Machine-Level Programming II: Arithmetic and Control

CSci 2021: Machine Architecture and Organization
Lectures #8-9, February 6th-9th, 2015
Your instructor: Stephen McCamant

Based on slides originally by:
Randy Bryant, Dave O'Hallaron, Antonia Zhai

Arithmetic and Control

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

Complete Memory Addressing Modes

- **Most General Form**
- **D(Rb,Ri,S)   Mem[Reg[Rb]+S*Reg[Ri]+ D]**
  - D: Constant "displacement" 1, 2, or 4 bytes
  - Rb: Base register: Any of 8 integer registers
  - Ri: Index register: Any, except for `%esp`
    - Unlikely you'd use `%ebp`, either
  - S: Scale: 1, 2, 4, or 8 (***why these numbers?***)
- **Special Cases**
- **(Rb,Ri)   Mem[Reg[Rb]+Reg[Ri]]**
- **D(Rb,Ri)   Mem[Reg[Rb]+Reg[Ri]+D]**
- **(Rb,Ri,S)   Mem[Reg[Rb]+S*Reg[Ri]]**

Address Computation Examples

`%edx`	`0xf000`
`%ecx`	`0x0100`

Expression	Address Computation	Address
`0x8(%edx)`		
`(%edx,%ecx)`		
`(%edx,%ecx,4)`		
`0x80(,%edx,2)`		

Address Computation Instruction

- `leal` ***Src,Dest***
  - *Src* is address mode expression
  - Set *Dest* to address denoted by expression
- **Uses**
  - Computing addresses without a memory reference
    - E.g., translation of `p = &x[i];`
  - Computing arithmetic expressions of the form x + k*y
    - k = 1, 2, 4, or 8
- **Example**

```
int mul12(int x)
{
  return x*12;
}
```

Converted to ASM by compiler:

```
leal (%eax,%eax,2), %eax  ;t <- x+x*2
sall $2, %eax             ;return t<<2
```


Arithmetic and Control

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

Some Arithmetic Operations

- **Two Operand Instructions:**

Format		*Computation*	
`addl`	*Src,Dest*	Dest = Dest + Src	
`subl`	*Src,Dest*	Dest = Dest – Src	
`imull`	*Src,Dest*	Dest = Dest * Src	
`sall`	*Src,Dest*	Dest = Dest << Src	*Also called shll*
`sarl`	*Src,Dest*	Dest = Dest >> Src	*Arithmetic*
`shrl`	*Src,Dest*	Dest = Dest >> Src	*Logical*
`xorl`	*Src,Dest*	Dest = Dest ^ Src	
`andl`	*Src,Dest*	Dest = Dest & Src	
`orl`	*Src,Dest*	Dest = Dest | Src	

- **Watch out for argument order!**
- **No distinction between signed and unsigned int (why?)**

Some Arithmetic Operations

- **One Operand Instructions**

`incl`	*Dest*	*Dest = Dest + 1*
`decl`	*Dest*	*Dest = Dest – 1*
`negl`	*Dest*	*Dest = – Dest*
`notl`	*Dest*	*Dest = ~Dest*

- **See book for more instructions**

Arithmetic Expression Example

```
int arith(int x, int y, int z)
{
  int t1 = x+y;
  int t2 = z+t1;
  int t3 = x+4;
  int t4 = y * 48;
  int t5 = t3 + t4;
  int rval = t2 * t5;
  return rval;
}
```

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

  movl   8(%ebp), %ecx
  movl   12(%ebp), %edx
  leal   (%edx,%edx,2), %eax
  sall   $4, %eax
  leal   4(%ecx,%eax), %eax        } Body
  addl   %ecx, %edx
  addl   16(%ebp), %edx
  imull  %edx, %eax

  popl   %ebp                      } Finish
  ret
```


Understanding `arith`

```
int arith(int x, int y, int z)
{
  int t1 = x+y;
  int t2 = z+t1;
  int t3 = x+4;
  int t4 = y * 48;
  int t5 = t3 + t4;
  int rval = t2 * t5;
  return rval;
}
```

Offset	
	•
	•
	•
16	z
12	y
8	x
4	Rtn Addr
0	Old %ebp ← %ebp

```
movl   8(%ebp), %ecx
movl   12(%ebp), %edx
leal   (%edx,%edx,2), %eax
sall   $4, %eax
leal   4(%ecx,%eax), %eax
addl   %ecx, %edx
addl   16(%ebp), %edx
imull  %edx, %eax
```


Understanding `arith`

```
int arith(int x, int y, int z)
{
  int t1 = x+y;
  int t2 = z+t1;
  int t3 = x+4;
  int t4 = y * 48;
  int t5 = t3 + t4;
  int rval = t2 * t5;
  return rval;
}
```

Stack

Offset	
	•
	•
	•
16	z
12	y
8	x
4	Rtn Addr
0	Old %ebp ← %ebp

```
movl   8(%ebp), %ecx          # ecx = x
movl   12(%ebp), %edx         # edx = y
leal   (%edx,%edx,2), %eax    # eax = y*3
sall   $4, %eax               # eax *= 16 (t4)
leal   4(%ecx,%eax), %eax     # eax = t4 +x+4 (t5)
addl   %ecx, %edx             # edx = x+y (t1)
addl   16(%ebp), %edx         # edx += z (t2)
imull  %edx, %eax             # eax = t2 * t5 (rval)
```


Observations about `arith`

```
int arith(int x, int y, int z)
{
  int t1 = x+y;
  int t2 = z+t1;
  int t3 = x+4;
  int t4 = y * 48;
  int t5 = t3 + t4;
  int rval = t2 * t5;
  return rval;
}
```

- Instructions in different order from C code
- Some expressions require multiple instructions
- Some instructions cover multiple expressions
- Get exact same code when compile:
- `(x+y+z)*(x+4+48*y)`

```
movl   8(%ebp), %ecx          # ecx = x
movl   12(%ebp), %edx         # edx = y
leal   (%edx,%edx,2), %eax    # eax = y*3
sall   $4, %eax               # eax *= 16 (t4)
leal   4(%ecx,%eax), %eax     # eax = t4 +x+4 (t5)
addl   %ecx, %edx             # edx = x+y (t1)
addl   16(%ebp), %edx         # edx += z (t2)
imull  %edx, %eax             # eax = t2 * t5 (rval)
```

Another Example

```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

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

    movl 12(%ebp),%eax      }
    xorl 8(%ebp),%eax       } Body
    sarl $17,%eax           }
    andl $8185,%eax         }

    popl %ebp               } Finish
    ret                     }
```

```
movl 12(%ebp),%eax    # eax = y
xorl 8(%ebp),%eax     # eax = x^y        (t1)
sarl $17,%eax         # eax = t1>>17     (t2)
andl $8185,%eax       # eax = t2 & mask (rval)
```


Another Example

```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

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

    movl 12(%ebp),%eax      }
    xorl 8(%ebp),%eax       } Body
    sarl $17,%eax           }
    andl $8185,%eax         }

    popl %ebp               } Finish
    ret                     }
```

```
movl 12(%ebp),%eax    # eax = y
xorl 8(%ebp),%eax     # eax = x^y        (t1)
sarl $17,%eax         # eax = t1>>17     (t2)
andl $8185,%eax       # eax = t2 & mask (rval)
```


Another Example

```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

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

    movl 12(%ebp),%eax      }
    xorl 8(%ebp),%eax       } Body
    sarl $17,%eax           }
    andl $8185,%eax         }

    popl %ebp               } Finish
    ret                     }
```

```
movl 12(%ebp),%eax    # eax = y
xorl 8(%ebp),%eax     # eax = x^y        (t1)
sarl $17,%eax         # eax = t1>>17     (t2)
andl $8185,%eax       # eax = t2 & mask (rval)
```


Another Example

```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

$2^{13} = 8192$, $2^{13} - 7 = 8185$

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

    movl 12(%ebp),%eax      }
    xorl 8(%ebp),%eax       } Body
    sarl $17,%eax           }
    andl $8185,%eax         }

    popl %ebp               } Finish
    ret                     }
```

```
movl 12(%ebp),%eax    # eax = y
xorl 8(%ebp),%eax     # eax = x^y        (t1)
sarl $17,%eax         # eax = t1>>17     (t2)
andl $8185,%eax       # eax = t2 & mask (rval)
```


Arithmetic and Control

- Complete addressing mode, address computation (leal)
- Arithmetic operations
- **Control: Condition codes**
- Conditional branches
- Loops
- Switch statements

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**)

%eax
%ecx
%edx
%ebx
%esi
%edi
} **General purpose registers**

%esp **Current stack top**
%ebp **Current stack frame**

%eip **Instruction pointer**

CF ZF SF OF **Condition codes**

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**
- **Intel documentation, others, have full details**

19

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)`

20

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`

21

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 ("Sign")
`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 >)
`setae`	`~CF`	Above or equal (unsigned >=)
`setb`	`CF`	Below (unsigned <)
`setbe`	`CF|ZF`	Below or equal (unsigned <=)

22

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

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

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
```

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

23

Reading Condition Codes: x86-64

- **SetX Instructions:**
  - Set single byte based on combination of condition codes
  - Does not alter remaining 7 bytes

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

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

Bodies

```
cmpl %esi, %edi
setg %al
movzbl %al, %eax
```

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

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

24

Exercise Break: More Conditions

- Every condition can be negated by putting "n" in the mnemonic, for "not"
  - We skipped some of these conditions in the previous tables, because they were equivalent to others
- Which other conditions are these equivalent to?

1. `setng`: not greater than

2. `setnbe`: not below or equal

Equivalents of More Conditions

- Intuition: cover three cases: <, =, >
- `setng` not greater than (signed)
  - If not greater, than either less than or equal: `setle`
  - Check conditions:
    - ~(~(SF ^ OF) & ~ZF) = ~~(SF ^ OF) | ~~ZF = (SF ^ OF) | ZF ✓
- `setnbe` not below or equal (unsigned)
  - If not below or equal, must be above: `seta`
  - Check conditions:
    - ~(CF | ZF) = ~CF & ~ZF ✓

Arithmetic and Control

- Complete addressing mode, address computation (leal)
- Arithmetic operations
- x86-64
- Control: Condition codes
- **Conditional branches & Moves**
- Loops
- Switch statements

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 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       Body1
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L6
    subl    %eax, %edx          Body2a
    movl    %edx, %eax
    jmp .L7
.L6:
    subl %edx, %eax             Body2b
.L7:
    popl %ebp                   Finish
    ret
```


Conditional Branch Example (Cont.)

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

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

```
absdiff:
    pushl   %ebp                Setup
    movl    %esp, %ebp
    movl    8(%ebp), %edx       Body1
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L6
    subl    %eax, %edx          Body2a
    movl    %edx, %eax
    jmp .L7
.L6:
    subl %edx, %eax             Body2b
.L7:
    popl %ebp                   Finish
    ret
```

Conditional Branch Example (Cont.)

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

```
absdiff:
    pushl   %ebp              } Setup
    movl    %esp, %ebp
    movl    8(%ebp), %edx     } Body1
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L6
    subl    %eax, %edx        } Body2a
    movl    %edx, %eax
    jmp .L7
.L6:
    subl %edx, %eax           } Body2b
.L7:
    popl %ebp                 } Finish
    ret
```

31

Conditional Branch Example (Cont.)

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

```
absdiff:
    pushl   %ebp              } Setup
    movl    %esp, %ebp
    movl    8(%ebp), %edx     } Body1
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L6
    subl    %eax, %edx        } Body2a
    movl    %edx, %eax
    jmp .L7
.L6:
    subl %edx, %eax           } Body2b
.L7:
    popl %ebp                 } Finish
    ret
```

32

Conditional Branch Example (Cont.)

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

```
absdiff:
    pushl   %ebp              } Setup
    movl    %esp, %ebp
    movl    8(%ebp), %edx     } Body1
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L6
    subl    %eax, %edx        } Body2a
    movl    %edx, %eax
    jmp .L7
.L6:
    subl %edx, %eax           } Body2b
.L7:
    popl %ebp                 } Finish
    ret
```

33

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;
  goto Done;
Else:
  val = Else_Expr;
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

34

Using Conditional Moves

- **Conditional Move Instructions**
  - Instruction supports:
    if (Test) Dest ← Src
  - Supported in post-1995 x86 processors
  - GCC does not always use them
    - For compatibility with ancient processors
    - Enabled for x86-64
    - Use switch `-march=686` for IA32
- **Why?**
  - Branches are very disruptive to instruction flow through pipelines
  - Conditional move do not require control transfer

C Code

```
val = Test
   ? Then_Expr
   : Else_Expr;
```

Goto Version

```
  tval = Then_Expr;
  result = Else_Expr;
  t = Test;
  if (t) result = tval;
  return result;
```

35

Conditional Move Example: x86-64

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

x in %edi

y in %esi

```
absdiff:
    movl    %edi, %edx
    subl    %esi, %edx   # tval = x-y
    movl    %esi, %eax
    subl    %edi, %eax   # result = y-x
    cmpl    %esi, %edi   # Compare x:y
    cmovg   %edx, %eax   # If >, result = tval
    ret
```

36

Bad Cases for Conditional Move

Expensive Computations

```
val = Test(x) ? Hard1(x) : Hard2(x);
```

- Both values get computed
- Only makes sense when computations are very simple

Risky Computations

```
val = p ? *p : 0;
```

- Both values get computed
- May have undesirable effects

Computations with side effects

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

- Both values get computed
- Must be side-effect free

37

Arithmetic and Control

- Complete addressing mode, address computation (leal)
- Arithmetic operations
- x86-64
- Control: Condition codes
- Conditional branches and moves
- **Loops**
- Switch statements

38

"Do-While" Loop Example

C Code

```
int pcount_do(unsigned x)
{
  int result = 0;
  do {
    result += x & 0x1;
    x >>= 1;
  } while (x);
  return result;
}
```

Goto Version

```
int pcount_do(unsigned x)
{
  int result = 0;
loop:
  result += x & 0x1;
  x >>= 1;
  if (x)
    goto loop;
  return result;
}
```

- Count number of 1's in argument x ("popcount")
- Use conditional branch to either continue looping or to exit loop

39

"Do-While" Loop Compilation

Goto Version

```
int pcount_do(unsigned x) {
  int result = 0;
loop:
  result += x & 0x1;
  x >>= 1;
  if (x)
    goto loop;
  return result;
}
```

- Registers:

%edx	x
%ecx	result

```
    movl  $0, %ecx     #  result = 0
.L2:        # loop:
    movl  %edx, %eax
    andl  $1, %eax     #  t = x & 1
    addl  %eax, %ecx   #  result += t
    shrl  %edx         #  x >>= 1
    jne   .L2          #  If !0, goto loop
```

40

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

41

"While" Loop Example

C Code

```
int pcount_while(unsigned x) {
  int result = 0;
  while (x) {
    result += x & 0x1;
    x >>= 1;
  }
  return result;
}
```

Goto Version

```
int pcount_do(unsigned x) {
  int result = 0;
  if (!x) goto done;
loop:
  result += x & 0x1;
  x >>= 1;
  if (x)
    goto loop;
done:
  return result;
}
```

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

42

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:
```

43

"For" Loop Example

C Code

```
#define WSIZE 8*sizeof(int)
int pcount_for(unsigned x) {
  int i;
  int result = 0;
  for (i = 0; i < WSIZE; i++) {
    unsigned mask = 1 << i;
    result += (x & mask) != 0;
  }
  return result;
}
```

- **Is this code equivalent to other versions?**

44

"For" Loop Form

General Form

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

```
for (i = 0; i < WSIZE; i++) {
    unsigned mask = 1 << i;
    result += (x & mask) != 0;
  }
```

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{
  unsigned mask = 1 << i;
  result += (x & mask) != 0;
}
```

45

"For" Loop → While Loop

For Version

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

While Version

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

46

"For" Loop → … → Goto

For Version

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

While Version

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

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

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

47

"For" Loop Conversion Example

C Code

```
#define WSIZE 8*sizeof(int)
int pcount_for(unsigned x) {
  int i;
  int result = 0;
  for (i = 0; i < WSIZE; i++) {
    unsigned mask = 1 << i;
    result += (x & mask) != 0;
  }
  return result;
}
```

- **Initial test can be optimized away**

Goto Version

```
int pcount_for_gt(unsigned x) {
  int i;
  int result = 0;
  i = 0;                      Init
  if (!(i < WSIZE))           ! Test
    goto done;
 loop:                        Body
  {
    unsigned mask = 1 << i;
    result += (x & mask) != 0;
  }
  i++;  Update
  if (i < WSIZE)  Test
    goto loop;
 done:
  return result;
}
```

48

Announcement Break: Bomb Lab Now Out

- **Analyze malicious software with a debugger**
  - Reverse engineering based on instructions, observation, and experiment
  - Find inputs to "defuse" a bomb program so it does not "explode"
- **We've covered enough material for you to start working now**
  - E.g., control flow structure and arithmetic
  - Will also cover in discussion sections tomorrow
- **Like data lab, difficulty increases between parts**
  - Last phase especially complex
  - Start early!

Arithmetic and Control

- Complete addressing mode, address computation (leal)
- Arithmetic operations
- x86-64
- Control: Condition codes
- Conditional branches and moves
- Loops
- **Switch statements**

Switch Statement Example

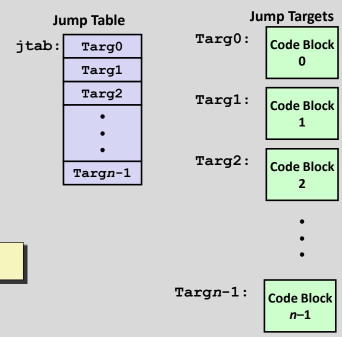
```
long switch_eg
    (long x, long y, long z)
{
    long w = 1;
    switch(x) {
    case 1:
        w = y*z;
        break;
    case 2:
        w = y/z;
        /* Fall Through */
    case 3:
        w += z;
        break;
    case 5:
    case 6:
        w -= z;
        break;
    default:
        w = 2;
    }
    return w;
}
```

- **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
}
```



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
   movl    8(%ebp), %eax     # %eax = x
   cmpl    $6, %eax          # Compare x to 6
   ja      .L2               # If unsigned > goto default
   jmp     *.L7(,%eax,4)     # Goto *JTab[x]
```

Note that **w** not initialized here

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
.L7:
  .long      .L2 # x = 0
  .long      .L3 # x = 1
  .long      .L4 # x = 2
  .long      .L5 # x = 3
  .long      .L2 # x = 4
  .long      .L6 # x = 5
  .long      .L6 # x = 6
```

Setup:

```
switch_eg:
   pushl   %ebp              # Setup
   movl    %esp, %ebp        # Setup
   movl    8(%ebp), %eax     # eax = x
   cmpl    $6, %eax          # Compare x:6
   ja      .L2               # If unsigned > goto default
   jmp     *.L7(,%eax,4)     # Goto *JTab[x]
```

Indirect jump

Assembly Setup Explanation

- **Table Structure**
  - Each target requires 4 bytes
  - Base address at `.L7`
- **Jumping**
  - Direct: `jmp .L2`
  - Jump target is denoted by label `.L2`
  - Indirect: `jmp *.L7(,%eax,4)`
  - Start of jump table: `.L7`
  - Must scale by factor of 4 (labels have 32-bits = 4 Bytes on IA32)
  - Fetch target from effective Address `.L7 + eax*4`
    - Only for $0 \le x \le 6$

Jump table

```
.section   .rodata
  .align 4
.L7:
  .long    .L2 # x = 0
  .long    .L3 # x = 1
  .long    .L4 # x = 2
  .long    .L5 # x = 3
  .long    .L2 # x = 4
  .long    .L6 # x = 5
  .long    .L6 # x = 6
```

55

Jump Table

Jump table

```
.section   .rodata
  .align 4
.L7:
  .long    .L2 # x = 0
  .long    .L3 # x = 1
  .long    .L4 # x = 2
  .long    .L5 # x = 3
  .long    .L2 # x = 4
  .long    .L6 # x = 5
  .long    .L6 # x = 6
```

```
switch(x) {
case 1:       // .L3
     w = y*z;
     break;
case 2:       // .L4
     w = y/z;
     /* Fall Through */
case 3:       // .L5
     w += z;
     break;
case 5:
case 6:       // .L6
     w -= z;
     break;
default:      // .L2
     w = 2;
}
```

56

Handling Fall-Through

```
long w = 1;
   . . .
switch(x) {
   . . .
case 2:
     w = y/z;
     /* Fall Through */
case 3:
     w += z;
     break;
   . . .
}
```

```
case 3:
        w = 1;
        goto merge;
```

```
case 2:
     w = y/z;
merge:
     w += z;
```

57

Code Blocks (Partial)

```
switch(x) {
case 1:       // .L3
     w = y*z;
     break;
  . . .
case 3:       // .L5
     w += z;
     break;
  . . .
default:      // .L2
     w = 2;
}
```

```
.L2:     # Default
  movl  $2, %eax  # w = 2
  jmp   .L8       # Goto done

.L5:     # x == 3
  movl  $1, %eax  # w = 1
  jmp   .L9       # Goto merge

.L3:     # x == 1
  movl  16(%ebp), %eax  # z
  imull 12(%ebp), %eax  # w = y*z
  jmp   .L8        # Goto done
```

58

Code Blocks (Rest)

```
switch(x) {
  . . .
  case 2:  // .L4
      w = y/z;
      /* Fall Through */
  merge:   // .L9
      w += z;
      break;
  case 5:
  case 6:  // .L6
      w -= z;
      break;
}
```

```
.L4:     # x == 2
  movl  12(%ebp), %edx
  movl  %edx, %eax
  sarl  $31, %edx
  idivl 16(%ebp)  # w = y/z

.L9:     # merge:
  addl  16(%ebp), %eax # w += z
  jmp   .L8       # goto done

.L6:     # x == 5, 6
  movl  $1, %eax        # w = 1
  subl  16(%ebp), %eax # w = 1-z
```

59

Switch Code (Finish)

```
return w;
```

```
.L8:     # done:
  popl  %ebp
  ret
```

- **Noteworthy Features**
  - Jump table avoids sequencing through cases
    - Constant time, rather than linear
  - Use jump table to handle holes and duplicate tags
  - Use program sequencing to handle fall-through
  - Don't initialize w = 1 unless really need it

60

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:       // .L3
    w = y*z;
    break;
  . . .
}
```

```
.L3:
   movq    %rdx, %rax
   imulq   %rsi, %rax
   ret
```

Jump Table

```
  .section  .rodata
  .align 8
.L7:
  .quad     .L2     # x = 0
  .quad     .L3     # x = 1
  .quad     .L4     # x = 2
  .quad     .L5     # x = 3
  .quad     .L2     # x = 4
  .quad     .L6     # X = 5
  .quad     .L6     # x = 6
```

61

IA32 Object Code

- **Setup**
  - Label `.L2` becomes address `0x8048422`
  - Label `.L7` becomes address `0x8048660`

Assembly Code

```
switch_eg:
  . . .
  ja     .L2              # If unsigned > goto default
  jmp    *.L7(,%eax,4)  # Goto *JTab[x]
```

Disassembled Object Code

```
08048410 <switch_eg>:
 . . .
 8048419:77 07          ja      8048422 <switch_eg+0x12>
 804841b:ff 24 85 60 86 04 08  jmp     *0x8048660(,%eax,4)
```

62

IA32 Object Code (cont.)

- **Jump Table**
  - Doesn't show up in disassembled code
  - Can inspect using GDB (or `objdump -s`)
  - `gdb switch`
  - `(gdb) x/7xw 0x8048660`
    - Examine 7 hexadecimal format "words" (4-bytes each)
    - Use command "`help x`" to get format documentation

```
0x8048660:    0x08048422    0x08048432    0x0804843b    0x08048429
0x8048670:    0x08048422    0x0804844b    0x0804844b
```

63

IA32 Object Code (cont.)

- **Deciphering Jump Table**

```
0x8048660:    0x08048422    0x08048432    0x0804843b    0x08048429
0x8048670:    0x08048422    0x0804844b    0x0804844b
```

Address	Value	x
0x8048660	0x8048422	0
0x8048664	0x8048432	1
0x8048668	0x804843b	2
0x804866c	0x8048429	3
0x8048670	0x8048422	4
0x8048674	0x804844b	5
0x8048678	0x804844b	6

64

Disassembled Targets

```
8048422:   b8 02 00 00 00          mov    $0x2,%eax
8048427:   eb 2a                   jmp    8048453 <switch_eg+0x43>
8048429:   b8 01 00 00 00          mov    $0x1,%eax
804842e:   66 90                   xchg   %ax,%ax # noop
8048430:   eb 14                   jmp    8048446 <switch_eg+0x36>
8048432:   8b 45 10                mov    0x10(%ebp),%eax
8048435:   0f af 45 0c             imul   0xc(%ebp),%eax
8048439:   eb 18                   jmp    8048453 <switch_eg+0x43>
804843b:   8b 55 0c                mov    0xc(%ebp),%edx
804843e:   89 d0                   mov    %edx,%eax
8048440:   c1 fa 1f                sar    $0x1f,%edx
8048443:   f7 7d 10                idivl  0x10(%ebp)
8048446:   03 45 10                add    0x10(%ebp),%eax
8048449:   eb 08                   jmp    8048453 <switch_eg+0x43>
804844b:   b8 01 00 00 00          mov    $0x1,%eax
8048450:   2b 45 10                sub    0x10(%ebp),%eax
8048453:   5d                      pop    %ebp
8048454:   c3                      ret
```

65

Matching Disassembled Targets

Value
0x8048422
0x8048432
0x804843b
0x8048429
0x8048422
0x804844b
0x804844b

```
8048422:   mov    $0x2,%eax
8048427:   jmp    8048453 <switch_eg+0x43>
8048429:   mov    $0x1,%eax
804842e:   xchg   %ax,%ax
8048430:   jmp    8048446 <switch_eg+0x36>
8048432:   mov    0x10(%ebp),%eax
8048435:   imul   0xc(%ebp),%eax
8048439:   jmp    8048453 <switch_eg+0x43>
804843b:   mov    0xc(%ebp),%edx
804843e:   mov    %edx,%eax
8048440:   sar    $0x1f,%edx
8048443:   idivl  0x10(%ebp)
8048446:   add    0x10(%ebp),%eax
8048449:   jmp    8048453 <switch_eg+0x43>
804844b:   mov    $0x1,%eax
8048450:   sub    0x10(%ebp),%eax
8048453:   pop    %ebp
8048454:   ret
```

66

Exercise Break: `switch` Bounds

- **Every jump table needs to check that the index is in bounds**
  - For each of these code patterns, what indexes are allowed?

```
cmpl     $5, %eax
ja       .Ldefault
jmp      *.L1(,%eax,4)
```

Unsigned <= 5: 0 .. 5

```
andl     $7, %eax
jmp      *.L2(,%eax,4)
```

Low 3 bits: 0 .. 7

```
movzbl   8(%ebp), %eax
jmp      *.L3(,%eax,4)
```

Low 8 bits: 0 .. 255

67

Summarizing

- **C Control**
  - if-then-else
  - do-while
  - while, for
  - switch
- **Assembler Control**
  - Conditional jump
  - Conditional move
  - Indirect jump
  - Compiler generates code sequence 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

68

Summary

- **These slides**
  - Complete addressing mode, address computation (leal)
  - Arithmetic operations
  - Control: Condition codes
  - Conditional branches & conditional moves
  - Loops
  - Switch statements
- **Next Up**
  - Stack
  - Call / return
  - Procedure call discipline

69