Controlling Program Flow

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

Conditional statements

Computers execute instructions in sequence.

Except when we change the flow of control

- **Jump and Call instructions**

Some jumps and calls are *conditional*

- **A computer needs to jump if certain a condition is true**
- **In C: `if`, `while`, `do-while`, `for`, and `switch` statements**
  - `if (x) {…} else {…}`
  - `while (x) {…}`
  - `do {…} while (x)`
  - `for (i=0; i<max; i++) {…}`
  - `switch (x) {`
    `case 1: …`
    `case 2: …`
    `}`

Condition codes

The IA32 processor has a register called *eflags*

(extended flags)

Each bit is a flag, or *condition code*

- `CF` Carry Flag
- `SF` Sign Flag
- `ZF` Zero Flag
- `OF` Overflow Flag

As programmers, we don't write to this register and seldom read it directly

Flags are set or cleared 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 (cont.)

Setting condition codes via compare instruction

```
cmpl  b,a
```

Computes `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`
- OF set if two's complement overflow

```
(a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)
```

Byte, word, long versions

```
cmpb  b,a
cmpw  b,a
cmpl  b,a
```

Condition Codes (cont.)

Setting condition codes via test instruction

```
testl  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

```
testl  %eax,%eax
```

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

Byte, word, long versions

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

Jump Instructions

Change sequential flow of execution

- Has a parameter as the jump target
- Can be unconditional or conditional jump

Example:

- Unconditional jump
  - Direct jump: `jmp Label`

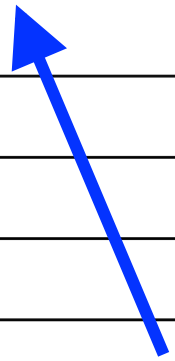
    Jump target is specified by a label (e.g., `jmp .L1`)
  - Indirect jump: `jmp *Operand`

    Jump target is specified by a register or memory location (e.g., `jmp *%eax`)
- Conditional jump: based on one or more 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)



Jump Instructions

What's the difference between jg and ja ?

Which one would you use to compare two pointers?

Conditional Branch Example

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

```
  %edx = x
  %eax = y
  if (x ≤ y) goto L9
        cmp   y,x
        jle   L9
  %eax = %edx
L9:
  Result is in %eax
```

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

    movl  8(%ebp),%edx    ; %edx = x
    movl  12(%ebp),%eax   ; %eax = y
    cmpl  %eax,%edx       ; x-y
    jle   L9              ; jmp to "else" (x <= y)
    movl  %edx,%eax       ; %eax = x (x > y)
L9:
    movl  %ebp,%esp
    popl  %ebp
    ret
```

Set Up

Body

Finish

Reading and saving condition codes

`set`*XX* Instructions

- Set single byte based on combinations of condition codes
  - Store `0x00` or `0x01` in low byte of register
  - Does not alter remaining 3 bytes
  - Or, store the byte in memory


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

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

set*XX* instructions

setX	Synonym	Effect	Set condition
`sete D`	`setz`	`D <- ZF`	Equal/zero
`setne D`	`setnz`	`D <- ~ZF`	Not equal / Not zero
`sets D`		`D <- SF`	Negative
`setns D`		`D <- ~SF`	Non-negative
`setg D`	`setnle`	`D <- ~(SF^OF) & ~ZF`	Greater (signed >)
`setge D`	`setnl`	`D <- ~(SF^OF)`	Greater or equal (signed >=)
`setl D`	`setnge`	`D <- SF^OF`	Less (signed <)
`setle D`	`setng`	`D <- (SF^OF)|ZF`	Less or equal (signed <=)
`seta`	`setnbe`	`D <- ~CF & ~ZF`	Above (unsigned >)
`setae`	`setnb`	`D <- ~CF`	Above or equal (unsigned >=)
`setb`	`setnae`	`D <- CF`	Below (unsigned <)
`setbe`	`setna`	`D <- CF | ZF`	Below or equal (unsigned <=)

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
  }
```

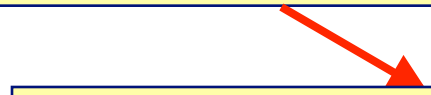
goto version

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

C examples

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

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



```
factorial_goto:
        pushl   %ebp
        movl    %esp, %ebp
        movl    8(%ebp), %edx   ; edx = x
        movl    $1, %eax        ; eax = result = 1
.L2:
        imull   %edx, %eax      ; result = result*x
        decl    %edx            ; x--
        cmpl    $1, %edx        ; if x > 1
        jg      .L2             ;    goto .L2
        popl    %ebp            ; return
        ret
```

“do-while” example revisited

C code: *do-while*

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

while-do

```
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***

```
factorial_do:
        pushl   %ebp
        movl    %esp, %ebp
        movl    8(%ebp), %edx
        movl    $1, %eax


.L2:
        imull   %edx, %eax
        decl    %edx
        cmpl    $1, %edx
        jg      .L2

        popl    %ebp
        ret
```

while-do

```
factorial_while:
        pushl   %ebp
        movl    %esp, %ebp
        movl    8(%ebp), %edx
        movl    $1, %eax
        cmpl    $1, %edx
        jle     .L6
.L2:
        imull   %edx, %eax
        decl    %edx
        cmpl    $1, %edx
        jg      .L2
.L6:
        popl    %ebp
        ret
```

“For” Loop Example

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

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

```
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:
        pushl   %ebp
        movl    %esp, %ebp
        movl    8(%ebp), %edx
        movl    $1, %eax
        cmpl    $1, %edx
        jle     .L7
.L5:
        imull   %edx, %eax
        decl    %edx
        cmpl    $1, %edx
        jg      .L5
.L7:
        popl    %ebp
        ret
```

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

“For” Loop Example

```
factorial_for:
        pushl   %ebp
        movl    %esp, %ebp
        movl    8(%ebp), %edx
        movl    $1, %eax
        cmpl    $1, %edx
        jle     .L7
.L5:
        imull   %edx, %eax
        decl    %edx
        cmpl    $1, %edx
        jg      .L5
.L7:
        popl    %ebp
        ret
```

```
factorial_while:
        pushl   %ebp
        movl    %esp, %ebp
        movl    8(%ebp), %edx
        movl    $1, %eax
        cmpl    $1, %edx
        jle     .L6
.L4:
        imull   %edx, %eax
        decl    %edx
        cmpl    $1, %edx
        jg      .L4
.L6:
        popl    %ebp
        ret
```

Reverse Engineer This!

```
        movl    8(%ebp),%ebx
        movl    16(%ebp),%edx
        xorl    %eax,%eax
        decl    %edx
        js      .L4
        movl    %ebx,%ecx
        imull   12(%ebp),%ecx
.L6:
        addl    %ecx,%eax
        subl    %ebx,%edx
        jns     .L6
.L4:
        popl    %ebp
        ret
```

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

Strategy:

- **Bind registers that are modified (%eax, %edx) to local variables (result, i) are modified**
- **Registers that are mostly constant (%ecx) can be bound to input parameters (x,y,z)**

Reverse Engineer This!

```
        movl    8(%ebp),%ebx
        movl    16(%ebp),%edx
        xorl    %eax,%eax
        decl    %edx
        js      .L4
        movl    %ebx,%ecx
        imull   12(%ebp),%ecx
.L6:
        addl    %ecx,%eax
        subl    %ebx,%edx
        jns     .L6
.L4:
        popl    %ebp
        ret
```

```
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!

```
        movl    8(%ebp),%ebx
        movl    16(%ebp),%edx
        xorl    %eax,%eax
        decl    %edx
        js      .L4
        movl    %ebx,%ecx
        imull   12(%ebp),%ecx
.L6:
        addl    %ecx,%eax
        subl    %ebx,%edx
        jns     .L6
.L4:
        popl    %ebp
        ret
```

```
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 and branches accordingly
- Without a “break” the code falls through to the next case
- If x matches no case, then “default” is executed

```
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

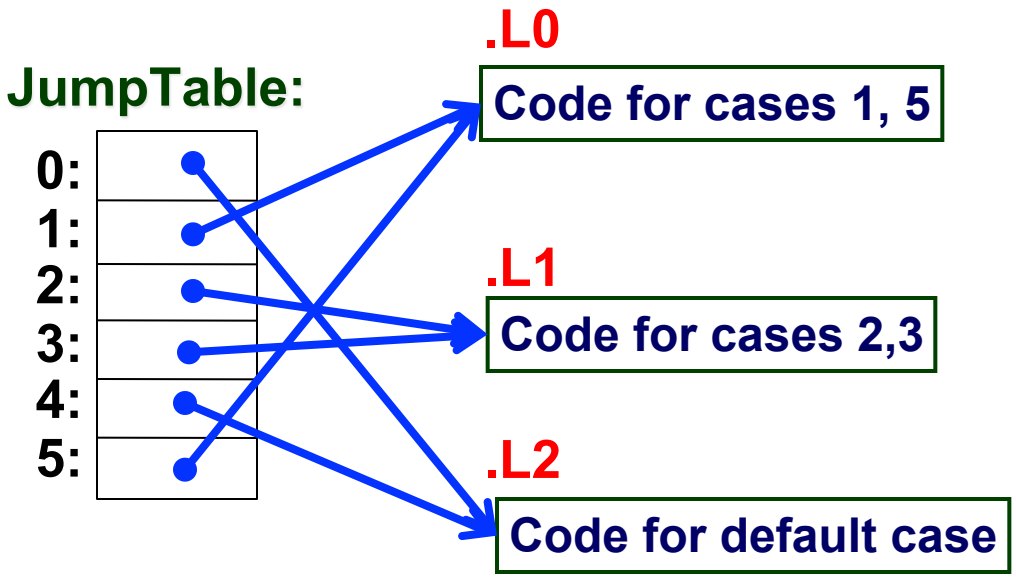
- Series of conditionals
  - `testl/cmpl` followed by `je`
  - Good, if only a few cases
  - Slow, if many cases
- Jump table (example below)
  - Build a table of addresses
  - Use the index value as an offset into this table
  - Each table entry points to the right chunk of code
  - Do an “indirect 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
        %eax = .L3 + (4 *x)
        jmp      * %eax
.L3:
        .long    .L2
        .long    .L0
        .long    .L1
        .long    .L1
        .long    .L2
        .long    .L0
```

Example revisited

```
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;
}
```

```
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;
}
```

```
        leal    -100(%edx),%eax
        cmpl    $6,%eax
        ja      .L9
        jmp     *.L10(,%eax,4)
        .p2align 4,,7
.section        .rodata
        .align  4
        .align  4
.L10:
        .long   .L4     100
        .long   .L9     101
        .long   .L5     102
        .long   .L6     103
        .long   .L8     104
        .long   .L9     105
        .long   .L8     106
.text
        .p2align 4,,7
.L4:                    100
        leal    (%edx,%edx,2),%eax
        leal    (%edx,%eax,4),%edx
        jmp     .L3
        .p2align 4,,7
.L5:                    102
        addl    $10,%edx
```

```
.L6:                    103
        addl    $11,%edx
        jmp     .L3
        .p2align 4,,7
.L8:                    104, 106
        imull   %edx,%edx
        jmp     .L3
        .p2align 4,,7
.L9:                    default
        xorl    %edx,%edx
.L3:
        movl    %edx,%eax
```

Key is *jump table at L10*

Array of pointers to jump locations

Reverse Engineering Challenge

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

```
          movl     8(%ebp), %eax
          addl     $2, $eax
          cmpl     $6, %eax
          ja       .L10
          jmp      *.L11(,%eax,4)
          .align   4
.L11:
          .long    .L4
          .long    .L10
          .long    .L5
          .long    .L6
          .long    .L8
          .long    .L8
          .long    .L9
```

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

```
          movl    8(%ebp), %eax
          addl    $2, $eax
          cmpl    $6, %eax
          ja      .L10
          jmp     *.L11(,%eax,4)
          .align  4
.L11:
          .long   .L4       -2
          .long   .L10      -1
          .long   .L5       0
          .long   .L6       1
          .long   .L8       2
          .long   .L8       3
          .long   .L9       4
```

```
case –2:
     /* Code at .L4 */
case 0:
     /* Code at .L5 */
case 1:
     /* Code at .L6 */
case 2,3:
     /* Code at .L8 */
case 4:
     /* Code at .L9 */
case –1:
default:
     /* Code at .L10 */
```

Avoiding conditional branches

Modern CPUs with deep pipelines

- Instructions fetched far in advance of execution
- Mask the latency of going to memory
- Problem: What if you hit a conditional branch?
  - Must predict which branch to take!
  - Branch prediction in CPUs well-studied, fairly effective
  - But, best to avoid conditional branching altogether

Conditional Move Instructions

Conditional instruction execution

```
cmovXX  src,dest
```

- Move value from src to dest if condition *XX* holds
- No branching
- Handled as operation within Execution Unit

Older versions of compiler won't use this instruction

- GCC may think it's compiling for a 386
- Added with P6 microarchitecture (PentiumPro onward)

Performance

- 14 cycles on all data
- More efficient than conditional branching (simple control flow)
- But overhead: both branches must be executed.

Conditional Move Example

C Code:

```
rval = (y<x) ? x : y;
```

Assembly Code:

```
movl    8(%ebp),%edx      ; Get x
movl    12(%ebp),%eax     ; rval=y
cmpl    %edx, %eax        ; compare y:x
cmovll  %edx,%eax         ; If <, rval=x
```

Performance

- Same number of cycles on all data
- More efficient than with conditional branching
- But overhead: both branches must be executed.

Conditional Move: General Form

C Code

```
val = Test ? Then-Expr : Else-Expr;
```

Conditional Move Version

```
val    = Then-Expr;
temp   = Else-Expr;
val    = temp if !Test;
```

Both values get computed

Overwrite then-value with else-value if condition doesn't hold

Cannot use when:

- **Then or else expression have side effects**
- **Then and else expression are more expensive than branch misprediction**

Conditional Move Example

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

```
absdiff:                        ; x in %edi, y in %esi
       movl   %edi,%eax         ; eax = x
       movl   %esi,%edx         ; edx = y
       subl   %esi,%eax         ; eax = x-y
       subl   %edi,%edx         ; edx = y-x
       cmpl   %esi,%edi         ; x:y
       cmovle %edx,%eax         ; eax=edx if <=
       ret
```

"For" Loop Example: `ipwr`

$$3^{11} = 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3$$

10 multiplications

$$3^{47} = 3 \times 3 \times 3 \times 3 \times 3 \times \ldots \times 3 \times 3 \times 3 \times 3 \times 3 \times 3$$

n-1 multiplications

Is there a better algorithm?

"For" Loop Example: `ipwr`

$$3^{11} = 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3$$

10 multiplications

$$3^{47} = 3 \times 3 \times 3 \times 3 \times 3 \times \ldots \times 3 \times 3 \times 3 \times 3 \times 3 \times 3$$

n-1 multiplications

Is there a better algorithm?

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

"For" Loop Example: `ipwr`

$$3^{11} = 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3$$

10 multiplications

$$3^{47} = 3 \times 3 \times 3 \times 3 \times 3 \times \ldots \times 3 \times 3 \times 3 \times 3 \times 3 \times 3$$

n-1 multiplications

Is there a better algorithm?

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

"For" Loop Example: `ipwr`

$$3^{11} = 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3$$

10 multiplications

Algorithm

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

Gives: $x^p = z_0 \cdot z_1{}^2 \cdot (z_2{}^2)^2 \cdot \ldots \cdot (\ldots((z_{n-1}{}^2)^2)\ldots)^2$

$z_i = 1$ when $p_i = 0$

$z_i = x$ when $p_i = 1$

n–1 **times**

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;
}
```

Algorithm

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

Gives: $x^p = z_0 \cdot z_1{}^2 \cdot (z_2{}^2)^2 \cdot \ldots \cdot (\ldots((z_{n-1}{}^2)^2)\ldots)^2$

$z_i = 1$ when $p_i = 0$

$z_i = x$ when $p_i = 1$

n–1 **times**

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:
        pushl   %ebp
        movl    %esp, %ebp
        movl    12(%ebp), %ecx    ecx = p
        movl    $1, %eax          result=1
        testl   %ecx, %ecx        if p==0
        je      Exit                goto ENDLOOP
        movl    8(%ebp), %edx     edx = x
LOOP:                           LOOP:
        testb   $1, %cl           if !(p&0x01)
        je      Test                goto Endif
        imull   %edx, %eax         res = x * res
ENDIF:                          ENDIF:
        imull   %edx, %edx        x = x * x
        shrl    %ecx              p = p>>1
        jne     LOOP              goto Loop
ENDLOOP:                        ENDLOOP
        popl    %ebp
        retl
```

result
1
3
27
27
177,147

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

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