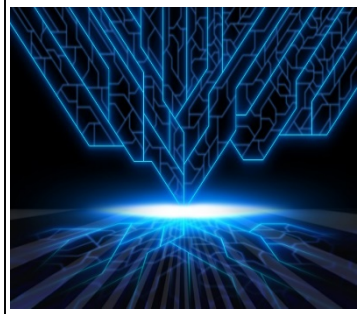
第9.2讲
循环语句



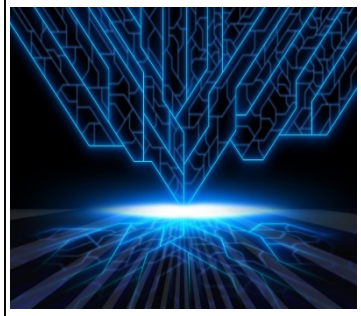
吴海军
南京大学计算机科学与技术系

循环语句

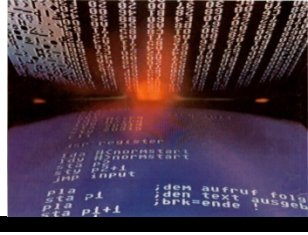
Do –while

While

For

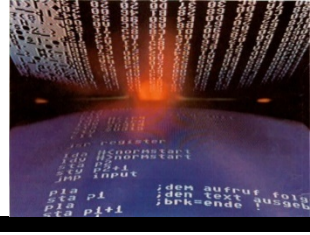


循环语句



- C 语言提供了多种循环结构
  - do-while语句
  - while语句
  - for语句
- 汇编中没有相应的指令存在，用条件测试和跳转组合起来实现循环的效果。
- GCC和其他汇编器产生的循环代码主要基于两种基本的循环模式。

do-while循环



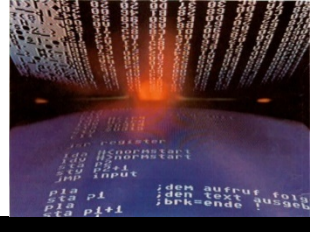
- do-while循环通用形式

```
do
   body-statement
   while (test-expr);
```

  - 重复执行body-statement，对test-expr求值，如果结果为非零，就继续循环。
  - body-statement至少执行一次。
- 翻译成条件和goto语句如下：

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

do-while循环

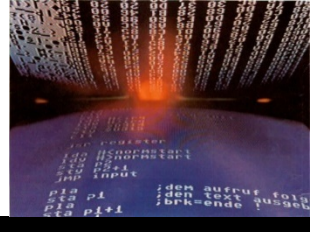


- 用do-while循环计算阶乘函数。

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

```
long fact_do_goto(long n)
 {
    long result = 1;
 loop:
    result*= n;
    n = n-1;
    if (n > 1)
        goto loop;
    return result;
}
```

do-while循环

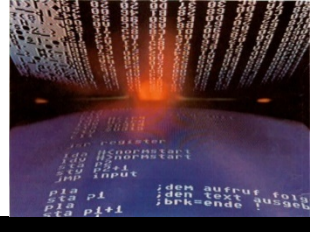


- 用do-while循环计算阶乘函数。

```
long fact_do_goto(long n)
 {
    long result = 1;
 loop:
    result*= n;
    n = n-1;
    if (n > 1)
        goto loop;
    return result;
}
```

```
/*long fact_do(long n)
 n in %rdi */
fact_do:
  movl    $1, %eax
.L2:
  imulq   %rdi,%rax
  subq    $1,%rdi
  cmpq    $1,%rdi
  jg      .L2
  rep;ret
```

while循环



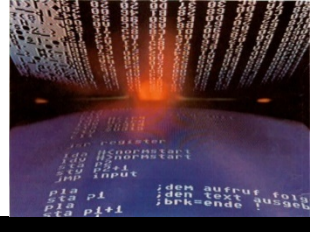
- while语句的通用形式如下：

```
while   (test-expr)
body-statement
```

- 第一种翻译方法跳转到中间，执行一个无条件跳转到循环结尾处的测试，以此来执行初始的测试。

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

while循环

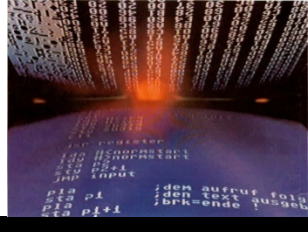


- 用while循环计算阶乘函数。

```
long fact_while (long
n)
 {
    long result = 1;
    while (n > 1) {
        result*= n;
        n = n-1;
    }
    return result;
}
```

```
long fact_while_jm_goto(long n)
 {
    long result = 1;
    goto test;
 loop:
    result*= n;
    n = n-1;
 test:
    if (n > 1)
        goto loop;
    return result;
}
```

while循环

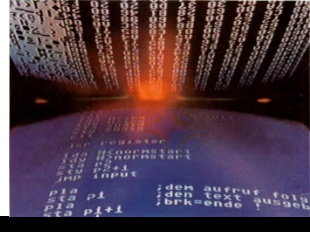


- 用while循环计算阶乘函数。

```
long fact_while_jm_goto(long n)
 {
   long result = 1;
   goto test;
 loop:
   result*= n;
   n = n-1;
 test:
   if (n > 1)
       goto loop;
   return result;
}
```

```
/*long fact_while(long
n)
 n in %rdi */
fact_while:
  movl    $1, %eax
  jmp     .L5
.L6:
  imulq   %rdi,%rax
  subq    $1,%rdi
.L5:
  cmpq    $1,%rdi
  jg      .L6
  rep;ret
```

while循环

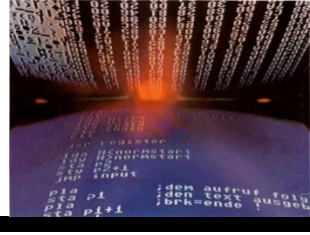


- 第二种翻译方法guarded-do，首先用条件分支， 如果初始条件不成立就跳过循环，把代码变换为do-while循环。
- 当使用较高优化等级编译时， 例如使用命令行选项-O1, GCC会采用这种策略。

```
 t= test-expr;
 if (t)
     goto done;
do:
  body-statement
  while (test-expr)
done:
```

```
 t= test-expr;
 if (t)
     goto done;
loop:
  body-statement
test:
  t= test-expr;
  if (t)
    goto loop;
done:
```

while循环

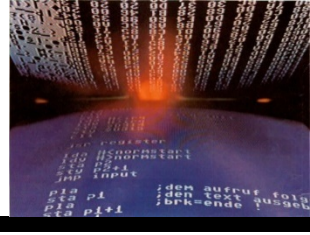


- 用while循环计算阶乘函数。

```
long fact_while (long
n)
 {
    long result = 1;
    while (n > 1) {
        result*= n;
        n = n-1;
    }
    return result;
}
```

```
long fact_while_gd_goto (long n)
 {
    long result = 1;
    if (n <= 1)
       goto done;
 loop:
    result*= n;
    n = n-1;
     if (n! =1)
        goto loop;
 done:
    return result;
}
```

while循环

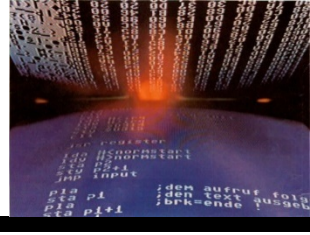


- 用while循环计算阶乘函数。

```
long fact_while_gd_goto (long n)
 {
   long result = 1;
   if (n <= 1)
      goto done;
 loop:
   result*= n;
   n = n-1;
    if (n! =1)
       goto loop;
 done:
   return result;
}
```

```
/*long fact_while(long
n)
 n in %rdi */
fact_while:
  cmpq $1,$rdi
  jle .L7
  movl   $1, %eax
.L6:
  imulq  %rdi,%rax
  subq   $1,%rdi
  cmpq   $1,%rdi
  jne      .L6
  rep; ret
.L7:
  movl   $1,%eax
  ret
```

for循环



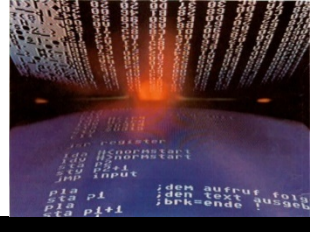
- for语句的通用形式如下：

```
for (init-expr; test-expr; update-expr)
body-statement
```

- 这样一个循环的行为与下面这段使用 while循环的代码的行为一样。

```
init-expr;
while (test-expr)
{
    body-statement
    update-exp;
}
```

for循环



- GCC根据优化的等级为for循环产生的代码是 while 循环的两种翻译之一，

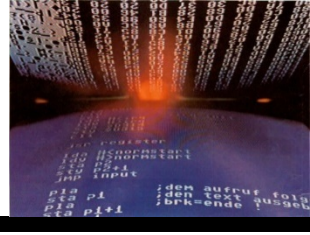
```
   init-expr;
   goto test;
loop:
   body-statement
   update-expr;
test:
   t= test-expr;
   if (t)
     goto loop;
```

跳转到中间策略

```
  init-expr;
  t= test-expr;
  if (!t)
      goto done;
loop:
   body-statement
   update-expr;
   t= test-expr;
  if (t)
     goto loop;
done:
```

guarded-do策略

for循环



- 用for循环写的阶乘函数，转换成while循环。

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

```
long fact_for_while (long
n)
 {
    long i = 2;
    long result = 1;
    while (i <= n) {
        result *= i;
        i++;
    }
return result;
}
```

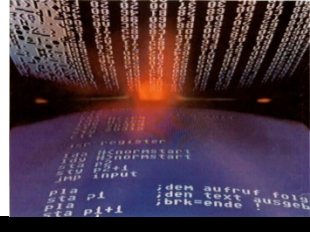
init-expr

test-expr

update-expr

body-statement

for循环

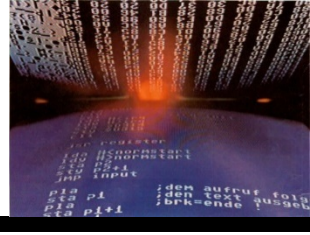


- while循环转成跳转到中间，得到goto代码

```
long fact_for_while (long
n)
 {
   long i = 2;
   long result = 1;
   while (i <= n) {
      result *= i;
      i++;
   }
return result;
}
```

```
long fact_for_while (long
n)  {
   long i = 2;
   long result = 1;
   goto test;
loop:
   result*= i;
   i++;
test:
   if (i<=n)
       goto loop;
   return result;
}
```

for循环

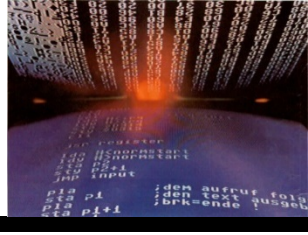


- while循环转成跳转到中间，得到goto代码

```
long fact_for_while (long n)
 {
    long i = 2;
    long result = 1;
    goto test;
loop:
    result*= i;
    i++;
test:
    if (i<=n)
        goto loop;
    return result;
}
```

```
/*long fact_for(long n)
 n in %rdi */
fact_for:
  movl   $1,$eax
  movl   $2,%edx
  jmp    .L8
.L9:
  imulq  %rdi,%rax
  addq   $1,%rdx
.L8:
  cmpq   %rdi,%rdx
  jle      .L9
  rep; ret
```

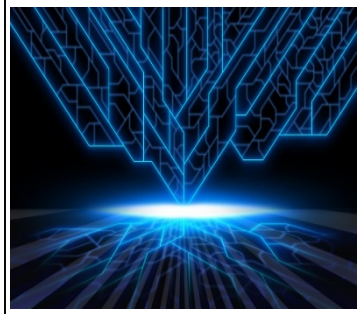
for循环



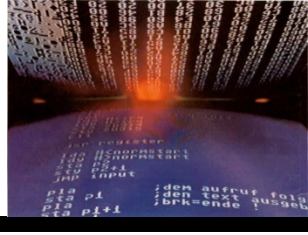
- C语言中三种形式的所有的循环 do-while、while和for—用一种简单的策略来翻译，产生包含一个或多个条件分支的代码。
- 控制的条件转移提供了将循环翻译成机器代码的基本机制。

开关语句

switch



switch语句



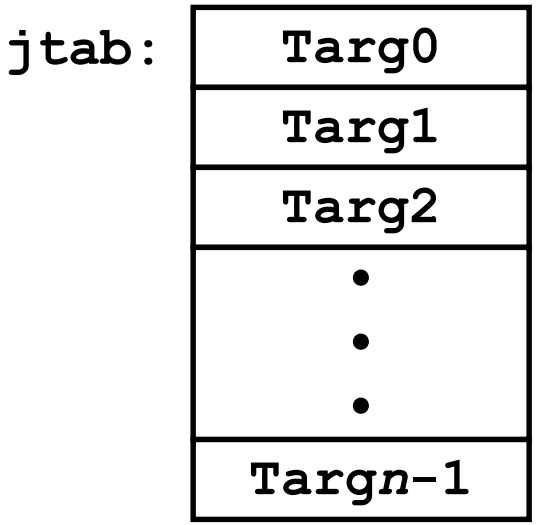
- 根据一个整数索引值进行多重分支
- 跳转表 jump table
  - 是一个数组
  - 表项i是一个代码段的地址
- 执行一个开关语句的时间和开关情况的数量无关。
- GCC
  - 根据开关的数量和开关情况值的稀疏程度来选择一种实现
  - 当开关情况数量比较多(>4)，并且值得范围跨度比较小的时候，使用跳转表。

跳转表结构

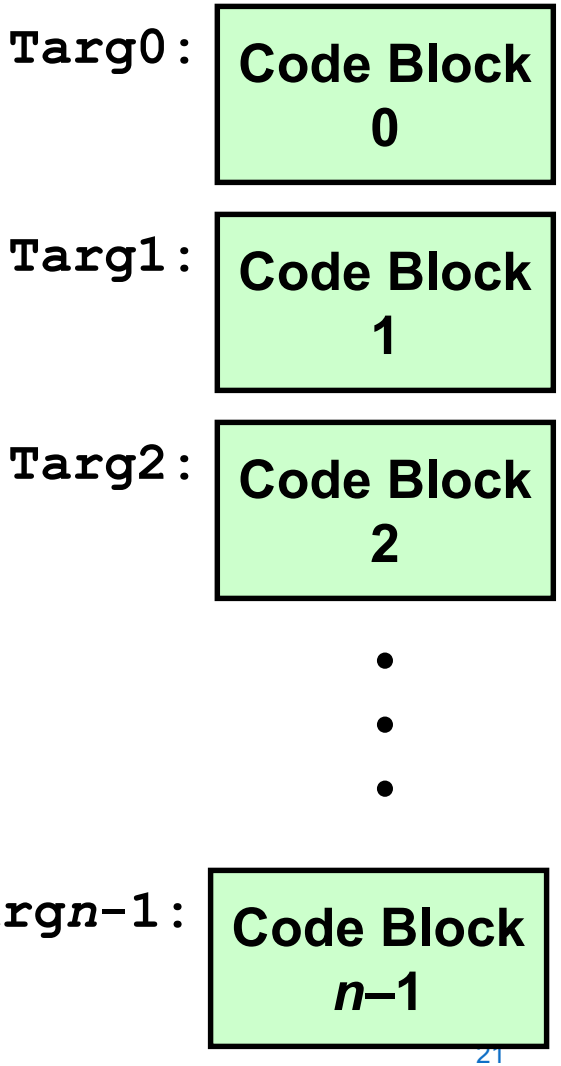
Switch结构

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

跳转表

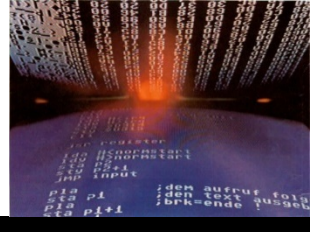


跳转目标



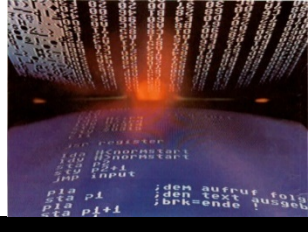
Approx. Translation

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

- switch语句的示例

```
void switch_eg
   (long x, long y, long
*dest)
{
    long val = x;
    switch(n) {
    case 100:
        val *=13;
        break;
    case 102:
        val +=10;
        /* Fall Through */
    case 103:
        val +=11;
        break;
    case 104:
    case 106:
        val *=val;
        break;
    default:
        val = 0;
    }
    *dest=val;
```

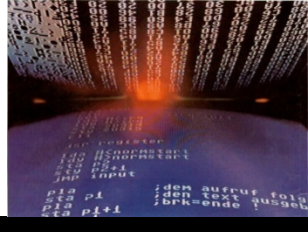



```
void switch_eg_impl(long x, long n, long
*dest)
{
  /* Table of code pointers•/
 static void *jt [7] = {
   &&loc_A, &&loc_def, &&loc_B,
&&loc_C,   &&loc_D, &&loc_def,
 &&loc_D
};
unsigned long index=n-100;
  long val;
  if (index> 6)
      goto loc_def;
/* Multiway branch */
   goto *jt[index];
```

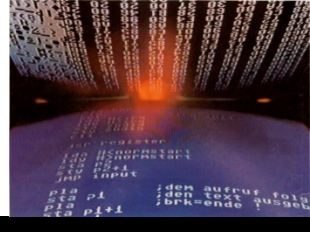
地址表

新索引项

```
loc_A:                  /* Case 100 */
    val= x * 13;
    goto done;
loc_B:                  /* Case 102 */
   val=x +10;
loc_C:                  /* Case 103 */
    val=x +11;
 goto done;
loc_D:                  /* Case 104、106 */
   val =x*x;
   goto done;
loc_def :               /* Default case */
      val = 0;
done:
    *dest=val;
}
```

```
void switch_eg(long x, long n,
long *dest) x in %rdi, n in
%Rsi, dest in %rdx
switch_eg:
    subq  $100, %rsi
    cmpq  $6, %rsi
    ja    .L8
    jmp    *.L4(,%rsi,8)
.L3:       /* loc_A */
    leaq
    leaq
    jmp   .L2
.L5:       /* loc_B */
    addq $10, %rdi
.L6:     /* loc_C */
    addq $11, %rdi
    jmp   .L2
.L7:     /* loc_D */
    imulq  %rdi, %rdi
    jmp   .L2
.L8:     /* loc_def */
    movl $0, %edi
.L2:    /* done */
    movq %rdi, (%rdx)
    ret
```

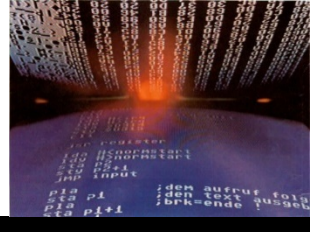



- 构建跳转表，给每一个索引项建立代码位置指针。
- 重复表项使用相同的代码标号
- 缺失的表项使用缺省的代码标号
- 建立在只读数据区.rodata的目标代码文件段中

&&loc_A,
&&loc_def,
&&loc_B,
&&loc_C,
&&loc_D,
&&loc_def,
&&loc_D

```
.section  .rodata
.align 8
.L4:
  .quad    .L3 # case =100
  .quad    .L8 # case =101
  .quad    .L5 # case =102
  .quad    .L6 # case =103
  .quad    .L7 # case =104
  .quad    .L8 # case =105
  .quad    .L7 # case =106
```

从二进制代码中抽取跳转表



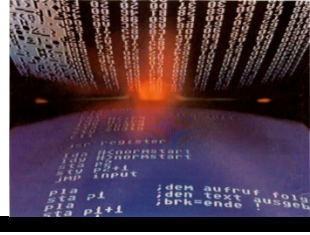
- 跳转表存储在只读数据段 (.rodata)
  - 代码中所需的各种各样的固定值
- 可以用objdump来检查

  `objdump code-examples -s --section=.rodata`
  - 显示给出的段中所有数据
- 很难读懂
  - 跳转表都是用反向的字节顺序显示。 Why?

```
.rodata一部分内容:
 8048bc0 30870408 37870408 40870408 47870408  0...7...@...G...
 8048bd0 50870408 57870408 46616374 28256429  P...W...Fact(%d)
 8048be0 203d2025 6c640a00 43686172 203d2025   = %ld..Char = %
 …
```

数据存放采用小端模式
0x30870408 实际上是0x08048730

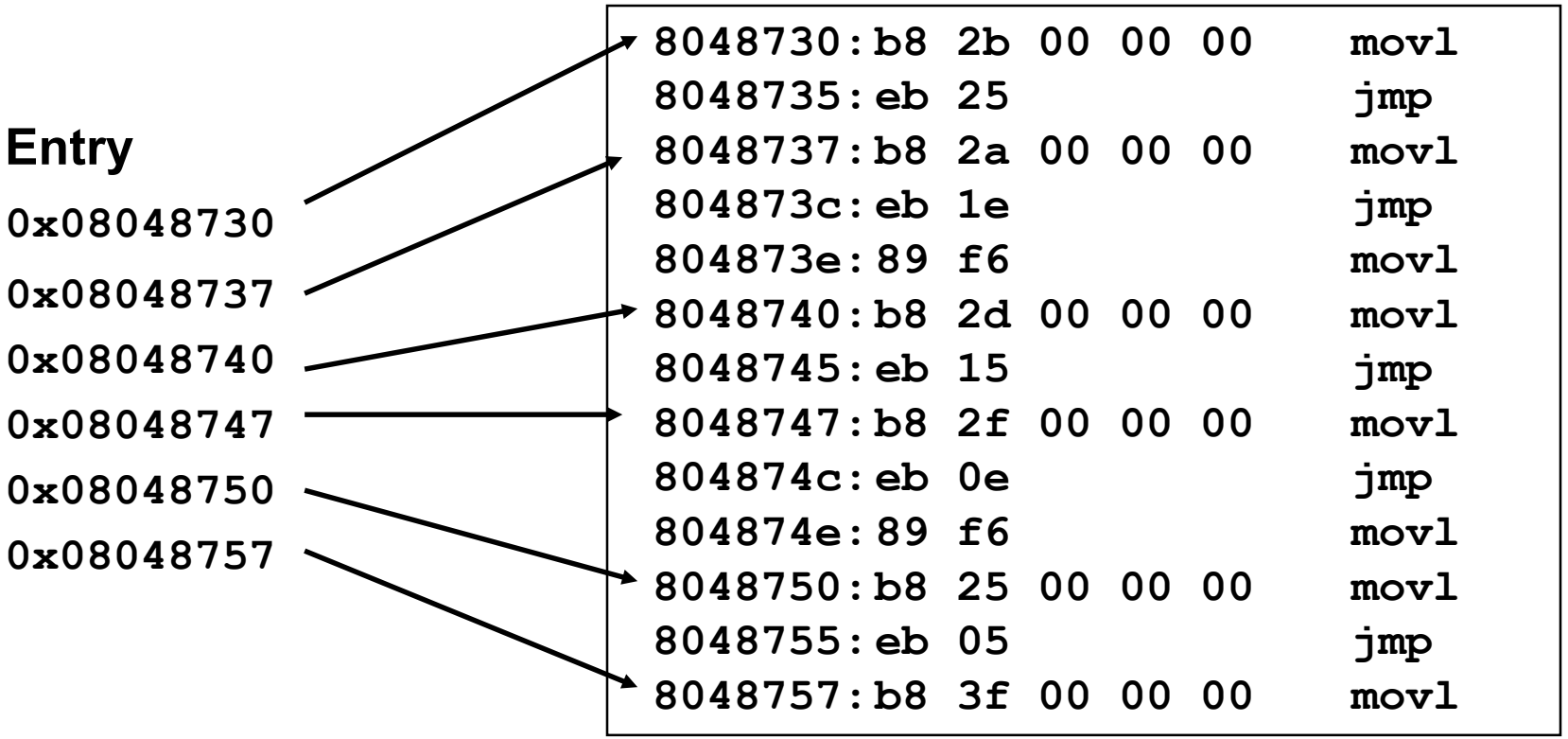
反汇编目标



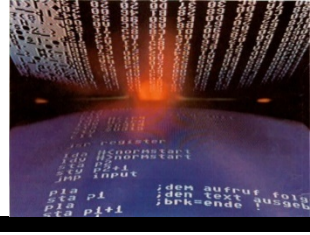
```
8048730:b8 2b 00 00 00   movl   $0x2b,%eax
8048735:eb 25            jmp    804875c <unparse_symbol+0x44>
8048737:b8 2a 00 00 00   movl   $0x2a,%eax
804873c:eb 1e            jmp    804875c <unparse_symbol+0x44>
804873e:89 f6            movl   %esi,%esi
8048740:b8 2d 00 00 00   movl   $0x2d,%eax
8048745:eb 15            jmp    804875c <unparse_symbol+0x44>
8048747:b8 2f 00 00 00   movl   $0x2f,%eax
804874c:eb 0e            jmp    804875c <unparse_symbol+0x44>
804874e:89 f6            movl   %esi,%esi
8048750:b8 25 00 00 00   movl   $0x25,%eax
8048755:eb 05            jmp    804875c <unparse_symbol+0x44>
8048757:b8 3f 00 00 00   movl   $0x3f,%eax
```

- `movl %esi,%esi` 指令什么也不做
- 插入是为了对齐指令来达到更好的Cache性能

与反汇编目标对应


Entry
0x08048730
0x08048737
0x08048740
0x08048747
0x08048750
0x08048757
8048730:b8 2b 00 00 00 movl
8048735:eb 25 jmp
8048737:b8 2a 00 00 00 movl
804873c:eb 1e jmp
804873e:89 f6 movl
8048740:b8 2d 00 00 00 movl
8048745:eb 15 jmp
8048747:b8 2f 00 00 00 movl
804874c:eb 0e jmp
804874e:89 f6 movl
8048750:b8 25 00 00 00 movl
8048755:eb 05 jmp
8048757:b8 3f 00 00 00 movl

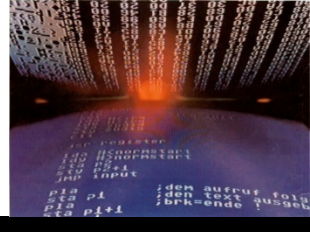
稀疏Switch例子



```
/* Return x/111 if x is multiple
   && <= 999.  -1 otherwise */
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;
  }
}
```

- 用跳转表不实用
  - 需要1000个表项
- 翻译成if-then-else 结构需要最多9个比较

稀疏Switch代码



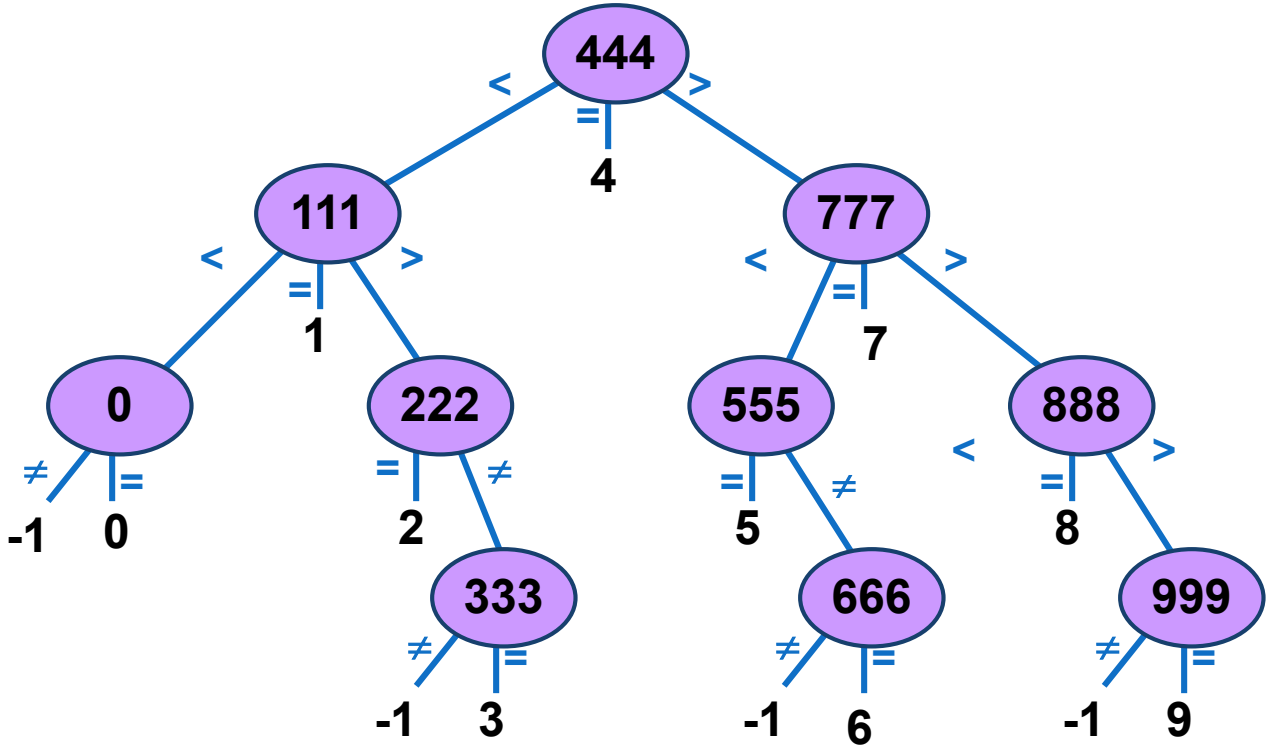
```
    movl 8(%ebp),%eax     # get x
    cmpl $444,%eax    # x:444
    je L8
    jg L16
    cmpl $111,%eax    # x:111
    je L5
    jg L17
    testl %eax,%eax   # x:0
    je L4
    jmp L14

    . . .
```

```
    . . .
L5:
    movl $1,%eax
    jmp L19
L6:
    movl $2,%eax
    jmp L19
L7:
    movl $3,%eax
    jmp L19
L8:
    movl $4,%eax
    jmp L19
    . . .
```

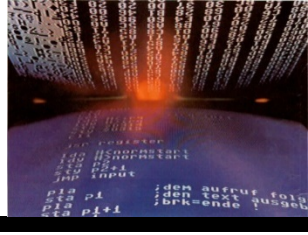
- 把 x 与可能的case值比较
- 根据结果跳到不同的地方

稀疏Switch代码结构



- 把所有情况组织成二叉树
- 对数性能

总结



- C 控制
  - if-then-else
  - do-while
  - while
  - switch
- 汇编控制
  - jump
  - Conditional jump
- 编译器
  - 必须能够生成汇编代码来实现更复杂的控制
- 标准技术
  - 所有循环转换到do-while形式
  - 情况数量多的switch语句用跳转表