Computer Systems I

Class 6

Today

- Complete addressing mode, address computation (`leal`)
- Arithmetic operations
- x86-64
- Control: Condition codes
- Conditional branches
- While loops

Data Representations: IA32 + x86-64

- Sizes of C Objects (in Bytes)

C Data Type	*Typical 32-bit*	*Intel IA32*	*x86-64*
unsigned	4	4	4
int	4	4	4
long int	4	4	8
char	1	1	1
short	2	2	2
float	4	4	4
double	8	8	8
long double	8	10/12	16
char *	4	4	8

Or any other pointer

x86-64 Integer Registers

%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

- Extend existing registers. Add 8 new ones.
- Make **`%ebp`/`%rbp`** general purpose

Instructions

- **Long word `l` (4 Bytes) ↔ Quad word `q` (8 Bytes)**
- **New instructions:**
  - **`movl`** → **`movq`**
  - **`addl`** → **`addq`**
  - **`sall`** → **`salq`**
  - etc.
- **32-bit instructions that generate 32-bit results**
  - Set higher order bits of destination register to 0
  - Example: **`addl`**

Swap in 32-bit Mode

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
    pushl %ebp
    movl  %esp,%ebp
    pushl %ebx

    movl 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax,(%edx)
    movl %ebx,(%ecx)

    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```

Setup

Body

Finish

Swap in 64-bit Mode

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
    movl    (%rdi), %edx
    movl    (%rsi), %eax
    movl    %eax, (%rdi)
    movl    %edx, (%rsi)
    retq
```

- **Operands passed in registers (why useful?)**
  - First (**xp**) in **`%rdi`**, second (**yp**) in **`%rsi`**
  - 64-bit pointers
- **No stack operations required**
- **32-bit data**
  - Data held in registers **`%eax`** and **`%edx`**
  - **`movl`** operation

Swap Long Ints in 64-bit Mode

```
void swap_l
  (long int *xp, long int *yp)
{
  long int t0 = *xp;
  long int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap_l:
    movq    (%rdi), %rdx
    movq    (%rsi), %rax
    movq    %rax, (%rdi)
    movq    %rdx, (%rsi)
    retq
```

- **64-bit data**
  - Data held in registers **`%rax`** and **`%rdx`**
  - **`movq`** operation
  - "q" stands for quad-word

Today

- Complete addressing mode, address computation (`leal`)
- Arithmetic operations
- x86-64
- **Control: Condition codes**
- Conditional branches
- While loops

Processor State (IA32, Partial)

- **Information about currently executing program**
  - Temporary data (**%eax**, …)
  - Location of runtime stack (**%ebp,%esp**)
  - Location of current code control point (**%eip**, …)
  - Status of recent tests (**CF,ZF,SF,OF**)

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

Condition Codes (Implicit Setting)

- **Single bit registers**

  CF Carry Flag (for unsigned) **SF** Sign Flag (for signed)
  ZF Zero Flag **OF** Overflow Flag (for signed)

- **Implicitly set (think of it as side effect) by arithmetic operations**

  Example: **addl/addq** *Src,Dest* ↔ **t = a+b**
  - CF set if carry out from most significant bit (unsigned overflow)
  - ZF set if **t == 0**
  - SF set if **t < 0** (as signed)
  - OF set if two's complement (signed) overflow
    (a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)

- ***Not*** **set by lea instruction**
- **Full documentation (IA32), link also on course website**

Condition Codes (Explicit Setting: Compare)

- **Explicit Setting by Compare Instruction**

  cmpl/cmpq *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** (as signed)
  - OF set if two's complement (signed) 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**
  `testl/testq` *Src2,Src1*
  `testl b,a` like computing `a&b` without setting destination
  - Sets condition codes based on value of *Src1* & *Src2*
  - Useful to have one of the operands be a mask
  - 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 combination of condition codes
- **One of 8 addressable byte registers**
  - Does not alter remaining 3 bytes
  - Typically use `movzbl` to finish job

%eax	%ah	%al
%ecx	%ch	%cl
%edx	%dh	%dl
%ebx	%bh	%bl
%esi		
%edi		
%esp		
%ebp		

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

Body

```
movl 12(%ebp),%eax
cmpl %eax,8(%ebp)
setg %al
movzbl %al,%eax
```

Reading Condition Codes: x86-64

- **SetX Instructions:**
  - Set single byte based on combination of condition codes
  - Does not alter remaining 3 bytes

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

```
long lgt (long x, long y)
{
  return x > y;
}
```

Body (same for both)

```
xorl %eax, %eax
cmpq %rsi, %rdi
setg %al
```

Is `%rax` zero?
Yes: 32-bit instructions set high order 32 bits to 0!

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)

Today

- Complete addressing mode, address computation (`leal`)
- Arithmetic operations
- x86-64
- Control: Condition codes
- **Conditional branches**
- While loops

Conditional Branch Example

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

```
absdiff:
    pushl  %ebp              } Setup
    movl   %esp, %ebp
    movl   8(%ebp), %edx
    movl   12(%ebp), %eax
    cmpl   %eax, %edx
    jle    .L7                } Body1
    subl   %eax, %edx
    movl   %edx, %eax
.L8:
    leave                     } Finish
    ret
.L7:
    subl   %edx, %eax         } Body2
    jmp    .L8
```


Conditional Branch Example (Cont.)

```
int goto_ad(int x, int y)
{
  int result;
  if (x <= y) goto Else;
  result = x-y;
Exit:
  return result;
Else:
  result = y-x;
  goto Exit;
}
```

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

```
absdiff:
    pushl  %ebp
    movl   %esp, %ebp
    movl   8(%ebp), %edx
    movl   12(%ebp), %eax
    cmpl   %eax, %edx
    jle    .L7
    subl   %eax, %edx
    movl   %edx, %eax
.L8:
    leave
    ret
.L7:
    subl   %edx, %eax
    jmp    .L8
```


Conditional Branch Example (Cont.)

```
int goto_ad(int x, int y)
{
  int result;
  if (x <= y) goto Else;
  result = x-y;
Exit:
  return result;
Else:
  result = y-x;
  goto Exit;
}
```

```
absdiff:
    pushl  %ebp
    movl   %esp, %ebp
    movl   8(%ebp), %edx
    movl   12(%ebp), %eax
    cmpl   %eax, %edx
    jle    .L7
    subl   %eax, %edx
    movl   %edx, %eax
.L8:
    leave
    ret
.L7:
    subl   %edx, %eax
    jmp    .L8
```

Conditional Branch Example (Cont.)

```
int goto_ad(int x, int y)
{
  int result;
  if (x <= y) goto Else;
  result = x-y;
Exit:
  return result;
Else:
  result = y-x;
  goto Exit;
}
```

```
absdiff:
    pushl  %ebp
    movl   %esp, %ebp
    movl   8(%ebp), %edx
    movl   12(%ebp), %eax
    cmpl   %eax, %edx
    jle    .L7
    subl   %eax, %edx
    movl   %edx, %eax
.L8:
    leave
    ret
.L7:
    subl   %edx, %eax
    jmp    .L8
```

Systems I

Conditional Branch Example (Cont.)

```
int goto_ad(int x, int y)
{
  int result;
  if (x <= y) goto Else;
  result = x-y;
Exit:
  return result;
Else:
  result = y-x;
  goto Exit;
}
```

```
absdiff:
    pushl  %ebp
    movl   %esp, %ebp
    movl   8(%ebp), %edx
    movl   12(%ebp), %eax
    cmpl   %eax, %edx
    jle    .L7
    subl   %eax, %edx
    movl   %edx, %eax
.L8:
    leave
    ret
.L7:
    subl   %edx, %eax
    jmp    .L8
```

Systems I

Conditional Branch Example (Cont.)

```
int goto_ad(int x, int y)
{
  int result;
  if (x <= y) goto Else;
  result = x-y;
Exit:
  return result;
Else:
  result = y-x;
  goto Exit;
}
```

```
absdiff:
    pushl  %ebp
    movl   %esp, %ebp
    movl   8(%ebp), %edx
    movl   12(%ebp), %eax
    cmpl   %eax, %edx
    jle    .L7
    subl   %eax, %edx
    movl   %edx, %eax
.L8:
    leave
    ret
.L7:
    subl   %edx, %eax
    jmp    .L8
```

General Conditional Expression Translation

C Code

```
val = Test ? Then-Expr : Else-Expr;
```

```
val = x>y ? x-y : y-x;
```

Goto Version

```
  nt = !Test;
  if (nt) goto Else;
  val = Then-Expr;
Done:
  . . .
Else:
  val = Else-Expr;
  goto Done;
```

- *Test* is expression returning integer
  - = 0 interpreted as false
  - ≠0 interpreted as true
- Create separate code regions for then & else expressions
- Execute appropriate one

Conditionals: x86-64

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

```
absdiff: # x in %edi, y in %esi
  movl   %edi, %eax
  movl   %esi, %edx
  subl   %esi, %eax
  subl   %edi, %edx
  cmpl   %esi, %edi
  cmovle %edx, %eax
  ret
```

- **Conditional move instruction**
  - **cmov*C*** src, dest
  - Move value from src to dest if condition ***C*** holds
  - More efficient than conditional branching (simple control flow)
  - But overhead: both branches are evaluated

General Form with Conditional Move

C Code

```
val = Test ? Then-Expr : Else-Expr;
```

Conditional Move Version

```
val1  = Then-Expr;
val2  = Else-Expr;
val1  = val2 if !Test;
```

- **Both values get computed**
- **Overwrite then-value with else-value if condition doesn't hold**
- **Don't use when:**
  - Then or else expression have side effects
  - Then and else expression are too expensive

Today

- Complete addressing mode, address computation (`leal`)
- Arithmetic operations
- x86-64
- Control: Condition codes
- Conditional branches
- While loops

“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

Registers:

%edx	x
%eax	result

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
```

General "Do-While" Translation

C Code

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

Goto Version

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

- ***Body:*** {
  Statement$_1$;
  Statement$_2$;
  …
  Statement$_n$;
  }
- ***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
```

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

Goto Version

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

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
```


Implementing Loops

- **IA32**
  - All loops translated into form based on "do-while"
- **x86-64**
  - Also make use of "jump to middle"
- **Why the difference**
  - IA32 compiler developed for machine where all operations costly
  - x86-64 compiler developed for machine where unconditional branches incur (almost) no overhead

Review

- **Complete memory addressing mode**
  - `(%eax), 17(%eax), 2(%ebx, %ecx, 8), …`
- **Arithmetic operations**
  - `subl %eax, %ecx`
  - `sall $4,%edx`
  - `addl 16(%ebp),%ecx`
  - `leal 4(%edx,%eax),%eax`
  - `imull %ecx,%eax`

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

%rax	%eax	%r8	%r8d
%rbx	%edx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%ebx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

- **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,` …)

CF ZF SF OF

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;
```


Onward

- **For loops**
- **Switch statements**
- **Procedures**

"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;
}
```

At iteration i, $x = x_0^{2^i}$ **and the result of (`p&0x1`) is the bit in position i of p.**

- **Algorithm**
  - Exploit bit representation: $p = p_0 + 2p_1 + 2^2 p_2 + \ldots 2^{n-1} p_{n-1}$
  - 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

```
p != 0
```

Init

```
result = 1
```

Update

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

Body

```
{
  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
```

Goto Version

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

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

```
  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
```

Goto Version

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

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

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


Today

- For loops
- **Switch statements**
- Procedures


```
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

```
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
```

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)
```

Indirect jump

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
```


Assembly Setup Explanation

- **Table Structure**
  - Each target requires 4 bytes
  - Base address at **.L62**
- **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 **.L62 + edx*4**
      - Only for $0 \leq \mathbf{x} \leq 6$

how?

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
```

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
```

Systems I

x86-64 Switch Implementation

- Same general idea, adapted to 64-bit code
- Table entries 64 bits (pointers)
- Cases use revised code

```
switch(x) {
case 1:       // .L50
    w = y*z;
    break;
  . . .
}
```

```
.L50: // Case 1:
   movq   %rsi, %r8  # w = y
   imulq  %rdx, %r8  # w *= z
   movq   %r8, %rax  # Return w
   ret
```

Jump Table

```
.section .rodata
   .align 8
.L62:
  .quad    .L55   # x = 0
  .quad    .L50   # x = 1
  .quad    .L51   # x = 2
  .quad    .L52   # x = 3
  .quad    .L55   # x = 4
  .quad    .L54   # x = 5
  .quad    .L54   # x = 6
```

Systems I

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)
```

Systems I

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

```
0x08048630
0x08048650
0x0804863a
0x08048642
0x08048630
0x08048649
0x08048649
```

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


x86-64 Object Code

- **Setup**
  - Label **.L61** becomes address **0x0000000000400716**
  - Label **.L62** becomes address **0x0000000000400990**

Assembly Code

```
switch_eg:
  . . .
  ja    .L55              # if > goto default
  jmp   *.L56(,%rdi,8)   # goto JTab[x]
```

Disassembled Object Code

```
0000000000400700 <switch_eg>:
 . . .
  40070d:  77 07                  ja     400716
  40070f:  ff 24 fd 90 09 40 00   jmpq   *0x400990(,%rdi,8)
```

x86-64 Object Code (cont.)

- **Jump Table**
  - Can inspect using GDB

```
gdb asm-cntl
(gdb) x/7xg 0x400990
```

    - Examine 7 hexadecimal format "giant words" (8-bytes each)
    - Use command "help x" to get format documentation

```
0x400990:
  0x0000000000400716
  0x0000000000400739
  0x0000000000400720
  0x000000000040072b
  0x0000000000400716
  0x0000000000400732
  0x0000000000400732
```


Sparse Switch Example

```
/* 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;
  }
}
```

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

Sparse Switch Code (IA32)

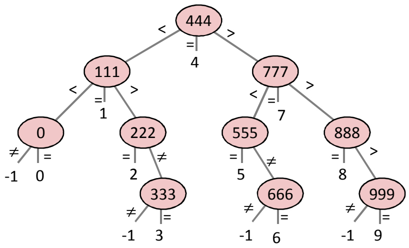
```
    movl 8(%ebp),%eax  # get x
    cmpl $444,%eax     # x:444
    je L8
    jg L16
    cmpl $111,%eax     # x:111
    je L5
    jg L17
    testl %eax,%eax    # x:0
    je L4
    jmp L14

    . . .
```

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

```
    . . .
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, for
  - switch
- **Assembler Control**
  - Conditional jump
  - Conditional move
  - Indirect jump
  - Compiler
  - Must generate assembly code to implement more complex control
- **Standard Techniques**
  - IA32 loops converted to do-while form
  - x86-64 loops use jump-to-middle
  - Large switch statements use jump tables
  - Sparse switch statements may use decision trees (not shown)
- **Conditions in CISC**
  - CISC machines generally have condition code registers