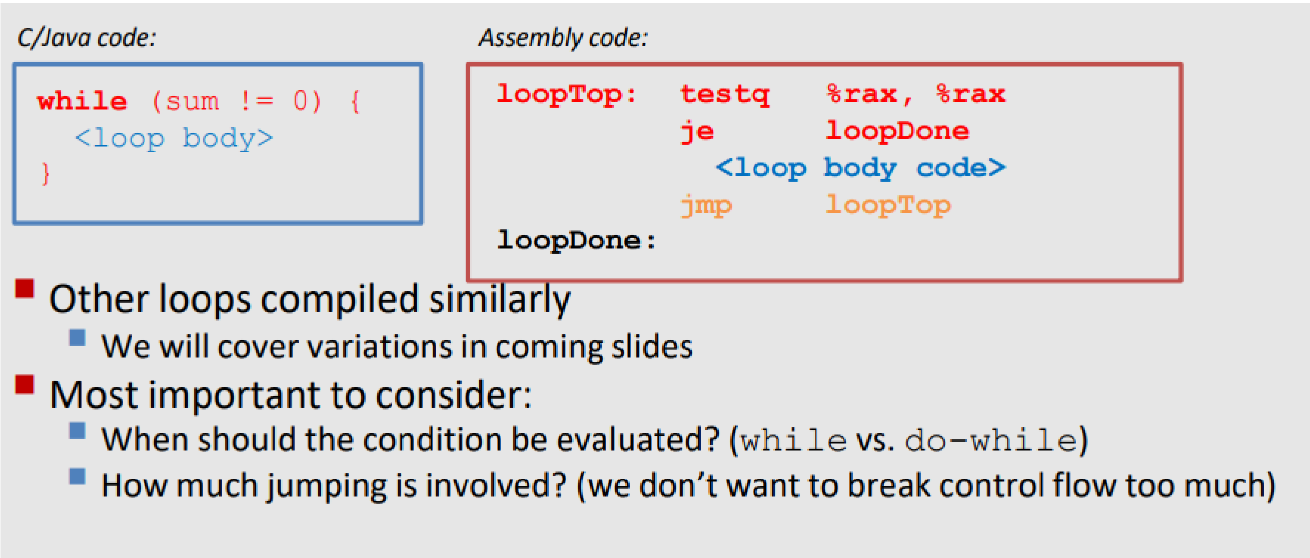
Machine-Level Programming III

Control flow (loops)

Loops

- `do-while`
- `while`
- `for`

Compiling loops



'do-while' loop example

C/Java, factorial

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

`goto` *version, factorial*

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

- Use backward branch to continue looping
- Only take the branch when "`while`" condition holds

"do-while" loop compilation to `asm`

`goto` *version, factorial*

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

Assembly

Register	Use
`%edi`	x
`%eax`	Result

```
fact_do:

  movl   $1, %eax       # eax = 1
.L2:
  imull  %edi, %eax     # eax = eax*edi
  subl   $1, %edi       # edi -= 1
  cmpl   $1, %edi       # compare 1 & edi
  jg     .L2            # jump if greater
  rep ret               # return
```

http://repzret.org/p/repzret/

- Use backward branch to continue looping
- Only take the branch when "`while`" condition holds

General 'do-while' translation

C/Java code

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

goto *version*

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

- Body:

```
{
  Statement_1;
  Statement_2;
  ...
  Statement_n;
}
```

General 'while' translation #1

- *"Jump-to-middle"* translation
- Used with -Og . Recent technique for GCC.

C/Java code

```
While (Test)
  Body
```

goto *version*

```
  goto test;
loop:
  Body
test:
  if (Test)
    goto loop;
done:
```

'while' loop example #1

“while” loop example #1

Register	Use
%edi	Argument x
%eax	Result

C/Java, factorial

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

goto *version*

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

Assembly

```
fact_while:
  movl   $1, %eax
  jmp    .L4
.L3:
  imull  %edi, %eax
  decl   %edi
.L4:
  cmpl   $1, %edi
  jg     .L3
  rep ret
```

- Uses same inner loop as `do-while` version
- Initial `goto` starts loop at `test`

General 'while' translation #2

- *“guarded do”* (`do-while`) conversion
- Used with **–O1**

`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:
```

'while' loop example #2

"while" loop example #2

Register	Use
%edi	Argument x
%eax	Result

C/Java, factorial

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

goto *version*

```
int fact_while(int x){
  int result = 1;
  if (!(x>1))
    goto done;
loop:
  result *= x;
  x = x-1;
  if (x>1)
    goto loop;
done:
  return result;
```

Assembly

```
fact_while:
  movl  $1, %eax
  cmpl  $1, %edi
  jle   .L4
.L3:
  imull %edi, %eax
  subl  $1, %edi
  cmpl  $1, %edi
  jg    .L3
.L4:
  ret
```

- Uses same inner loop as `do-while` version
- Guards loop entry with extra test

Implementing loops

- The machine code generated by GCC for a **for** loops can follow one of the two strategies:
  - *"guarded to"* (`do-while`) strategy
  - *"jump-to-middle"* strategy

'for' loop example

Code factorial – `for` *loop*

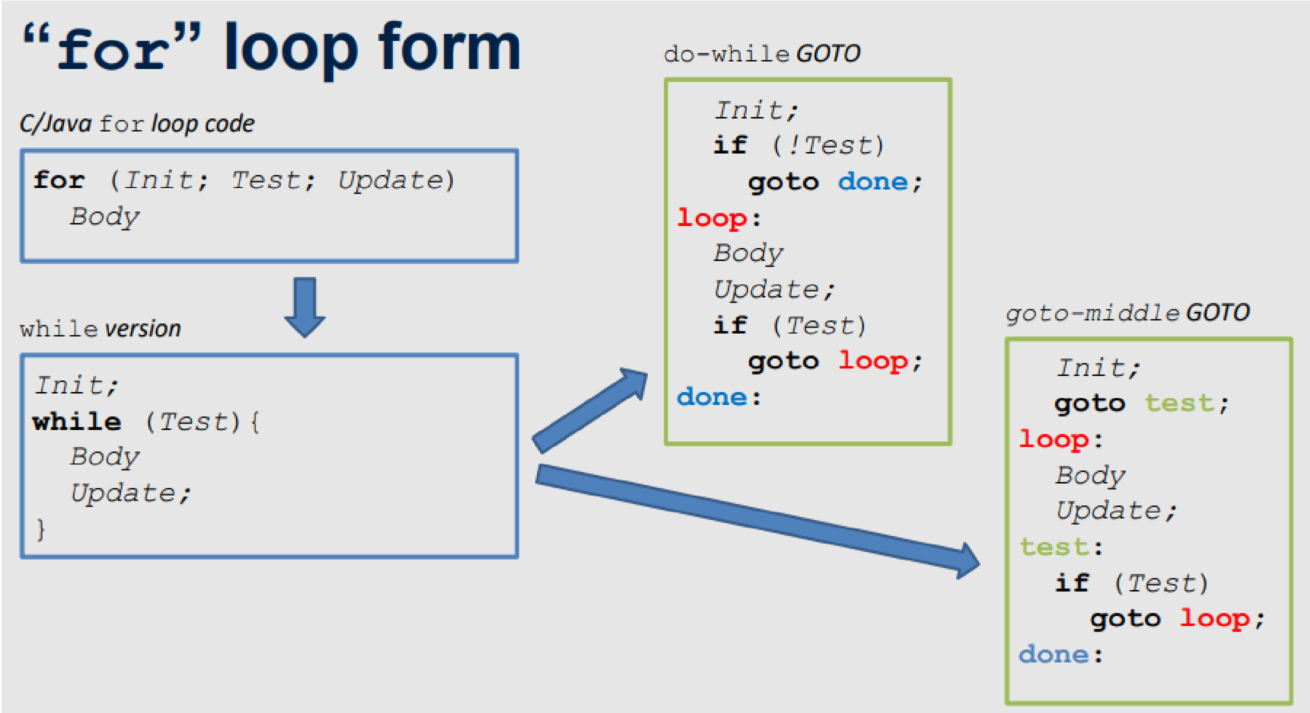
```
int fact_for(int n)
{
  int i;
  int result = 1;
  for (i = 2; i <= n; i++){
    result *= i;
  }
  return result;
}
```

`for` *loop general form*

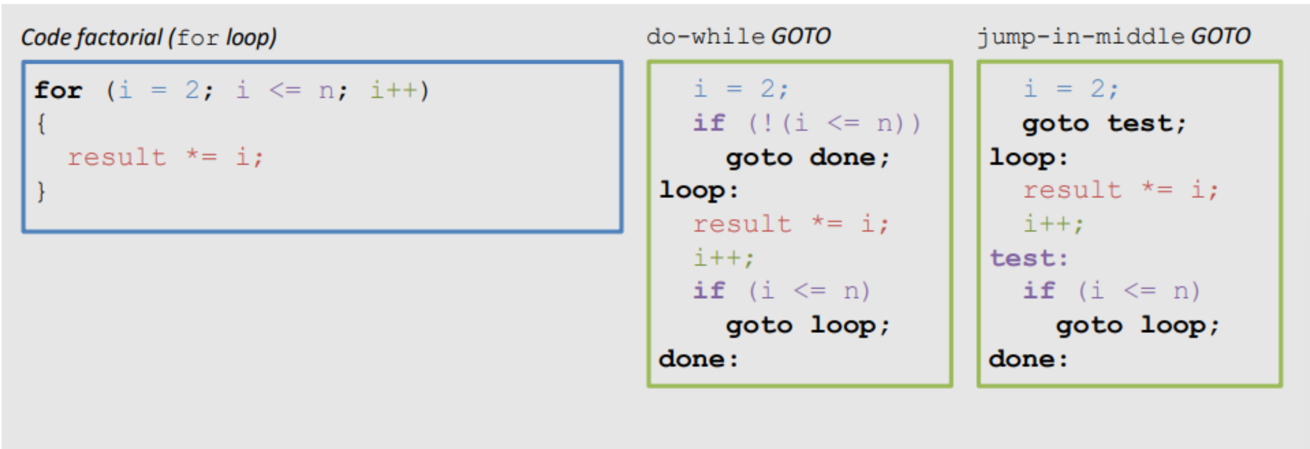
```
for (Init; Test; Update)
  Body
```

```
Init:   i = 2;
Test:   i <= n;
Update: i++;
Body:   result *= i;
```

'for' loop form



'for' loop example: translation to `goto`



Caveat for `break` and `continue`

- C and Java have `break` and `continue`
- Conversion works fine for `break`
  - jump to same label as loop exit condition
- But not for `continue`, would skip doing ***Update,*** which it should do with for-loops.
  - Introduce a new label at ***Update***

Control flow (switch)

switch

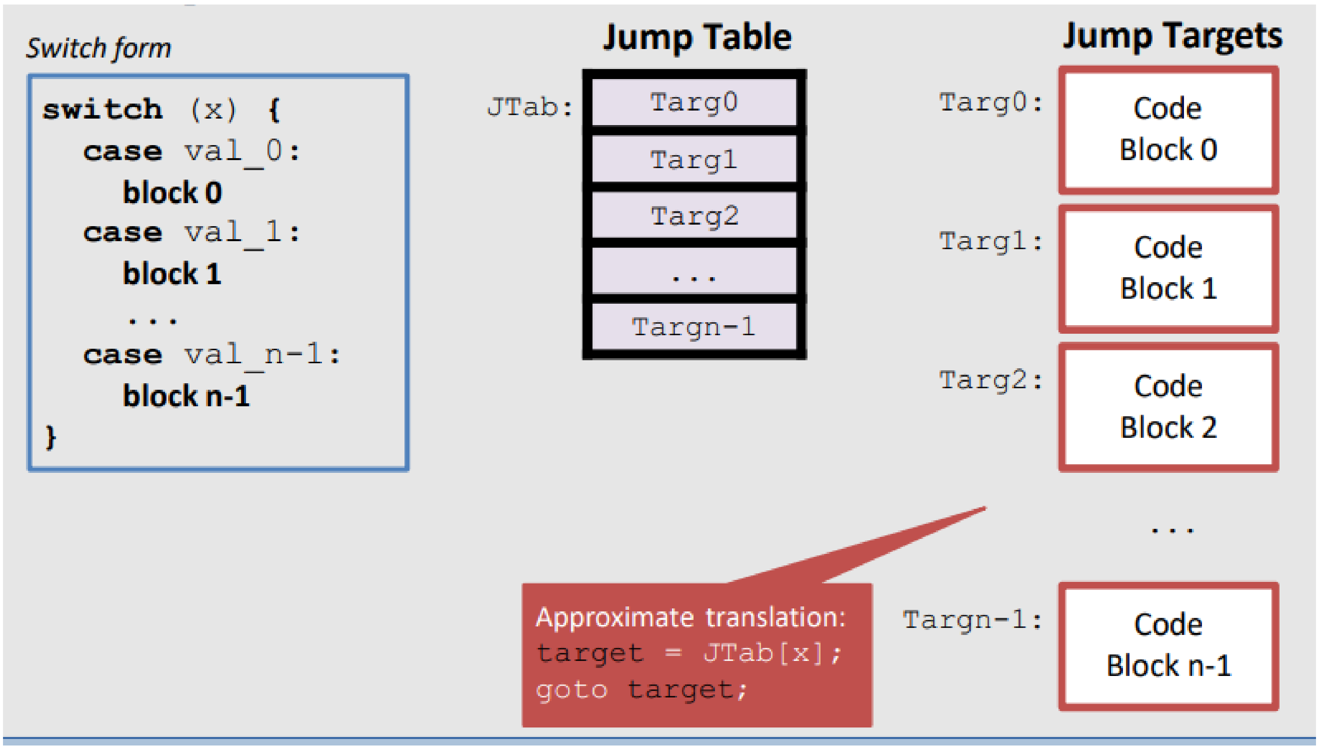
- Compact switch statements
- Sparse switch statements

Compact 'switch' statement example

```
long switch_ex(long x, long y, long z){
    long w = 1;
    switch (x) {
        case 1:
            w = y*z;
            break;
        case 2:
            w = y/z; // fall through
        case 3:
            w += z;
            break;
        case 5:
        case 6:
            w -= z;
            break;
        default:
            w = 2;
    }
    return w;
}
```

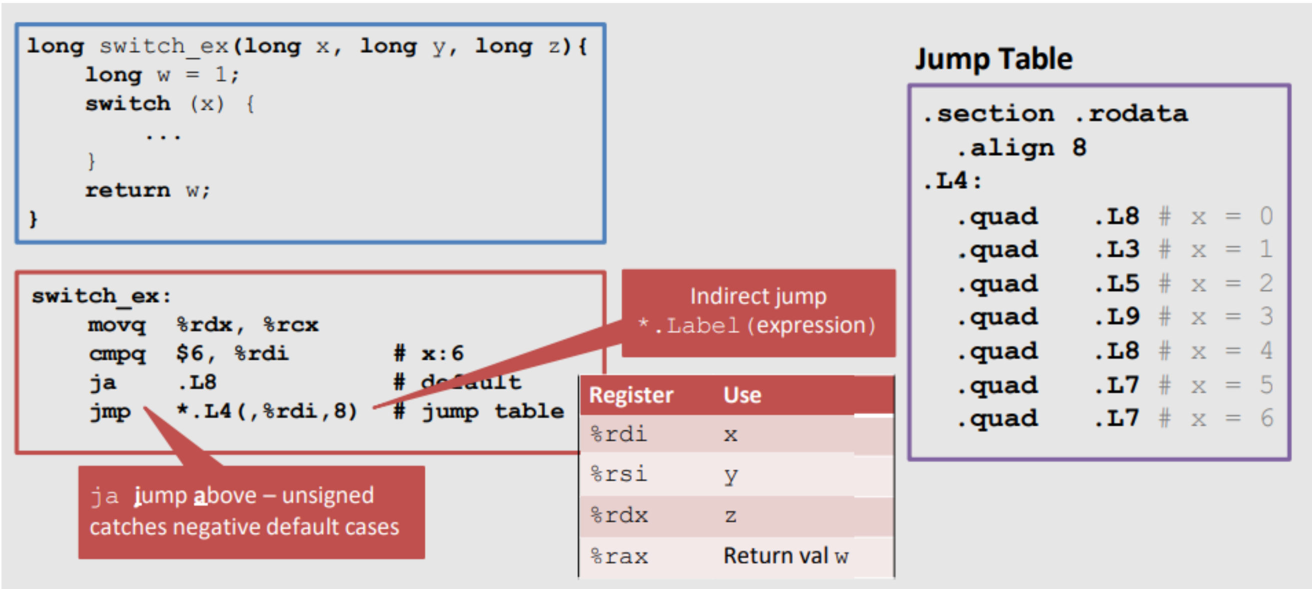
- Multiple case labels
  - here: 5 and 6
- Fall through cases
  - here: 2
- Missing cases
  - here: 4
- Implemented with
  - Jump table
  - Indirect jump instruction

Jump Table structure




```
data Pointers = Anything that refers to memory | 64 bits | quad
word
```

Switch statement example



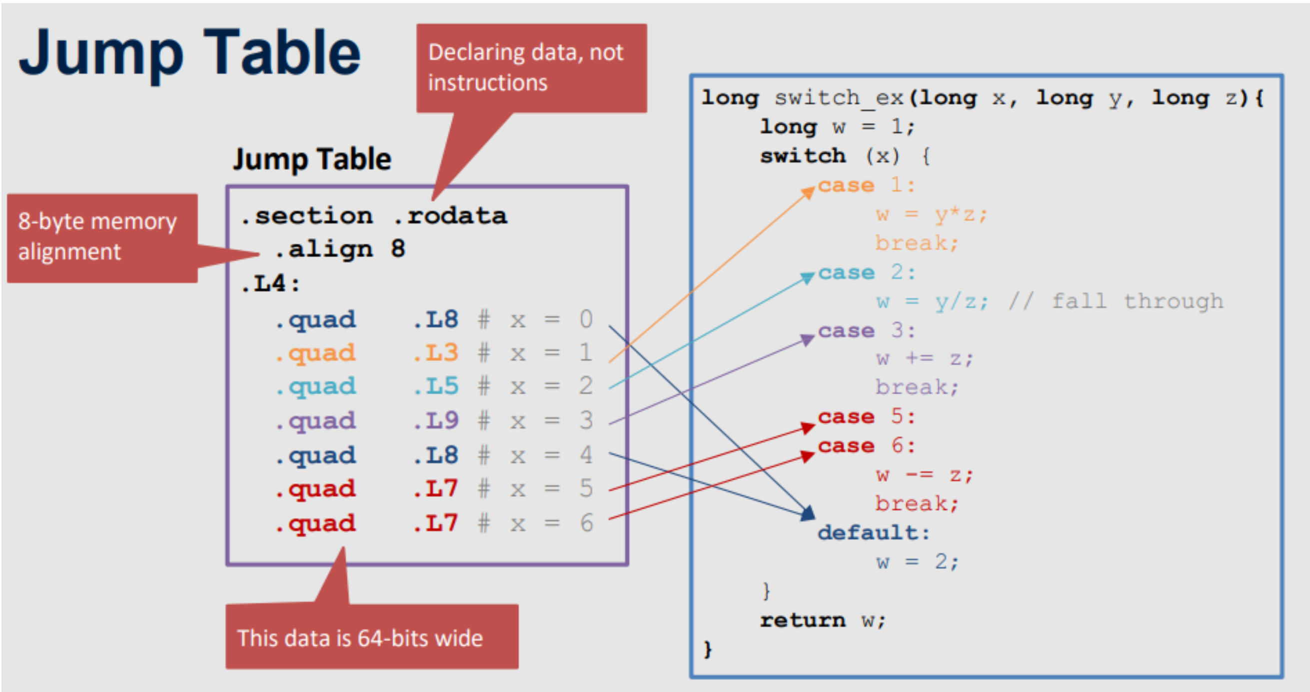
Assembly setup explanation

- **Jump table structure**
  - Each target requires 8 bytes (address)
  - Base address at `.L4`
- **Direct jump:** `jmp .L8`
  - Jump target is denoted by label `.L8`
- **Indirect jump:** `jmp *.L4(,%rdi,8)`
  - Start of jump table: `.L4`
  - Must scale by factor of 8 (addresses are 8 bytes)
  - Fetch target from effective address `.L4 + x*8`
    - only for 0 ≤ x ≤ 6

Jump Table

```
.section .rodata
  .align 8
.L4:
  .quad   .L8 # x = 0
  .quad   .L3 # x = 1
  .quad   .L5 # x = 2
  .quad   .L9 # x = 3
  .quad   .L8 # x = 4
  .quad   .L7 # x = 5
  .quad   .L7 # x = 6
```

Jump Table



Code Blocks (x=1,x=2,x=3)

```
long switch_ex(long x, long y, long z){
    long w = 1;          // .L9
    switch (x) {
        case 1:          // .L3
            w = y*z;
            break;
        case 2:          // .L5
            w = y/z;     // fall through
        case 3:          // .L6
            w += z;
            break;
        . . .
    }
}
```

```
case 2:
    w = y/z;
    goto merge;
```

```
case 3:
    w = 1;
merge:
    w += z;
```

Register	Use
%rdi	x
%rsi	y
%rdx	z
%rax	Return val w

```
.L3:                          # case 1
    movq  %rdx, %rax          # z
    imulq %rsi, %rax          # y*z
    ret                       # break
.L5:                          # case 2
    movq   %rsi, %rax         # y in rax
    cqto                      # div prep
    idivq  %rcx               # y/z
    jmp    .L6                # goto merge
.L9:                          # case 3
    movl   $1, %eax           # w = 1
.L6:                          # merge:
    addq   %rcx, %rax         # w += z
    ret
```

Code Blocks (x=5, x=6, default)

```
long switch_ex(long x, long y, long z){

    long w = 1;

    switch (x) {
        . . .
        case 5:      // .L7
        case 6:      // .L7
            w -= z;
            break;
        default:     // .L8
            w = 2;
    }
    return w;
}
```

Register	Use
%rdi	x
%rsi	y
%rdx	z
%rax	Return val w

```
.L7:                          # case 5,6
    movl   $1, %eax           # w = 1
    subq   %rdx, %rax         # w -= z
    ret
.L8:                          # default:
    movl   $2, %eax           # w = 2
    ret
```

Question

- Would you implement this with a jump table?

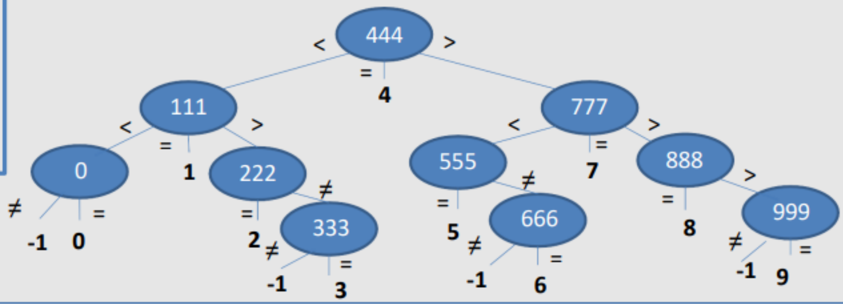
```
switch (x) {
    case 0:     <some code>
                break;
    case 10:    <some code>
                break;
    case 32767: <some code>
                break;
    default:    <some code>
                break;
}
```

- Probably not:
  - 32768-entry jump table is too big (256KiB) for only 4 cases
  - For comparison, text of this statement is 193 B

Sparse `switch` statements

```
int div111(int x){
    switch (x) {
        case 0: return 0;
        case 111: return 1;
        case 222: return 2;
        case 333: return 3;
        case 444: return 4;
        case 555: return 5;
        case 666: return 6;
        case 777: return 7;
        case 888: return 8;
        case 999: return 9;
        default: return -1;
    }
}
```

- Not practical to use a jump table:
  - would require 1000 entries
- A translation into if-then-else (bad practice)
  - would have maximum of 9 tests
- Sparse switch code structure
  - organizes the cases as a binary tree



```
div111:
    cmpl   $444, %edi
    je     .L3
    jl     .L28
    cmpl   $777, %edi
    je     .L10
    jg     .L11
    cmpl   $555, %edi
    movl   $5, %eax
    je     .L1
    cmpl   $666, %edi
    movl   $6, %eax
    jne    .L2
.L1:
    rep ret
```

```
.L28:
    cmpl   $111, %edi
    movl   $1, %eax
    je     .L1
    jl     .L29
    cmpl   $222, %edi
    movl   $2, %eax
    je     .L1
    cmpl   $333, %edi
    movl   $3, %eax
    je     .L1
.L2:
    movl   $-1, %eax
    ret
.L11:
    cmpl   $888, %edi
    movl   $8, %eax
    je     .L1
```

```
    cmpl   $999, %edi
    movl   $9, %eax
    jne    .L2
    rep ret
.L29:
    xorl   %eax, %eax
    testl  %edi, %edi
    jne    .L2
    rep ret
.L10:
    movl   $7, %eax
    ret
.L3:
    movl   $4, %eax
    ret
```

Summary

- **Control flow**
  - `if-then-else`
  - `do-while`
  - `while`, `for`
  - `switch`
- **Assembler control flow**
  - conditional jump
  - conditional move
  - indirect jump
  - compiler must generate assembly
  - code to implement more complex
  - control flow
- **Standard techniques**
  - Loops converted to
    - Guarded to
    - Jump-to-middle
  - Large `switch` statements use jump tables
  - Sparse `switch` statements may use
  - decision trees
- **Conditions in CISC (x86)**
  - CISC machines generally have condition code registers