15-213

"The course that gives CMU its Zip!"

Machine-Level Programming II
Control Flow
Sept. 10, 1998

Topics

- **Control Flow**
  - Varieties of Loops
  - Switch Statements

Alpha Register Convention

General Purpose Registers

- **32 total**
- **Store integers and pointers**
- **Fast access: 2 reads, 1 write in single cycle**

Usage Conventions

- **Established as part of architecture**
- **Used by all compilers, programs, and libraries**
- **Assures object code compatibility**
  - e.g., can mix Fortran and C

Name	Register	Use
`v0`	`$0`	Return value from integer functions
`t0`	`$1`	Temporaries (not preserved across procedure calls)
`t1`	`$2`	
`t2`	`$3`	
`t3`	`$4`	
`t4`	`$5`	
`t5`	`$6`	
`t6`	`$7`	
`t7`	`$8`	
`s0`	`$9`	Callee saved
`s1`	`$10`	
`s2`	`$11`	
`s3`	`$12`	
`s4`	`$13`	
`s5`	`$14`	
`s6,fp`	`$15`	Frame pointer, or callee saved

Registers (cont.)

Important Ones for Now

$0	**Return Value**
$1…$8	**Temporaries**
$16	**First argument**
$17	**Second argument**
$26	**Return address**
$31	**Constant 0**

a0	$16	Integer arguments
a1	$17	
a2	$18	
a3	$19	
a4	$20	
a5	$21	
t8	$22	Temporaries
t9	$23	
t10	$24	
t11	$25	
ra	$26	Return address
pv,t12	$27	Current proc addr or Temp
AT	$28	Reserved for assembler
gp	$29	Global pointer
sp	$30	Stack pointer
zero	$31	Always zero

“Do-While” Loop Example

C Code

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

Goto Version

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

- **C allows “goto” as means of transferring control**
  - Closer to machine-level programming style
- **Generally considered bad coding style**

“Do-While” Loop Compilation

Goto Version

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

Assembly

```
   bis $31,1,$0      # result = 1
$37:                 # loop:
   mulq $0,$16,$0    # result *= x
   subq $16,1,$16    # x = x-1
   cmple $16,1,$1    # if !(x<=1)
   beq $1,$37        # goto loop
   ret $31,($26),1   # return
```

Registers

```
$16   x
$0    result
```

New Instructions

```
bis   a, b, c    c = a | b
cmple a, b, c    c = a <= b
```

General "Do-While" Translation

C Code

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

Goto Version

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

- ***Body* can be any C statement**
  - Typically compound statement:

```
{
  Statement_1;
  Statement_2;
    …
  Statement_n;
}
```

- ***Test* is expression returning integer**

  = 0 interpreted as false   ≠ 0 interpreted as true

“While” Loop Example

C Code

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

First Goto Version

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

- Is this code equivalent to the do-while version?
- Must jump out of loop if test fails

Actual "While" Loop Translation

C Code

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

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

Second Goto Version

```
long int fact_while_goto2
  (long int x)
{
  long int result = 1;
  if (!(x > 1))
    goto done;
loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto loop;
done:
  return result;
}
```

General "While" Translation

C Code

```
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

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

Algorithm

- **Exploit property that** $p = p_0 + 2p_1 + 4p_2 + \ldots\, 2^{n-1}p_{n-1}$
- **Gives:** $x^p = z_0 \cdot z_1{}^2 \cdot (z_2{}^2)^2 \cdot \ldots \cdot (\ldots((z_{n-1}{}^2)^2)\ldots)^2$ — $\underbrace{\qquad}_{n \text{ times}}$

  $z_i = 1$ **when** $p_I = 0$

  $z_i = x$ **when** $p_I = 1$

  ***n* times**
- **Complexity** $\mathbf{O(log}\ p\mathbf{)}$

"While" → "Do-While " → "Goto "

```
  long int result = 1;
  while (p) {
    if (p & 0x1)
      result *= x;
    x = x*x;
    p = p>>1;
  }
```

↓

```
  long int result = 1;
  if (!p) goto done;
  do {
    if (p & 0x1)
      result *= x;
    x = x*x;
    p = p>>1;
  } while (p);
done:
```

→

```
  long int result = 1;
  if (!p)
    goto done;
loop:
  if (!(p & 0x1))
    goto skip;
  result *= x;
skip:
  x = x*x;
  p = p>>1;
  if (p)
    goto loop;
done:
```

- **Also converted conditional update into test and branch around update code**

Example #2 Compilation

Goto Version

```
  long int result = 1;
  if (!p)
    goto done;
loop:
  if (!(p & 0x1))
    goto skip;
    result *= x;
skip:
    x = x*x;
    p = p>>1;
  if (p)
    goto loop;
done:
```

Assembly

```
   bis $31,1,$0        # result = 1
   beq $17,$52         # if p=0
                       # goto done
$53:                   # loop:
   blbc $17,$54        # if (p&0x1)
                       # goto skip
   mulq $0,$16,$0      # result *= x
$54:                   # skip:
   mulq $16,$16,$16    # x *= x
   srl $17,1,$17       # p = p>>1
   bne $17,$53         # if p != 0
                       # goto loop
$52:                   # done
   ret $31,($26),1     # return
```

"For" Loop Example

```
long int result;
for (result = 1;
     p != 0;
     p = p>>1) {
  if (p & 0x1)
    result *= x;
  x = x*x;
}
```

General Form

```
for (Init; Test; Update )

    Body
```

Init	*Test*	*Update*
`result = 1`	`p != 0`	`p = p >> 1`

Body

```
{
  if (p & 0x1)
    result *= x;
  x = x*x;
}
```

"For" → "While"

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 Compilation

Goto Version

```
  Init;
  if (!Test)
    goto done;
loop:
  Body
  Update ;
  if (Test)
    goto loop;
done:
```

→

```
  result = 1;
  if (p == 0)
    goto done;
loop:
  if (p & 0x1)
    result *= x;
  x = x*x;
  p = p >> 1;
  if (p != 0)
    goto loop;
done:
```

Init

```
result = 1
```

Test

```
p != 0
```

Update

```
p = p >> 1
```

Body

```
  {
    if (p & 0x1)
      result *= x;
    x = x*x;
  }
```

Compiling Switch Statements

```
typedef enum
  {ADD, MULT, MINUS, DIV, MOD, BAD}
    op_type;

char unparse_symbol(op_type op)
{
  switch (op) {
  case ADD :
    return '+';
  case MULT:
    return '*';
  case MINUS:
    return '-';
  case DIV:
    return '/';
  case MOD:
    return '%';
  case BAD:
    return '?';
  }
}
```

Implementation Options

- **Series of conditionals**
  - Good if few cases
  - Slow if many
- **Jump Table**
  - Lookup branch target
  - Avoids conditionals
  - Possible when cases are small integer constants
- **GCC**
  - Picks one based on case structure
- **Bug in example code**
  - No default given

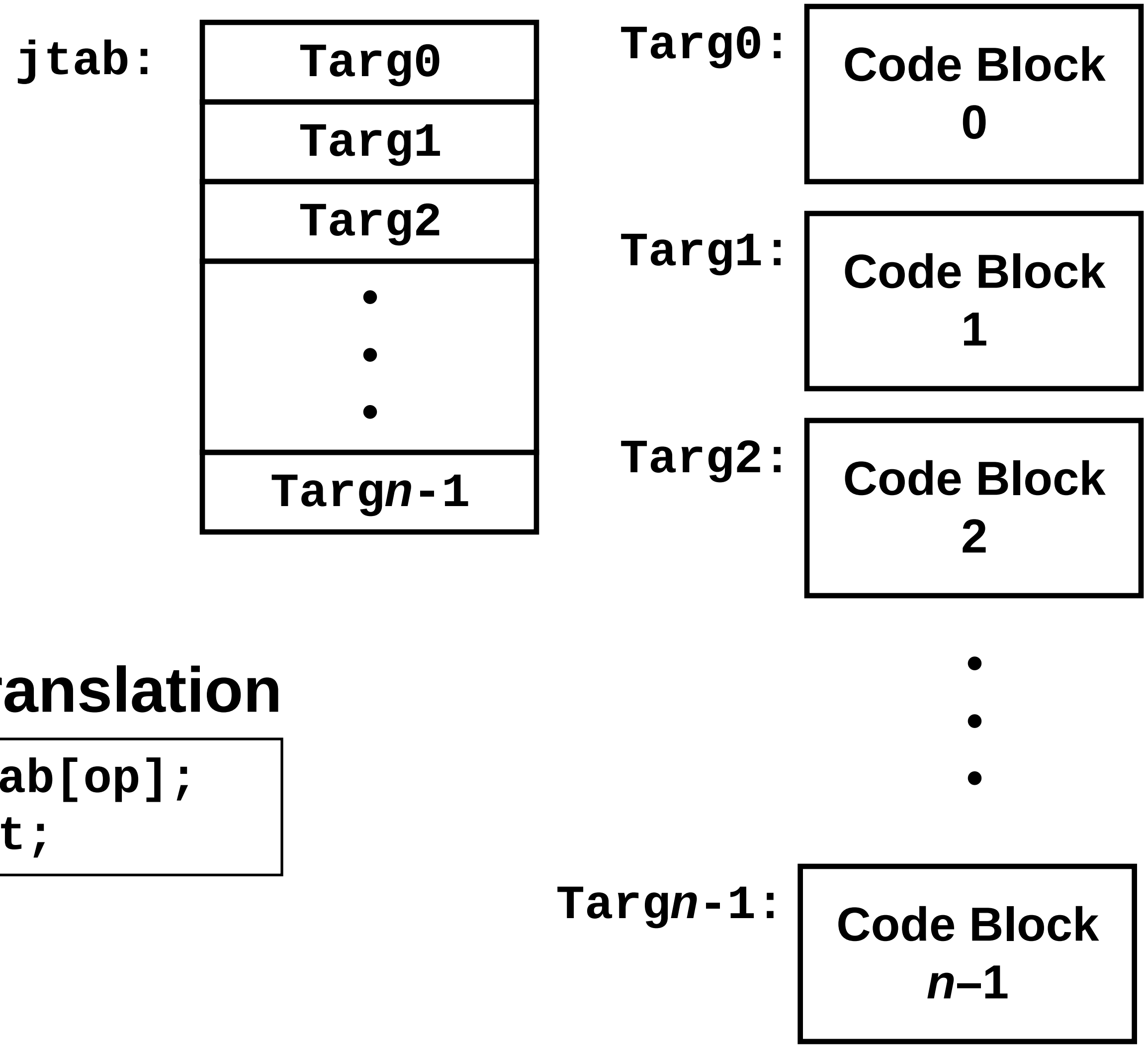
Jump Table Structure

Switch Form

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

Jump Table

Jump Targets



Approx. Translation

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

Switch Statement Example

Branching Possibilities

```
typedef enum
  {ADD, MULT, MINUS, DIV, MOD, BAD}
    op_type;

char unparse_symbol(op_type op)
{
  switch (op) {
    • • •
  }
}
```

Enumerated Values

```
ADD   0
MULT  1
MINUS 2
DIV   3
MOD   4
BAD   5
```

Setup:

Check for out-of-range cases

Set up jump

```
# op in $16
zapnot $16,15,$16 # zero upper 32 bits
cmpule $16,5,$1   # if (op > 5) then
beq $1,$66        #  branch to return
lda $1,$74        # $1 = &jtab[0] - $gp
s4addq $16,$1,$1  # $1 = &jtab[op] - $gp
ldl $1,0($1)      # $1 = jtab[op] - $gp
addq $1,$29,$2    # $2 = jtab[op]
jmp $31,($2),$68  # jump to *jtab[op]
```

Assembly Setup Explanation

Instructions

`zapnot a, b, c`

Use low order byte of **`b`** as mask **`m`**

`byte(c,i) = m[i] ? byte(a,i) : 0`

`cmpule a, b, c`

`c = ((unsigned long) a <= (unsigned long) b)`

Symbolic Labels

- **Labels of form $*XX* translated into addresses by assembler**

Table Structure

- **Each target requires 4 bytes**
- **Base address of `jtab` at `$gp + $74`**

Jump Table

Table Contents

```
$74:
     .gprel32 $68
     .gprel32 $69
     .gprel32 $70
     .gprel32 $71
     .gprel32 $72
     .gprel32 $73
```

Enumerated Values

```
ADD    0
MULT   1
MINUS  2
DIV    3
MOD    4
BAD    5
```

Targets & Completion

```
$68:
    bis $31,43,$0   # return '+'
    ret $31,($26),1
$69:
    bis $31,42,$0   # return '*'
    ret $31,($26),1
$70:
    bis $31,45,$0   # return '-'
    ret $31,($26),1
$71:
    bis $31,47,$0   # return '/'
    ret $31,($26),1
$72:
    bis $31,37,$0   # return '%'
    ret $31,($26),1
$73:
    bis $31,63,$0   # return '?'
$66:
    ret $31,($26),1
```