Machine Programming II: Instructions (cont'd)

- Move instructions, registers, and operands
- Complete addressing mode, address computation (`leal`)
- Arithmetic operations (including some x86-64 instructions)
- Condition codes
- Control, unconditional and conditional branches
- **While loops**
- **For loops**
- **Switch statements**

"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;
}
```

Assembly

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

.L11:
  imull %edx,%eax
  decl %edx
  cmpl $1,%edx
  jg .L11

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

Registers:

%edx	x
%eax	result

Translation?

"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

.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
```

Registers:

%edx	x
%eax	result

General "Do-While" Translation

C Code

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

Goto Version

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

- ***Body:***

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

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

"While" Loop Example

C Code

```
int fact_while(int x)
{
  int result = 1;
  while (x > 1) {


    result *= x;
    x = x-1;
  };

  return result;
}
```

Goto Version #1

```
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**

Alternative "While" Loop Translation

C Code

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

Goto Version #2

```
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;
}
```

- Historically used by GCC
- Uses same inner loop as do-while version
- Guards loop entry with extra test

General "While" Translation

While version

```
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:
```

New Style "While" Loop Translation

C Code

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

Goto Version

```
int fact_while_goto3(int x)
{
  int result = 1;
  goto middle;
loop:
  result *= x;
  x = x-1;
middle:
  if (x > 1)
    goto loop;
  return result;
}
```

- Recent technique for GCC
  - Both IA32 & x86-64
- First iteration jumps over body computation within loop

Jump-to-Middle While Translation

C Code

```
while (Test)
  Body
```


Goto Version

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

- Avoids duplicating test code
- Unconditional `goto` incurs no performance penalty
- `for` loops compiled in similar fashion

Goto (Previous) Version

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

Jump-to-Middle Example

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

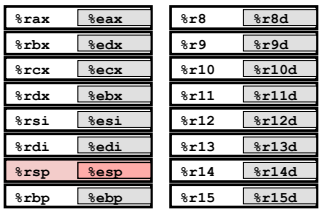
```
# x in %edx, result in %eax
  jmp   .L34        #   goto Middle
.L35:               # Loop:
  imull %edx, %eax  #   result *= x
  decl  %edx        #   x--
.L34:               # Middle:
  cmpl  $1, %edx    #   x:1
  jg    .L35        #   if >, goto Loop
```


Quick Review

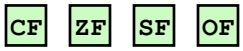
- Complete memory addressing mode
  - `(%eax), 17(%eax), 2(%ebx, %ecx, 8), …`
- Arithmetic operations (set condition codes)
  - `subl %eax, %ecx` `# ecx = ecx + eax`
  - `sall $4,%edx` `# edx = edx << 4`
  - `addl 16(%ebp),%ecx` `# ecx = ecx + Mem[16+ebp]`
  - `imull %ecx,%eax` `# eax = eax * ecx`
- Arithmetic operations (do NOT set condition codes)
  - `leal 4(%edx,%eax),%eax` `# eax = 4 + edx + eax`

Quick Review

- **x86-64 vs. IA32**
  - Integer registers: **16 x 64-bit** vs. **8 x 32-bit**
  - **`movq, addq,`** … vs. **`movl, addl`**, …
  - Better support for passing function arguments in registers



- **Control**
  - Condition code registers
  - Set as side effect or by **`cmp, test`**
  - Used:
    - Read out by setx instructions (**`setg, setle`**, …)
    - Or by conditional jumps (**`jle .L4, je .L10,`** …)



Quick Review

- **Do-While loop**

C Code

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

Goto Version

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

- **While-Do loop**

While version

```
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:
```

or

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

"For" Loop Example: Square-and-Multiply

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

- **Algorithm**
  - Exploit bit representation: $p = p_0 + 2p_1 + 2^2p_2 + \ldots 2^{n-1}p_{n-1}$
  - Gives: $x^p = z_0 \cdot z_1^{\,2} \cdot (z_2^{\,2})^2 \cdot \ldots \cdot \underbrace{(\ldots((z_{n-1}^{\,2})^2)\ldots)^2}_{n-1 \text{ times}}$
    $z_i = 1$ when $p_i = 0$
    $z_i = x$ when $p_i = 1$
  - Complexity $O(\log p)$

Example

$3^{10} = 3^2 * 3^8$

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

`ipwr` Computation

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

before iteration	result	x=3	p=10
1	1	3	$10=1010_2$
2	1	9	$5=\ 101_2$
3	9	81	$2=\ \ 10_2$
4	9	6561	$1=\ \ \ 1_2$
5	59049	43046721	0

"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
```

Test	*Init*	*Update*	*Body*
`p != 0`	`result = 1`	`p = p >> 1`	`{ if (p & 0x1) result *= x; x = x*x; }`

"For" → "While" → "Do-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 #1

For Version

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

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

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:
```


"For" → "While" (Jump-to-Middle)

For Version

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

While Version

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

Goto Version

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

For-Loop: Compilation #2

For Version

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

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

Goto Version

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

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



```
long switch_eg
   (long x, long y, long z)
{
    long 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;
}
```

Switch Statement Example

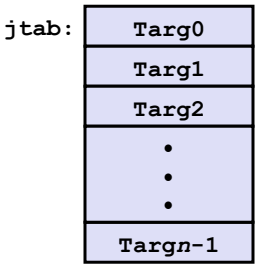
- **Multiple case labels**
  - Here: 5, 6
- **Fall through cases**
  - Here: 2
- **Missing cases**
  - Here: 4

Jump Table Structure

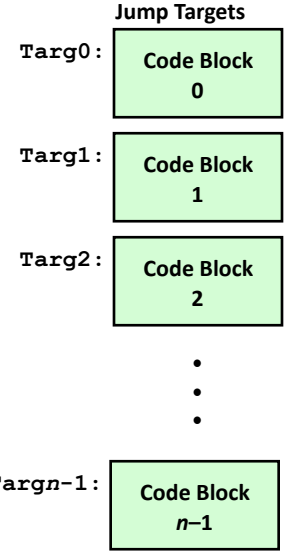
Switch Form

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

Jump Table



Jump Targets



Approximate Translation

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


Switch Statement Example (IA32)

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

Setup:

```
switch_eg:
   pushl %ebp            # Setup
   movl  %esp, %ebp      # Setup
   pushl %ebx            # Setup
   movl  $1, %ebx
   movl  8(%ebp), %edx
   movl  16(%ebp), %ecx
   cmpl  $6, %edx
   ja    .L61
   jmp   *.L62(,%edx,4)
```

Will disappear Blackboard?

Switch Statement Example (IA32)

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

Jump table

```
.section .rodata
   .align 4
.L62:
  .long   .L61  # x = 0
  .long   .L56  # x = 1
  .long   .L57  # x = 2
  .long   .L58  # x = 3
  .long   .L61  # x = 4
  .long   .L60  # x = 5
  .long   .L60  # x = 6
```

Setup:

```
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        # x:6
   ja    .L61            # if > goto default
   jmp   *.L62(,%edx,4)  # goto JTab[x]
```

Indirect jump 

Assembly Setup Explanation

- **Table Structure**
  - Each target requires 4 bytes
  - Base address at `.L62`

Jump table

```
.section .rodata
   .align 4
.L62:
  .long   .L61  # x = 0
  .long   .L56  # x = 1
  .long   .L57  # x = 2
  .long   .L58  # x = 3
  .long   .L61  # x = 4
  .long   .L60  # x = 5
  .long   .L60  # x = 6
```

- **Jumping**

  Direct: `jmp .L61`
  - Jump target is denoted by label `.L61`

  Indirect: `jmp *.L62(,%edx,4)`
  - Start of jump table: `.L62`
  - Must scale by factor of 4 (labels have 32-bit = 4 Bytes on IA32)
  - Fetch target from effective Address `.L61 + edx*4`
    - Only for $0 \le x \le 6$

Jump Table

Jump table

```
.section .rodata
   .align 4
.L62:
  .long   .L61  # x = 0
  .long   .L56  # x = 1
  .long   .L57  # x = 2
  .long   .L58  # x = 3
  .long   .L61  # x = 4
  .long   .L60  # x = 5
  .long   .L60  # x = 6
```

```
    switch(x) {
    case 1:       // .L56
        w = y*z;
        break;
    case 2:       // .L57
        w = y/z;
        /* Fall Through */
    case 3:       // .L58
        w += z;
        break;
    case 5:
    case 6:       // .L60
        w -= z;
        break;
    default:      // .L61
        w = 2;
    }
```


Code Blocks (Partial)

```
  switch(x) {
    . . .
  case 2:       // .L57
      w = y/z;
      /* Fall Through */
  case 3:       // .L58
      w += z;
      break;
    . . .
  default:      // .L61
      w = 2;
  }
```

```
.L61:  // Default case
   movl  $2, %ebx     # w = 2
   movl  %ebx, %eax   # Return w
   popl  %ebx
   leave
   ret
.L57:  // Case 2:
   movl  12(%ebp), %eax  # y
   cltd                  # Div prep
   idivl %ecx            # y/z
   movl  %eax, %ebx # w = y/z
# Fall through
.L58:  // Case 3:
   addl  %ecx, %ebx # w+= z
   movl  %ebx, %eax # Return w
   popl  %ebx
   leave
   ret
```

Code Blocks (Rest)

```
    switch(x) {
    case 1:       // .L56
        w = y*z;
        break;
      . . .
    case 5:
    case 6:       // .L60
        w -= z;
        break;
      . . .
    }
```

```
.L60: // Cases 5&6:
   subl  %ecx, %ebx  # w -= z
   movl  %ebx, %eax  # Return w
   popl  %ebx
   leave
   ret
.L56: // Case 1:
   movl  12(%ebp), %ebx # w = y
   imull %ecx, %ebx     # w*= z
   movl  %ebx, %eax  # Return w
   popl  %ebx
   leave
   ret
```


IA32 Object Code

- **Setup**
  - Label **.L61** becomes address **0x8048630**
  - Label **.L62** becomes address **0x80488dc**

Assembly Code

```
switch_eg:
  . . .
   ja    .L61            # if > goto default
   jmp   *.L62(,%edx,4)  # goto JTab[x]
```

Disassembled Object Code

```
08048610 <switch_eg>:
 . . .
 8048622:  77 0c                    ja     8048630
 8048624:  ff 24 95 dc 88 04 08     jmp    *0x80488dc(,%edx,4)
```

IA32 Object Code (cont.)

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

```
gdb asm-cntl
(gdb) x/7xw 0x80488dc
```

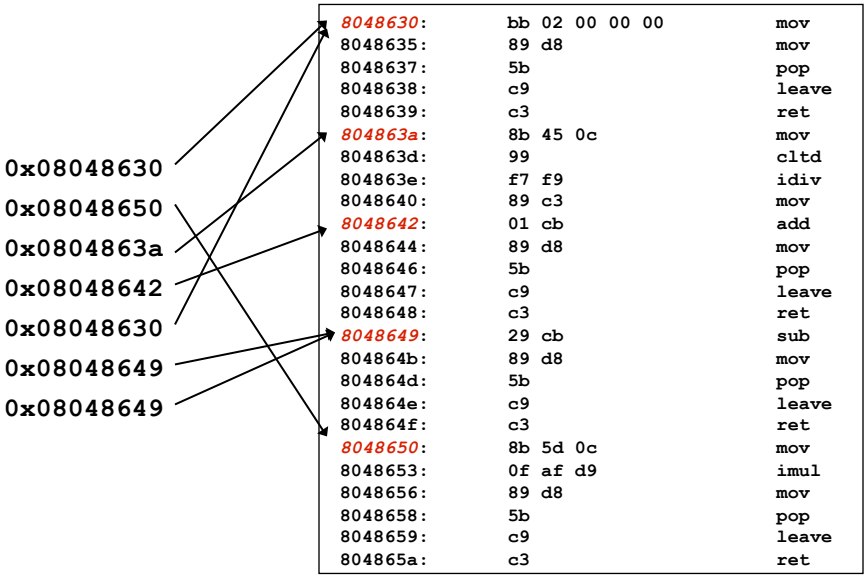
    - Examine 7 hexadecimal format "words" (4-bytes each)
    - Use command "**help x**" to get format documentation

```
0x80488dc:
  0x08048630
  0x08048650
  0x0804863a
  0x08048642
  0x08048630
  0x08048649
  0x08048649
```


Disassembled Targets

```
8048630:    bb 02 00 00 00          mov    $0x2,%ebx
8048635:    89 d8                   mov    %ebx,%eax
8048637:    5b                      pop    %ebx
8048638:    c9                      leave
8048639:    c3                      ret
804863a:    8b 45 0c                mov    0xc(%ebp),%eax
804863d:    99                      cltd
804863e:    f7 f9                   idiv   %ecx
8048640:    89 c3                   mov    %eax,%ebx
8048642:    01 cb                   add    %ecx,%ebx
8048644:    89 d8                   mov    %ebx,%eax
8048646:    5b                      pop    %ebx
8048647:    c9                      leave
8048648:    c3                      ret
8048649:    29 cb                   sub    %ecx,%ebx
804864b:    89 d8                   mov    %ebx,%eax
804864d:    5b                      pop    %ebx
804864e:    c9                      leave
804864f:    c3                      ret
8048650:    8b 5d 0c                mov    0xc(%ebp),%ebx
8048653:    0f af d9                imul   %ecx,%ebx
8048656:    89 d8                   mov    %ebx,%eax
8048658:    5b                      pop    %ebx
8048659:    c9                      leave
804865a:    c3                      ret
```

Matching Disassembled Targets



Summarizing

- **C Control**
  - if-then-else
  - do-while
  - while, for
  - switch
- **Assembler Control**
  - Conditional jump
  - Conditional move
  - Indirect jump
  - Compiler
  - Must generate assembly code to implement more complex control
- **Standard Techniques**
  - Loops converted to do-while form
  - Large switch statements use jump tables
  - Sparse switch statements may use decision trees (see text)
- **Conditions in CISC**
  - CISC machines generally have condition code registers