Machine-Level Programming II: Control Flow

Today
- Condition codes
- Control flow structures

Next time
- Procedures

Processor state (ia32, partial)

- Information about currently executing program
  - Temporary data
  - Location of runtime stack
  - Location of current code control point
  - Status of recent tests

General purpose registers	**%eax**
	%ecx
	%edx
	%ebx
	%esi
	%edi
Current stack top	**%esp**
Current stack frame	**%ebp**
Instruction pointer	**%eip**
Condition codes	**CF** **ZF** **SF** **OF**

Condition codes (Implicit setting)

- Single bit registers

`CF`	Carry Flag	`SF`	Sign Flag
`ZF`	Zero Flag	`OF`	Overflow Flag

- Implicitly set 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 (positive or negative)

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

- Logical operations leave `CF` and `OF` set to 0
- Not set by `leal` instruction (*why?*)

Condition codes (Explicit setting: compare)

- Explicit setting by compare instruction

  `cmp{b,w,l} Src2,Src1`

  `cmpl b,a` like computing `a-b` without setting destination

  - `CF` set if carry 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)`

Condition codes (Explicit setting: test)

- Explicit setting by test instruction

  `test{b,w,l}` *`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 `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
  - Note the suffixes do not indicate operand sizes, but conditions

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

- 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 movzbl to finish job

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

`%eax`	`%ah`	`%al`
`%edx`	`%dh`	`%dl`
`%ecx`	`%ch`	`%cl`
`%ebx`	`%bh`	`%bl`

Body

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

Note inverted ordering!

Breaking with sequential execution

- “Normal” execution executes instructions in listed order
- To move to a different location – jump
  - Jump to different part of code depending on condition codes
  - Destination of a jump – label
  - Label addresses are determined when generating the object code

```
        movl 8(%ebp),%edx
        movl 12(%ebp),%eax
        cmpl %eax,%edx
        jle .L9
        movl %edx,%eax
.L9:
        leave
        ret
```

Jump if <=a

Label for the jump

Jumping

- jX Instructions
  - `jmp` can be direct (to a label) or indirect (e.g. *(%eax), to M[R[%eax]]); everything else only direct

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

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

```
max:
        pushl %ebp
        movl %esp,%ebp

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

        leave
        ret
```

Set Up

Body

Finish

Conditional branch example

```
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

General "if-else" translation

C Code

```
If (test-expr)
    then-statement
else
    else-statement
```

Goto Version

```
   t = test-expr;
   if (!t)
      goto false;
   then-statement
   goto done;
false:
   else-statement;
```

- *test-expr* is expression returning integer
  - = 0 interpreted as false
  - ≠0 interpreted as true
- Only one of the two branches is executed
- There are other models/goto versions

Conditional expression

C Code

```
val = test? then_expr : else_expr;
```

- Test is an expression returning an int
  - 0 is false; !0 is true

Goto Version

```
  nt = !test;
  if (nt) goto else;
  val = then_expr;
  goto done;
else:
  val = else_expr;
done:
  …
```

```
int absdiff(int x, int y)
{
   return x<y? y-x:x-y;
}
```

```
   movl     8(%ebp), %eax
   movl     12(%ebp), %edx
   cmpl     %edx, %eax
   jge      .L2
   subl     %eax, %edx
   movl     %edx, %eax
   jmp      .L3
.L2:
   subl     %edx, %eax
.L3:
   …
```

Loops

- C provides different looping constructs
  - `do-while`, `while`, `for`
- No corresponding instruction in machine code
- Most compilers
  - Transform general loops into `do-while`
  - Then compile them into machine code

“Do-While” loop example

- Compute factorial of n (*n!*)

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;
}
```

Assembly

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

.L2:
  imull %edx,%eax     # result *= x
  decl %edx           # x--
  cmpl $1,%edx        # Compare x : 1
  jg .L2              # if > goto loop

  leave               # Finish
  ret                 # Finish
```

- Registers

  `%edx`   `x`

  `%eax`   `result`

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;
}
```

- Historically used by GCC
- 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:
```

“For” loop example

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

for (*Init*; *Test*; *Update*)

Body

Init

```
i = 2;
```

Test

```
i <= n;
```

Update

```
i++;
```

Body

```
{
     result *= i;
}
```

"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:
```

```
{
    int result = 1;
    i = 2;
    if (!(i <=n))
      goto done;
  loop:
    result *= i;
    i++;
    if (i <= n)
      goto loop;
  done:
    return result;
}
```

Init

```
i = 2;
```

Test

```
i <= n;
```

Update

```
i++;
```

Body

```
{
    result *= i;
}
```

"For" loop compilation result

```
{
    int result = 1;
    i = 2;
    if (!(i <=n))
      goto done;
  loop:
    result *= i;
    i++;
    if (i <= n)
      goto loop;
  done:
    return result;
}
```

Assembly

```
fact_for:
  pushl %ebp
  movl  %esp, %ebp
  movl  8(%ebp), %ecx     Get n
  movl  $2, %edx          i <- 2
  movl  $1, %eax          result <- 1
  cmpl  $1, %ecx          Comp n:1
  jle   .L3               If <=, goto done
.L6:                      loop
  imull %ecx, %eax        result *= i
  addl  $1, %edx          i++
  cmpl  %edx, %ecx        Comp n:i
  jge   .L6               If >=, goto loop
.L3:                      done
  popl  %ebp
  ret
```

Conditional moves

- Conditional move instructions
  - Instruction supports if (test) dest ← src
  - Supported starting with PentiumPro, since 1995
  - GCC tends not to use it for backward compatibility; it's starting to change (use switch `-march=i686` -i.e. PentiumPro)
- Why?
  - Branches disrupt the instruction flows of pipelines
  - Conditional moves avoid control transfers

C Code

```
val = test
  ? then_expr
  : else_expr;
```

Goto Version

```
tval = then_expr;
result = else_expr;
t = test;
if (t) result = tval;
return result;
```

Conditional moves example

```
int absdiff(int x, int y)
{
  int result;
  if (x<y) {
    result = y-x;
  } else {
    result = x-y;
  }
  return result;
}
```

```
x in %ecx    y in %edx

movl    %edx, %eax
subl    %ecx, %eax   tval=y-x
movl    %ecx, %ebx
subl    %edx, %ebx   result=x-y
cmpl    %edx, %ecx   Comp x:y
cmovge %ebx, %eax   if >, result=tval
…
ret
```

- When using conditional moves maybe a bad idea
  - Expensive computation; one of the sides will be discarded
  - Computation has side effects (must be side-effect free!)

    `val = x>0? x*=7 : x+=3;`

Switch statements

```
long switch_eg (long x, long y,
long z)
{
   long w = 1;
   switch(x) {
   case 1:
      w = y * x;
      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;
}
```

- A multi-way branching capability based on the value of an `int`
- Useful when many possible outcomes
- Switch cases
  - Multiple case labels
    - Here 5 & 6;
  - Fall through cases:
    - Here 2
  - Missing cases:
    - Here 4
- Easier to read C code and more efficient implementation with jump tables

Jump table structure

- An array where entry *i* is the address of a code segment to run when switch index equals *i*

Switch form

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

Jump table

jtab:

Targ0
Targ1
Targ2
• • •
Targn-1

Jump targets

Targ0: **Code Block 0**

Targ1: **Code Block 1**

Targ2: **Code Block 2**

• • •

Targn-1: **Code Block *n*–1**

Approx. translation

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

Switch statement example

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

Setup:

```
switch_eg:
   pushl %ebp
   movl %esp,%ebp
   pushl %ebx
   movl $1, %ebx
   movl 8(%ebp),%edx
   movl 16(%ebp), %ecx
   cmpl $6, %edx
   ja .L8
   jmp *.L9(,%edx,4)
```

Switch statement example

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

Jump table

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

Setup:

Indirect jump

```
switch_eg:
   pushl %ebp                     # Setup
   movl %esp,%ebp                 # Setup
   pushl %ebx                     # Setup
   movl $1, %ebx                  # w = 1
   movl 8(%ebp),%edx              # edx = x
   movl 16(%ebp), %ecx            # ecx = z
   cmpl $6, %edx                  # Compare x-6 to 0
   ja .L8                         # if > goto default
   jmp *.L9(,%edx,4)              # goto JTab[x]
```

Assembly setup explanation

- Table structure
  - Each target requires 4 bytes
  - Base address at `.L9`

- Jumping

  `jmp .L8`
  - **Jump target is denoted by label** `.L9`

  `jmp *.L9(,%edx,4)`
  - **Start of jump table denoted by label** `.L9`
  - **Register** `%edx` **holds** `op`
  - **Must scale by factor of 4 to get offset into table (labels have 32-bits, 4 bytes on IA32)**
  - **Fetch target from effective Address** `.L9 + op*4`

Jump table

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

Jump table

Jump table

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

```
switch(x) {
   case 1:       /* .L3 */
      w = y * x;
      break;
   case 2:       /* .L4 */
      w = y/z;
      /* fall through */
   case 3:       /* .L5 */
      w += z;
      break;
   case 5:
   case 6:       /* .L7 */
      w -= z;
      break;
   default:      /* .L8 */
      w = 2;
   }
```

Code block

```
switch(x) {
   case 1:         /* .L3 */
      w = y * x;
      break;
   case 2:         /* .L4 */
      w = y/z;
      /* fall through */
   case 3:         /* .L5 */
      w += z;
      break;
   case 5:
   case 6:         /* .L7 */
      w -= z;
      break;
   default:        /* .L8 */
      w = 2;
   }
```

```
.L3:
      movl     12(%ebp), %ebx
      imull    %edx, %ebx
      jmp      .L2
.L4:
      movl     12(%ebp), %eax
      cltd
      idivl    %ecx
      movl     %eax, %ebx
.L5:
      addl     %ecx, %ebx
      jmp      .L2
.L7:
      subl     %ecx, %ebx
      jmp      .L2
.L8:
      movl     $2, %ebx
```

- `imull` S R[%edx]:R[%eax] ← S * R[%eax]
- `cltd` R[%edx]:R[%eax] ←SignExtend(R[%eax])
  Convert to quad word for division
- `idivl` S R[%edx] ← R[%edx]:R[%eax] mod S
  R[%eax] ← R[%edx]:R[%eax] / S

Object code

- Setup
  - Label `.L8` becomes address `0x8048367`
  - Label `.L9` becomes address `0x8048470`

```
08048334 <switch_eg>:
8048334:       55                      push   %ebp
8048335:       89 e5                   mov    %esp,%ebp
8048337:       53                      push   %ebx
8048338:       8b 55 08                mov    0x8(%ebp),%edx
804833b:       8b 4d 10                mov    0x10(%ebp),%ecx
804833e:       bb 01 00 00 00          mov    $0x1,%ebx
8048343:       83 fa 06                cmp    $0x6,%edx
8048346:       77 1f                   ja     8048367 <switch_eg+0x33>
8048348:       ff 24 95 70 84 04 08    jmp    *0x8048470(,%edx,4)
```

Object code

- Jump table
  - Doesn't show up in disassembled code
  - Can inspect using GDB

```
gdb code-examples
(gdb) x/7xw 0x8048470
```

    - Examine 7 hexadecimal format "words" (4-bytes each)
    - Use command "`help x`" to get format documentation

```
0x8048470:
    0x08048367
    0x0804834f
    0x08048357
    0x0804835f
    0x08048367
    0x08048363
    0x08048363
```

Disassembled targets

```
0x08048367
0x0804834f
0x08048357
0x0804835f
0x08048367
0x08048363
0x08048363
```

```
804834f:   8b 5d 0c          mov    0xc(%ebp),%ebx
8048352:   0f af da          imul   %edx,%ebx
8048355:   eb 15             jmp    804836c <switch_eg+0x38>
8048357:   8b 45 0c          mov    0xc(%ebp),%eax
804835a:   99                cltd
804835b:   f7 f9             idiv   %ecx
804835d:   89 c3             mov    %eax,%ebx
804835f:   01 cb             add    %ecx,%ebx
8048361:   eb 09             jmp    804836c <switch_eg+0x38>
8048363:   29 cb             sub    %ecx,%ebx
8048365:   eb 05             jmp    804836c <switch_eg+0x38>
8048367:   bb 02 00 00 00    mov    $0x2,%ebx
804836c:   89 d8             mov    %ebx,%eax
804836e:   5b                pop    %ebx
804836f:   c9                leave
8048370:   c3                ret
```

Sparse switch example

- Not practical to use jump table
  - Would require 1000 entries
- Obvious translation into if-then-else would have max. of 9 tests

```
/* 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;
  }
}
```

Sparse switch code

- Compares x to possible case values
- Jumps different places depending on outcomes

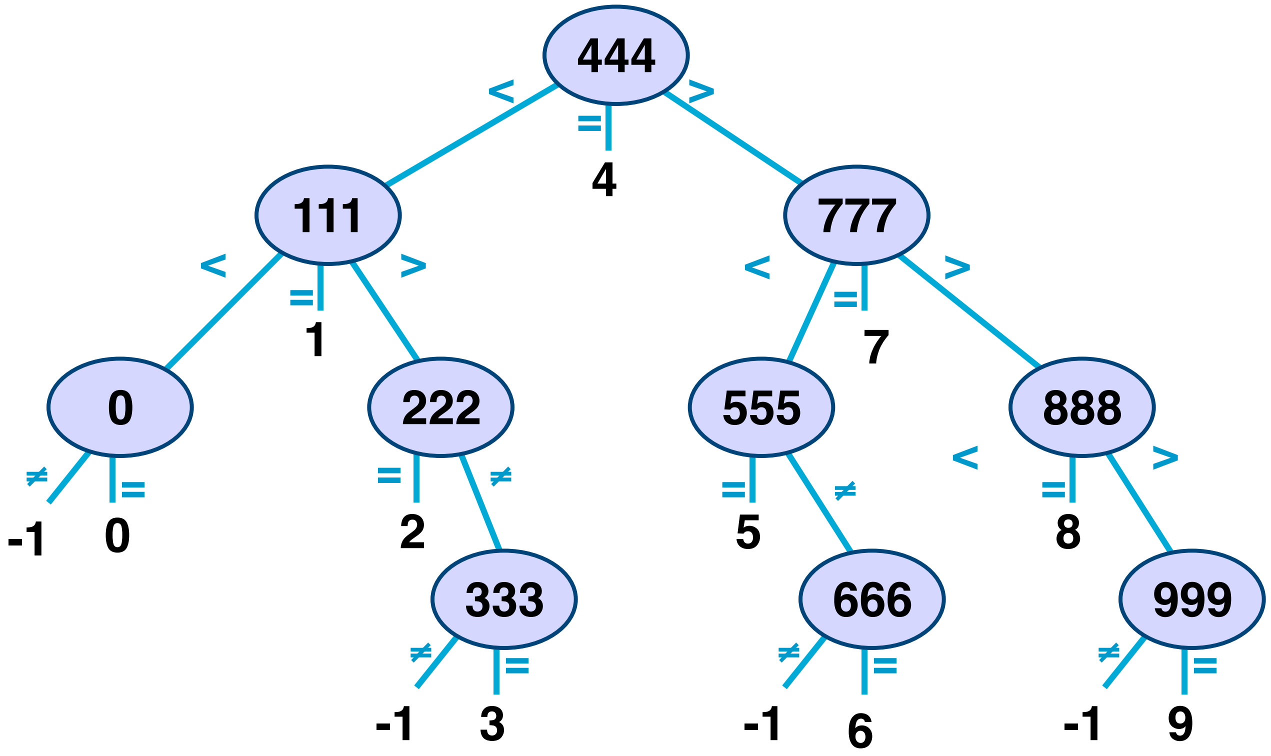
```
movl 8(%ebp),%eax  # get x
cmpl $444,%eax     # x:444
je L8
jg L16
cmpl $111,%eax     # x:111
je L5
jg L17
testl %eax,%eax    # x:0
je L4
jmp L14

. . .
```

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

Sparse switch code structure

- Organizes cases as binary tree
- Logarithmic performance



Summarizing

- C Control
  - if-then-else, do-while, while, switch
  - Conditional transfer of data
- Assembler control
  - Jump & conditional jump
- Compiler
  - Must generate assembly code to implement more complex control
- Standard techniques
  - All loops → do-while form
  - Large switch statements use jump tables