15-213

Machine-Level Programming II
Control Flow
Feb. 1, 2001

Topics

- **Condition Codes**
  - Setting
  - Testing
- **Control Flow**
  - If-then-else
  - Varieties of Loops
  - Switch Statements

Condition Codes

Single Bit Registers

`CF` **Carry Flag**

`ZF` **Zero Flag**

`SF` **Sign Flag**

`OF` **Overflow Flag**

Implicit Setting By Arithmetic Operations

`addl` ***Src*,*Dest***

C analog: `t = a+b`

- **CF set if carry out from most significant bit**
  - Used to detect unsigned overflow
- **ZF set if** `t == 0`
- **SF set if** `t < 0`
- **OF set if two's complement overflow**

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

Not Set by `leal` instruction

Setting Condition Codes (cont.)

Explicit Setting by Compare Instruction

`cmpl` *Src2*,*Src1*

- **`cmpl b,a` like computing `a-b` without setting destination**
- **CF set if carry in/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)`

Explicit Setting by Test instruction

`testl` *Src2*,*Src1*

- **Sets condition codes based on value of *Src1* & *Src2***
  - Useful to have one of the operands be a mask
- **`testl b,a` like computing `a&b` without setting destination**
- **ZF set when `a&b == 0`**
- **SF set when `a&b < 0`**

Reading Condition Codes

SetX Instructions

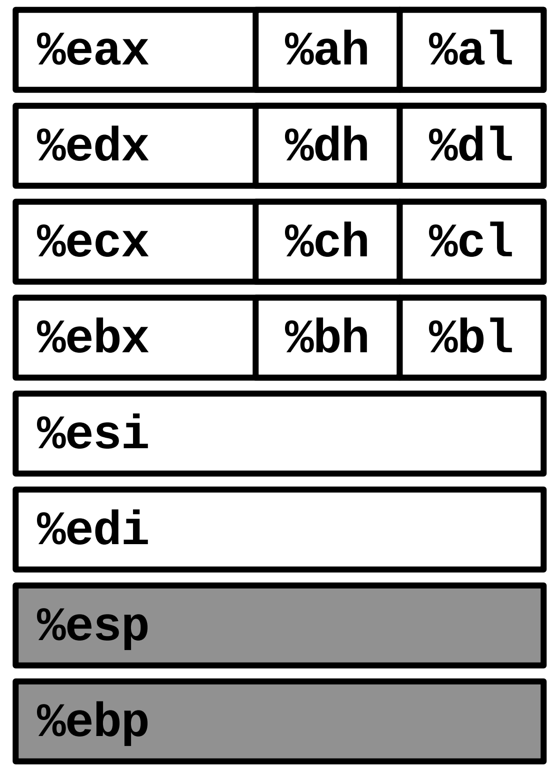
- Set single byte based on combinations of condition codes

SetX	Condition	Description
`sete`	`ZF`	Equal / Zero
`setne`	`~ZF`	Not Equal / Not Zero
`sets`	`SF`	Negative
`setns`	`~SF`	Nonnegative
`setg`	`~(SF^OF)&~ZF`	Greater (Signed)
`setge`	`~(SF^OF)`	Greater or Equal (Signed)
`setl`	`(SF^OF)`	Less (Signed)
`setle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`seta`	`~CF&~ZF`	Above (unsigned)
`setb`	`CF`	Below (unsigned)

Reading Condition Codes (Cont.)

SetX Instructions

- **Set single byte based on combinations of condition codes**
- **One of 8 addressable byte registers**
  - Embedded within first 4 integer registers
  - Does not alter remaining 3 bytes
  - Typically use `andl 0xFF,%eax` to finish job




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

Body

```
movl 12(%ebp),%eax   # eax = y
cmpl %eax,8(%ebp)    # Compare x : eax
setg %al             # al = x > y
andl $255,%eax       # Zero rest of %eax
```

Note inverted ordering!

Jumping

jX Instructions

- Jump to different part of code depending on condition codes

jX	Condition	Description
jmp	1	Unconditional
je	ZF	Equal / Zero
jne	~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)

Conditional Branch Example

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

```
_max:
        pushl %ebp              } Set
        movl %esp,%ebp          } Up

        movl 8(%ebp),%edx       }
        movl 12(%ebp),%eax      }
        cmpl %eax,%edx          } Body
        jle L9                  }
        movl %edx,%eax          }
L9:

        movl %ebp,%esp          }
        popl %ebp               } Finish
        ret                     }
```

Conditional Branch Example (Cont.)

```
int goto_max(int x, int y)
{
  int rval = y;
  int ok = (x <= y);
  if (ok)
    goto done;
  rval = x;
done:
  return rval;
}
```

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

```
    movl 8(%ebp),%edx   # edx = x
    movl 12(%ebp),%eax  # eax = y
    cmpl %eax,%edx      # x : y
    jle L9              # if <= goto L9
    movl %edx,%eax      # eax = x
  L9:                 # Done:
```

Skipped when x ? y

“Do-While” Loop Example

C Code

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

Goto Version

```
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 branch when “while” condition holds

“Do-While” Loop Compilation

Goto Version

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

Registers

```
%edx   x
%eax   result
```

Assembly

```
_fact_goto:
  pushl %ebp           # Setup
  movl %esp,%ebp       # Setup
  movl $1,%eax         # eax = 1
  movl 8(%ebp),%edx    # edx = x

L11:
  imull %edx,%eax      # result *= x
  decl %edx            # x--
  cmpl $1,%edx         # Compare x : 1
  jg L11               # if > goto loop

  movl %ebp,%esp       # Finish
  popl %ebp            # Finish
  ret                  # Finish
```

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:

```
{
  Statement1;
  Statement2;
    …
  Statementn;
}
```

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

  = 0 interpreted as false   ?0 interpreted as true

“While” Loop Example #1

C Code

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

First Goto Version

```
int fact_while_goto
  (int x)
{
  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

```
int fact_while(int x)
{
  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

```
int fact_while_goto2
  (int x)
{
  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 */
int ipwr_while(int x, unsigned p)
{
  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: $x^p = z_0 \cdot z_1{}^2 \cdot (z_2{}^2)^2 \cdot \ldots \cdot (\ldots((z_{n-1}{}^2)^2)\ldots)^2$

  $z_i = 1$ when $p_I = 0$

  $z_i = x$ when $p_I = 1$

  (n times)
- Complexity $O(\log p)$

Example

3^{10}

$= 3^2 * 3^8$

$= 3^2 * ((3^2)^2)^2$

ipwr Computation

```
int ipwr(int x, unsigned p)
{
  int result = 1;
  while (p) {
    if (p & 0x1)
      result *= x;
    x = x*x;
    p = p>>1;
  }
  return result;
}
```

result	x	p
1	3	10
1	9	5
9	81	2
9	6561	1
531441	43046721	0

"While" ? "Do-While " ? "Goto "

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

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

```
  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

```
  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:
```

Registers

```
%ecx   x
%edx   p
%eax   result
```

```
   pushl %ebp               # Setup
   movl %esp,%ebp           # Setup
   movl $1,%eax             # eax = 1
   movl 8(%ebp),%ecx        # ecx = x
   movl 12(%ebp),%edx       # edx = p
   testl %edx,%edx          # Test p
   je L36                   # If 0, goto done
L37:                      # Loop:
   testb $1,%dl             # Test p & 0x1
   je L38                   # If 0, goto skip
   imull %ecx,%eax          # result *= x
L38:                      # Skip:
   imull %ecx,%ecx          # x *= x
   shrl $1,%edx             # p >>= 1
   jne L37                  # if p goto Loop
L36:                      # Done:
   movl %ebp,%esp           # Finish
   popl %ebp                # Finish
   ret                      # Finish
```

"For" Loop Example

```
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;
}
```

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

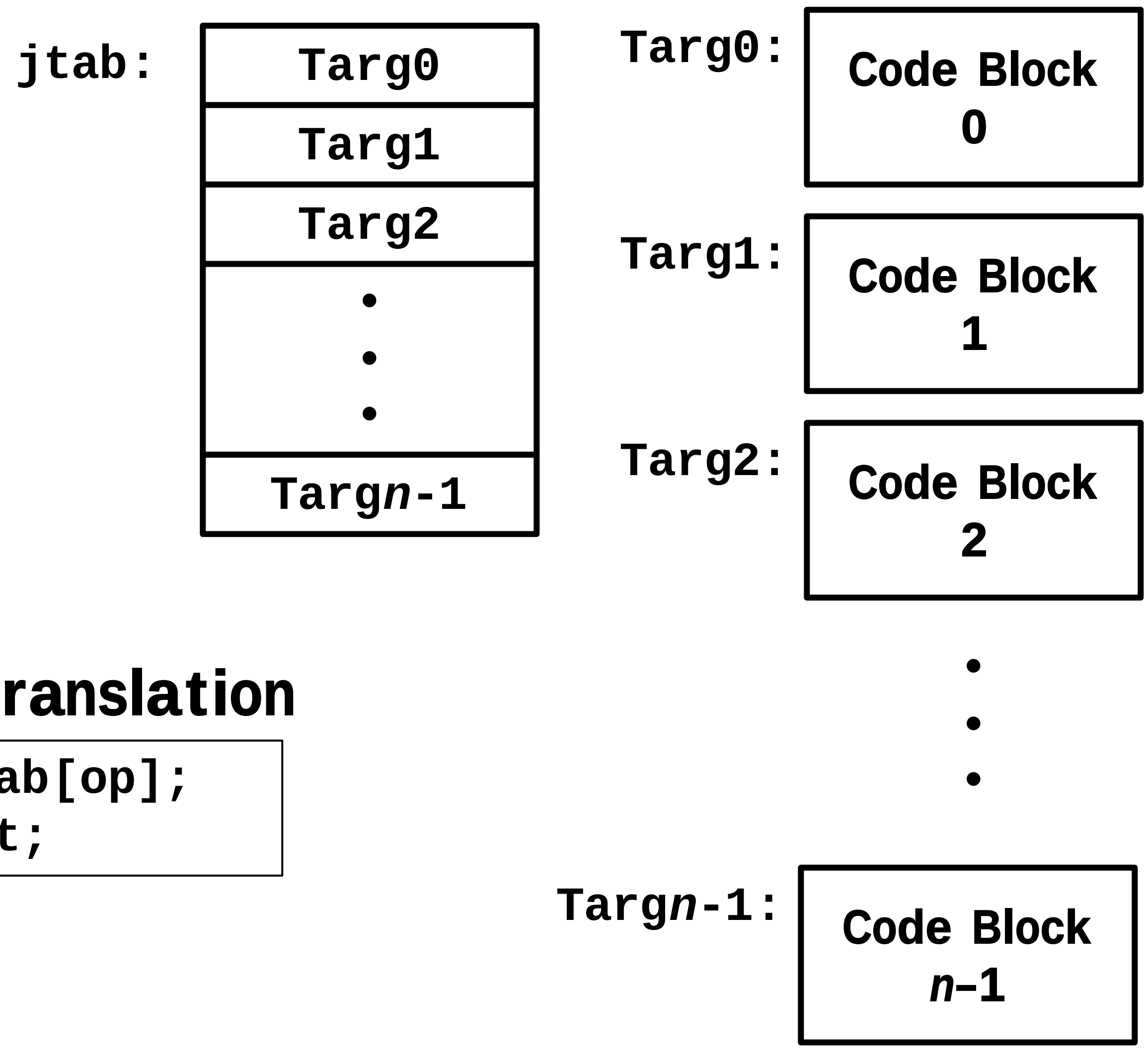
Jump Table Structure

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:

```
pushl %ebp              # Setup
movl %esp,%ebp          # Setup
movl 8(%ebp),%eax       # eax = op
cmpl $5,%eax            # Compare op : 5
ja .L64                 # If > goto done
jmp *.L72(,%eax,4)      # goto Table[op]
```

Assembly Setup Explanation

Symbolic Labels

- **Labels of form `.L`*`XX`* translated into addresses by assembler**

Table Structure

- **Each target requires 4 bytes**
- **Base address at `.L72`**

Jumping

`jmp .L64`

- Jump target is denoted by label **`.L64`**

`jmp *.L72(,%eax,4)`

- Start of jump table denoted by label **`.L72`**
- Register **`%eax`** holds **`op`**
- Must scale by factor of 4 to get offset into table
- Fetch target from effective Address **`.L72 + op*4`**

Jump Table

Table Contents

```
.L72:
  .long .L66 #Op = 0
  .long .L67 #Op = 1
  .long .L68 #Op = 2
  .long .L69 #Op = 3
  .long .L70 #Op = 4
  .long .L71 #Op = 5
```

Enumerated Values

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

Targets & Completion

```
.L66:
    movl $43,%eax # '+'
    jmp .L64
.L67:
    movl $42,%eax # '*'
    jmp .L64
.L68:
    movl $45,%eax # '-'
    jmp .L64
.L69:
    movl $47,%eax # '/'
    jmp .L64
.L70:
    movl $37,%eax # '%'
    jmp .L64
.L71:
    movl $63,%eax # '?'
    # Fall Through to .L64
```

class06.ppt

Switch Statement Completion

```
.L64:               # Done:
    movl %ebp,%esp  # Finish
    popl %ebp       # Finish
    ret             # Finish
```

Puzzle

- What value returned when op is invalid?

Answer

- Register `%eax` set to op at beginning of procedure
- This becomes the returned value

Advantage of Jump Table

- Can do *k*-way branch in *O*(1) operations

Object Code

Setup

- Label `.L64` becomes address `0x80487b5`
- Label `.L72` becomes address `0x8048770`

```
 804875d: 89 e5          movl   %esp,%ebp
 804875f: 8b 45 08       movl   0x8(%ebp),%eax
 8048762: 83 f8 05       cmpl   $0x5,%eax
 8048765: 77 4e          ja     80487b5
<unparse_symbol+0x59>
 8048767: ff 24 85 70 87    jmp    *0x8048770(,%eax,4)
```

Object Code (cont.)

Jump Table

- Disassembler tries to interpret byte sequence as instructions
- Very strange results!

```
804876c: 04 08
 804876e: 89 f6              movl   %esi,%esi
 8048770: 88 87 04 08 90     movb   %al,0x87900804(%edi)
 8048775: 87
 8048776: 04 08              addb   $0x8,%al
 8048778: 98                 cwtl
 8048779: 87 04 08           xchgl  %eax,(%eax,%ecx,1)
 804877c: a0 87 04 08 a8     movb   0xa8080487,%al
 8048781: 87 04 08           xchgl  %eax,(%eax,%ecx,1)
 8048784: b0 87              movb   $0x87,%al
 8048786: 04 08              addb   $0x8,%al
```

Decoding Jump Table

Known

- Starts at 8048770
- 4 bytes / entry
- Little Endian byte ordering

```
804876c: 04 08
804876e: 89 f6
8048770: 88 87 04 08 90
8048775: 87
8048776: 04 08
8048778: 98
8048779: 87 04 08
804877c: a0 87 04 08 a8
8048781: 87 04 08
8048784: b0 87
8048786: 04 08
```

Address	Entry
8048770:	08048788
8048774:	08048790
8048778:	08048798
804877c:	080487a0
8048780:	080487a8
8048784:	080487b0

Alternate Decoding Technique

Use GDB

`gdb code-examples`

`(gdb) x/6xw 0x8048770`

- – E*x*amine 6 he*x*adecimal format "*w*ords" (4-bytes each)
- – Use command "**`help x`**" to get format documentation

```
0x8048770 <unparse_symbol+20>:
  0x08048788
  0x08048790
  0x08048798
  0x080487a0
0x8048780 <unparse_symbol+36>:
  0x080487a8
  0x080487b0
```

Disassembled Targets

- No-operations (nop) inserted to align target addresses

```
8048788: b8 2b 00 00 00   movl   $0x2b,%eax
804878d: eb 26            jmp    80487b5 <unparse_symbol+0x59>
804878f: 90               nop
8048790: b8 2a 00 00 00   movl   $0x2a,%eax
8048795: eb 1e            jmp    80487b5 <unparse_symbol+0x59>
8048797: 90               nop
8048798: b8 2d 00 00 00   movl   $0x2d,%eax
804879d: eb 16            jmp    80487b5 <unparse_symbol+0x59>
804879f: 90               nop
80487a0: b8 2f 00 00 00   movl   $0x2f,%eax
80487a5: eb 0e            jmp    80487b5 <unparse_symbol+0x59>
80487a7: 90               nop
80487a8: b8 25 00 00 00   movl   $0x25,%eax
80487ad: eb 06            jmp    80487b5 <unparse_symbol+0x59>
80487af: 90               nop
80487b0: b8 3f 00 00 00   movl   $0x3f,%eax
```

Matching Disassembled Targets

Address	Entry
8048770:	08048788
8048774:	08048790
8048778:	08048798
804877c:	080487a0
8048780:	080487a8
8048784:	080487b0

```
8048788: b8 2b 00 00 00   movl
804878d: eb 26            jmp
804878f: 90               nop
8048790: b8 2a 00 00 00   movl
8048795: eb 1e            jmp
8048797: 90               nop
8048798: b8 2d 00 00 00   movl
804879d: eb 16            jmp
804879f: 90               nop
80487a0: b8 2f 00 00 00   movl
80487a5: eb 0e            jmp
80487a7: 90               nop
80487a8: b8 25 00 00 00   movl
80487ad: eb 06            jmp
80487af: 90               nop
80487b0: b8 3f 00 00 00   movl
```

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