} = 3 \times \underbrace{3 \times 3 \times 3 \times 3 \times \ldots \times\ 3 \times 3 \times 3 \times 3 \times 3 \times 3}_{\textbf{n-1 multiplications}}}$$

Is there a better algorithm?

"For" Loop Example: `ipwr`

$$3^{11} = \underbrace{3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3}_{\textbf{10 multiplications}}$$

$$3^{47} = \underbrace{3 \times 3 \times 3 \times 3 \times 3 \times \ldots \times 3 \times 3 \times 3 \times 3 \times 3 \times 3}_{\textbf{n-1 multiplications}}$$

Is there a better algorithm?

$$3^{11} = 3^{1+2+8} = 3^{1+2+0+8+0+\ldots} = 3^{1} \times 3^{2} \qquad \times 3^{8}$$

"For" Loop Example: `ipwr`

$$3^{11} = \underbrace{3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3}_{\textbf{10 multiplications}}$$

$$3^{47} = \underbrace{3 \times 3 \times 3 \times 3 \times 3 \times \ldots \times 3 \times 3 \times 3 \times 3 \times 3 \times 3}_{\textbf{n-1 multiplications}}$$

Is there a better algorithm?

$$\begin{aligned} 3^{11} = 3^{1+2+8} = 3^{1+2+0+8+0+\ldots} &= 3^1 \times 3^2 \times 3^4 \times 3^8 \times 3^{16} \times \ldots \\ &= 3 \times 3^2 \times (3^2)^2 \times ((3^2)^2)^2 \times (((3^2)^2)^2)^2 \times \ldots \end{aligned}$$

"For" Loop Example: `ipwr`

$$3^{11} = 3 \times \underbrace{3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3}_{\textbf{10 multiplications}}$$

Algorithm

Exploit property that $p = p_0 + 2p_1 + 4p_2 + \ldots\ 2^{n-1}p_{n-1}$

Gives: $x^p = z_0 \cdot z_1^{\,2} \cdot (z_2^{\,2})^2 \cdot\ \ldots\ \cdot \underbrace{(\ldots((z_{n-1}^{\,2})^2)\ldots)^2}_{n\text{–}1 \text{ times}}$

$z_i = 1$ when $p_i = 0$

$z_i = x$ when $p_i = 1$

Complexity $\mathrm{O}(\log p)$

$$\begin{aligned} 3^{11} = 3^{1+2+8} = 3^{1+2+0+8+0+\ldots} &= 3^1 \times 3^2 \times 3^4 \times 3^8 \times 3^{16} \times \ldots \\ &= 3 \times 3^2 \times (3^2)^2 \times ((3^2)^2)^2 \times (((3^2)^2)^2)^2 \times \ldots \end{aligned}$$

"For" Loop Example: `ipwr`

```
/* Compute x raised to nonnegative power p */
int ipwr_for(int x, unsigned p) {
  int result;
  for (result = 1; p != 0; p = p>>1) {
    if (p & 0x1)
      result *= x;
    x = x*x;
  }
  return result;
}
```

Algorithm

Exploit property that $p = p_0 + 2p_1 + 4p_2 + \ldots 2^{n-1}p_{n-1}$

Gives: $x^p = z_0 \cdot z_1^{\;2} \cdot (z_2^{\;2})^2 \cdot \ \ldots \ \cdot \underbrace{(\ldots((z_{n-1}^{\;2})^2)\ldots)^2}_{n\text{–}1 \text{ times}}$

- $z_i = 1$ when $p_i = 0$
- $z_i = x$ when $p_i = 1$

Complexity $O(\log p)$

$$3^{11} = 3^{1+2+8} = 3^{1+2+0+8+0+\ldots} = 3^1 \times 3^2 \times 3^4 \times 3^8 \times 3^{16} \times \ldots$$
$$= 3 \times 3^2 \times (3^2)^2 \times ((3^2)^2)^2 \times (((3^2)^2)^2)^2 \times \ldots$$

"For" Loop Example: `ipwr`

```
/* Compute x raised to nonnegative power p */
int ipwr_for(int x, unsigned p) {
  int result;
  for (result = 1; p != 0; p = p>>1) {
    if (p & 0x1)
      result *= x;
    x = x*x;
  }
  return result;
}
```

result	x	p	p
1	3	11	1011
3	9	5	101
27	81	2	10
27	6561	1	1
177,147	43,046,721	0	0

$$3^{11} = 3^{1+2+8} = 3^{1+2+0+8+0+\ldots} = 3^1 \times 3^2 \times 3^4 \times 3^8 \times 3^{16} \times \ldots$$

$$= 3 \times 3^2 \times (3^2)^2 \times ((3^2)^2)^2 \times (((3^2)^2)^2)^2 \times \ldots$$

"For" Loop Example: ipwr

```
/* Compute x raised to nonnegative power p */
int ipwr_for(int x, unsigned p) {
  int result;
  for (result = 1; p != 0; p = p>>1) {
    if (p & 0x1)
      result *= x;
    x = x*x;
  }
  return result;
}
```

```
ipwr_for:                                (edi←x, esi←P)
    movl    $1, %eax                     result = 1
    jmp     TEST                         goto TEST
LOOP:                                  LOOP:
    testb   $1, %sil                     if (p & 0x01) goto ELSE
    je      ELSE                         .
    imull   %edi, %eax                     result *= x
ELSE:                                  ELSE:
    imull   %edi, %edi                   x = x*x
    shrl    %esi                         p = p>>1
TEST:                                  TEST:
    testl   %esi, %esi                   if p≠0 goto LOOP
    jne     LOOP                         .
    rep     ret                          return
```

result			
1			
3			
27			
27			
177,147	43,046,721	0	0

$$3^{11} = 3^{1+2+8} = 3^{1+2+0+8+0+\ldots} = 3^1 \times 3^2 \times 3^4 \times 3^8 \times 3^{16} \times \ldots$$

$$= 3 \times 3^2 \times (3^2)^2 \times ((3^2)^2)^2 \times (((3^2)^2)^2)^2 \times \ldots$$

Summary

C Control

- if-then-else
- do-while
- while
- switch

Assembler Control

- Jump
- Conditional Jump

Compiler

- Must generate assembly code to implement more complex control

Standard Techniques

All loops converted to do-while form

Large switch statements use jump tables

Conditions in CISC

CISC machines generally have condition code registers

Conditions in RISC

Use general registers to store condition information

Special comparison instructions

E.g., on Alpha:

```
cmple $16,1,$1
```

Sets register $1 to 1 when Register $16 <= 1