# Machine-Level Programming: Loops, Switch statements

Arquitectura de Computadores
Departamento de Engenharia Informática
Instituto Superior de Engenharia do Porto

Luís Nogueira (lmn@isep.ipp.pt)

# Loops

- **C provides several looping constructs - namely, *do-while*, *while*, and *for***

- **No corresponding instructions exist in machine code**
  - Instead, combinations of conditional tests and jumps are used to implement the effect of loops

- **We will study the translation of loops as a progression, starting with *do-while* and then working toward ones with more complex implementations**
  - Most compilers generate loop code based on the *do-while* form of a loop

# "Do-While" loop example

- **Count number of 1's in argument `x`**

**C code**

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

**Goto version**

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

- **Use conditional branch to either continue or exit loop**

# "Do-While" loop in Assembly

- **Assume:**

| | |
|---|---|
| %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
```

**Goto version**

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

# General "Do-While" translation

**C code**

```
do{
  statement_1
  ...
  statement_n
}while(Test);
```

Body

**Goto version**

```
loop:
    statement_1
    ...
    statement_n
    if(Test)
      goto loop
```

Body

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

# "While" loop example

**C code**

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

- **It differs from *do-while* in that *test-expr* is first evaluated**
  - The loop is potentially terminated before the first execution of *body-statement*
- **There are a number of ways to translate a *while* loop into machine code**

# General "While" translation #1

- **"Jump-to-middle" translation**
  - Used with **–Og**

**While version**

```
while (Test) {
  Body
}
```

**Goto version**

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

- **Avoids duplicating test code**
- **Unconditional jump incurs no performance penalty on modern CPUs**
  - It occupies a decode unit but never makes it into the main pipeline

# While loop – Jump to middle translation

### C code

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

### Jump to middle version

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

# General "While" translation #2

### While version

```
while(Test) {
  Body
}
```

- **"Do-while" conversion**
  - Used with `-O1`

### 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 – do while translation

### C code

```
int pcount(unsigned int x)
{
  int result = 0;

  while(x){
    result += x & 0x1;
    x >>= 1;
  }

  return result;
}
```

### Goto Version

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

- **The compiler can often optimize the initial test**
  - For example, determining that the test condition will always hold

# General "For" loop form

### General form

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

### Example

```
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 → … → Goto

**For version**

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

**While version**

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

**Do-While version**

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

**Goto Version**

```
  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(unsigned int x)
{
  int i;
  int result = 0;

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

  return result;
}
```

## Goto version

```
int pcount(unsigned int 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;
}
```

# The *loop* instructions

- **Use the ECX register as a counter and automatically decrease its value as the loop instruction is executed**
  - Without affecting the EFLAGS register flag bits when ECX reaches zero

- **Support only an 8-bit offset, so only short jumps can be performed**

| loopX | Condition | Description |
|---|---|---|
| `loop` | `ECX != 0` | Loop until the ECX register is zero |
| `loope/loopz` | `ECX != 0 or ZF` | Loop until either the ECX register is zero, or the ZF flag is not set |
| `loopne/loopnz` | `ECX != 0 and ~ZF` | Loop until either the ECX register is zero, or the ZF flag is set |

# The *loop* instructions example

C code

```
for (i = 100; i > 0; i--)
{
      …
}
```

Assembly *loop* version

```
  movl $100,%ecx
for_loop:
  …
  loop for_loop
```

- **Be careful with code inside the loop**
  - If the ECX register is modified, it will affect the operation of the loop
  - Function calls within the loop can easily trash the value of the ECX register without you knowing it
  - If ECX is already <= 0 before the loop, it will eventually exit when the register overflows

# Today

- Loops
- **Switch statements**

# Switch statements

- **Provide a multi-way branching capability based on the value of an integer index**
  - Particularly useful when dealing with tests where there can be a large number of possible outcome

- **Large blocks are implemented using a *jump table***
  - An array where entry *i* is the address of a code segment implementing the action the program should take when the switch index equals *i*

- **The time taken to perform the switch is independent of the number of switch cases**
  - As opposed to a long sequence of if-else statements

# Example

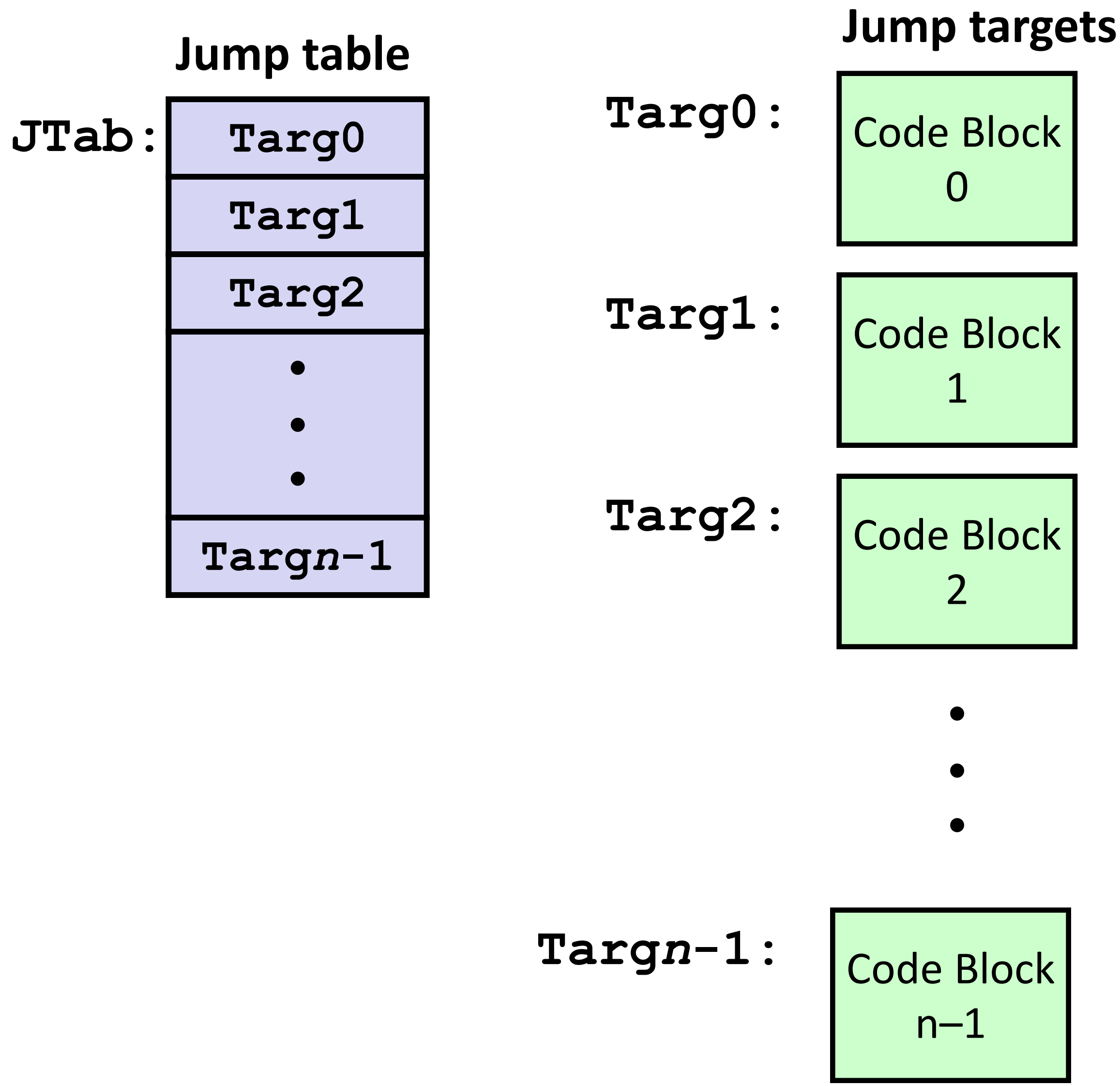
```
int switch_eg(int x, int y, int z){
    int w = 1;
    switch(x) {
    case 1:
        w = y*z;
        break;
    case 2:
        w = y/z;
        /* Fall through */
    case 3:
        w += z;
        break;
    case 5:
    case 6:
        w -= z;
        break;
    default:
        w = 2;
   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

- **Multiple case labels**
  - case 5 & 6
- **Fall through cases**
  - case 2
- **Missing cases**
  - case 4

# Jump table structure

### Switch form

```
switch(x) {
  case val_0:
    Block 0
  case val_1:
    Block 1
    • • •
  case val_n-1:
    Block n–1
}
```




### Approximate translation

```
target = JTab[x]
goto *target;
```

# Switch statement example

```
int switch_eg(int x, int y, int z){
    int w = 1;
    switch(x) {
       . . .
    }
    return w;
}
```

**Setup:**

```
switch_eg:
  pushl %ebp
  movl  %esp,%ebp
  movl  8(%ebp),%eax       # %eax = x
  cmpl  $6,%eax            # Compare x:6
  ja    .L2                # if > goto default
  jmp   *.L7(,%eax,4)      # goto *JTab[x]
```

What range of values takes *default*?

# Switch statement example

```
int switch_eg(int x, int y, int z){
   int w = 1;
   switch(x) {
     . . .
   }
   return w;
}
```

## Jump table

```
.section  .rodata
  .align 4
.L7:
  .int     .L2  # x = 0
  .int     .L3  # x = 1
  .int     .L4  # x = 2
  .int     .L5  # x = 3
  .int     .L2  # x = 4
  .int     .L6  # x = 5
  .int     .L6  # x = 6
```

**Setup:**

```
switch_eg:
  pushl %ebp
  movl  %esp,%ebp
  movl  8(%ebp),%eax       # %eax = x
  cmpl  $6,%eax            # Compare x:6
  ja    .L2                # if > goto default
  jmp   *.L7(,%eax,4)      # goto *JTab[x]
```

← Indirect jump

# Setup explanation

- **Table structure**
  - Each target requires 4 bytes
  - Base address at **`.L7`**

- **Jumping**
  - Direct: **`jmp .L2`**
  - Jump target is denoted by label **`.L2`**

  - Indirect: **`jmp *.L7(,%eax,4)`**
  - Start of jump table: **`.L7`**
  - Must scale by factor of 4 (addresses are 4 bytes on IA32)
  - Fetch target from effective address **`.L7 + %eax*4`**
    - Only for $0 \le x \le 6$

**Jump table**

```
.section   .rodata
  .align 4
.L7:
  .int      .L2  # x = 0
  .int      .L3  # x = 1
  .int      .L4  # x = 2
  .int      .L5  # x = 3
  .int      .L2  # x = 4
  .int      .L6  # x = 5
  .int      .L6  # x = 6
```

# Jump table

```
.section .rodata
  .align 4
.L7:
  .int    .L2 # x = 0
  .int    .L3 # x = 1
  .int    .L4 # x = 2
  .int    .L5 # x = 3
  .int    .L2 # x = 4
  .int    .L6 # x = 5
  .int    .L6 # x = 6
```

```
switch(x) {
case 1:        /* .L3 */
      w = y*z;
      break;
case 2:        /* .L4 */
      w = y/z;
      /* Fall Through */
case 3:        /* .L5 */
      w += z;
      break;
case 5:
case 6:        /* .L6 */
      w -= z;
      break;
default:       /* .L2 */
      w = 2;
}
```

- Duplicates have same label
- Missing cases use label for the default case

# Code blocks (x == 1, default)

```
 switch(x) {
 case 1:        /* .L3 */
        w = y*z;
        break;
  . . .
 default:       /* .L2 */
        w = 2;
}
```

```
.L3:
   movl  16(%ebp),%eax   # z
   imull 12(%ebp),%eax   # w = y*z
   jmp   .L8             # Goto done

.L2:
   movl $2,%eax          # w = 2
   jmp .L8               # Goto done
```

- **Jump table avoids sequencing through cases**
  - Constant time, rather than linear

# Code blocks (x == 2, x == 3)

```
int w = 1;
   . . .
switch(x) {
 . . .
case 2:         /* .L4 */
    w = y/z;
    /* Fall Through */

case 3:         /* .L5 */
    w += z;
    break;
 . . .
}
```

```
.L4:
   movl   12(%ebp),%edx    # y
   movl   %edx,%eax
   sarl   $31, %edx
   idivl  16(%ebp)         # w = y/z
   jmp    .L9

.L5:
   movl   $1, %eax         # w = 1

.L9:
   addl   16(%ebp),%eax    # w += z
   jmp    .L8              # goto done
```

- **Do not initialize `w = 1` unless really need it**
- **Use program sequencing to handle fall-through**

# Handling fall-through

```
    int w = 1;
      . . .
    switch(x) {
      . . .
    case 2:
        w = y/z;
        /* Fall through */
    case 3:
        w += z;
        break;
      . . .
    }
```

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

```
case 3:
        w = 1;
```

```
merge:
        w += z;
```

# Code blocks (x == 5, x == 6)

```
switch(x) {
  . . .
  case 5:     /* .L6 */
  case 6:     /* .L6 */
      w -= z;
      break;
  . . .
}

return w;
```

```
.L6:
  movl   $1,%eax           # w = 1
  subl   16(%ebp),%eax     # w -= z

.L8:                       # done
  movl %ebp,%esp
  popl %ebp
  ret
```

- **Use jump table to handle holes and duplicate tags**

# Summary

- **C control**
  - do-while
  - while, for
  - switch

- **Assembler control**
  - Conditional and unconditional jumps
  - Indirect jump (via jump tables)

- **Standard techniques**
  - Loops converted to do-while form
  - Large switch statements use jump tables