X86 Assembly, and C-to-assembly

- 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

Compiling Into Assembly

**C Code**
```c
int sum(int a, int b)
{
    int t = a + b;
    return t;
}
```

**Generated IA32 Assembly**
```
sum:
    pushl %ebp
    movl %esp, %ebp
    addl $12, %esp
    leal (%ebp), %eax
    addl $0, %eax
    movl %eax, %ebx
    popl %ebp
    ret
```

Obtain with command
```
gcc -0 -S code.c
```
Produces file code.s

Turning C into Object Code

- Code in files `p1.c` `p2.c`
- Compile with command: `gcc -o p1.o p2.o -o p`
- Use optimizations (-O)
- Put resulting binary in file `p`

Three Kinds of Instructions

- Perform arithmetic function on register or memory data

  ```
  c = a - b
  ```

- Transfer data between memory and register

  ```
  load data from memory into register
  ```
  ```
  store register data into memory
  ```

- Transfer control (control flow)

  ```
  Unconditional jumps to/from procedures
  ```
  ```
  Conditional branches
  ```

Assembly Characteristics: Data Types

- “Integer” data of 1, 2, or 4 bytes
  ```
  signed:
  ```
  Addresses (untyped pointers)

- Floating point data of 4, 8, or 16 bytes

- How about arrays, structs, etc?

Object Code

- Assembler:
  ```
  Translates into...
  ```
  ```
  Binary encoding of each instruction
  ```
  ```
  Nearly-complete image of executable code
  ```
  ```
  Missing linkages between code in different files
  ```

- Linker:
  ```
  Resolves references between files
  ```
  ```
  Combines with static run-time libraries
  ```
  ```
  E.g., code for malloc, printf
  ```
  ```
  Some libraries are dynamically linked
  ```
  ```
  Linking occurs when program begins execution
  ```
Example

- C Code
  - Add two signed integers

- Assembly
  - Add 2 2-byte integers
  - "long" words in GCC speak
  - Same instruction whether signed or unsigned
  - Use registers:
    - eax: Memory
    - ebx: Memory
    - ebp: Memory
  - Operands:
    - eax = eax + ebx
    - ebx = eax + ebx
  - Return function value in eax

Object Code

- 3 byte instruction
- Stored at address 0x401046

Disassembling Object Code

- Disassembler
  - objdump -d p
  - Useful tool for examining object code
  - Analyzes bit pattern of series of instructions
  - Produces approximate rendition of assembly code
  - Can run on an either a.out (complete executable) or .o file

Integer Registers (IA32)

<table>
<thead>
<tr>
<th>eax</th>
<th>ebx</th>
<th>ecx</th>
<th>edx</th>
<th>esi</th>
<th>edi</th>
<th>ebp</th>
</tr>
</thead>
</table>

16-bit virtual registers

(Mostly obsolete)

Moving Data: IA32

- Moving Data
  - move source, dest
    - x is one of b, w, l
  - move source, dest
    - Move 4-byte "long word"
  - move source, dest
    - Move 2-byte "word"
  - move source, dest
    - Move 1-byte "byte"

- Lots of these in typical code

Moving Data: IA32

- Moving Data
  - mov source, dest

- Operands

- Register
  - One of 8 integer registers
  - eax, ebx, ecx, edx, esi, edi, ebp, esp
  - But esp and ebp reserved for special use
  - Others have special uses for particular instructions

- Memory
  - 4 consecutive bytes of memory at address given by register
  - Simplest example: \( \text{lea eax} \)
  - Various other "address modes"
movl Operand Combinations

Source | Dest | Src, Dest | C Analog
--- | --- | --- | ---
Imm | Reg | movl $0x4, %eax | temp = 0x4
Imm | Mem | movl $147, (%eax) | *p = 147
Reg | Reg | movl %eax, %edx | 
Reg | Mem | movl %eax, (%edx) | 
Mem | Reg | movl (%eax), %edx | 

Cannot do memory-memory transfer with a single instruction. How do you copy from a memory location to another then?

movl Operand Combinations

Source | Dest | Src, Dest | C Analog
--- | --- | --- | ---
Imm | Reg | movl $0x4, %eax | temp = 0x4
Imm | Mem | movl $147, (%eax) | *p = 147
Reg | Reg | movl %eax, %edx | temp2 = temp1;
Reg | Mem | movl %eax, (%edx) | *p = temp;
Mem | Reg | movl (%eax), %edx | 

Simple Memory Addressing Modes

- Normal (R) Mem[Reg[R]]
  - Register R specifies memory address
  - movl (%ecx), %eax

- Displacement [D(R)] Mem[Reg[R]+D]
  - Register R specifies start of memory region
  - Constant displacement D specifies offset
  - movl %ebp, %edx

Using Simple Addressing Modes

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

Understanding Swap

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

Register Values
- %eax, %edx, %ebp, %ebp
- %eax, %edx, %ebp
- %eax, %edx, %ebp
- %eax, %edx
- %eax, %edx
- %eax

Understanding Swap

<table>
<thead>
<tr>
<th>Address</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x124</td>
<td>0</td>
</tr>
<tr>
<td>0x120</td>
<td>4</td>
</tr>
<tr>
<td>0x11c</td>
<td>4</td>
</tr>
<tr>
<td>0x118</td>
<td>4</td>
</tr>
<tr>
<td>0x114</td>
<td>4</td>
</tr>
<tr>
<td>0x110</td>
<td>4</td>
</tr>
<tr>
<td>0x10c</td>
<td>4</td>
</tr>
<tr>
<td>0x104</td>
<td>4</td>
</tr>
<tr>
<td>0x100</td>
<td>4</td>
</tr>
</tbody>
</table>
Complete Memory Addressing Modes

- Most General Form
  \[ D(Rb,Rl) \quad Mem[Reg[Rb]+Reg[Rl]+D] \]
  - D: Constant "displacement" 1, 2, or 4 bytes
  - Rb: Base register: Any of 8 integer registers
  - Rl: Index register: Any, except for `esp`
  - Unlike you'd use `esp`, either
- Special Cases
  - \( (Rb,Rl) \quad Mem[Reg[Rb]+Reg[Rl]] \)
  - \( D(Rb,Rl) \quad Mem[Reg[Rb]+Reg[Rl]+D] \)
  - \( (Rb,Rl,5) \quad Mem[Reg[Rb]+5\times Reg[Rl]] \)

Address Computation Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 0x8(%edx) )</td>
<td>0x0000 + 0x8</td>
<td>0x0008</td>
</tr>
<tr>
<td>( (%edx,%ecx) )</td>
<td>0x0000 + 0x100</td>
<td>0x100</td>
</tr>
<tr>
<td>( (%edx,%ecx,4) )</td>
<td>0x0000 + 4\times0x100</td>
<td>0x400</td>
</tr>
<tr>
<td>( 0x80!(%edx,2) )</td>
<td>2\times0x0000 + 0x80</td>
<td>0x100</td>
</tr>
</tbody>
</table>

Address Computation Instruction

- **lea**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 \( p = \&a[i] \):
    - Computing arithmetic expressions of the form \( x + k*i \)
      - \( k = 1, 2, 4, \) or \( 8 \)

Some Arithmetic Operations

- Two Operand Instructions:
  - add\( l \) Src, Dest \( \rightarrow \) Dest = Dest + Src
  - sub\( l \) Src, Dest \( \rightarrow \) Dest = Dest - Src
  - sl\( l \) Src, Dest \( \rightarrow \) Dest = Dest \& Src
  - sar\( l \) Src, Dest \( \rightarrow \) Dest = Dest \&\& Src
  - shr\( l \) Src, Dest \( \rightarrow \) Dest = Dest >> Src
  - cl\( l \) Src, Dest \( \rightarrow \) Dest = Dest \& Src
  - and\( l \) Src, Dest \( \rightarrow \) Dest = Dest \& Src
  - or\( l \) Src, Dest \( \rightarrow \) Dest = Dest | Src
  - Also called cl\( l \)
  - Arithmetic
  - Logical

Some Arithmetic Operations

- Two Operand Instructions:
  - add\( l \) Src, Dest \( \rightarrow \) Dest = Dest + Src
  - sub\( l \) Src, Dest \( \rightarrow \) Dest = Dest - Src
  - sl\( l \) Src, Dest \( \rightarrow \) Dest = Dest \& Src
  - sar\( l \) Src, Dest \( \rightarrow \) Dest = Dest \&\& Src
  - shr\( l \) Src, Dest \( \rightarrow \) Dest = Dest >> Src
  - cl\( l \) Src, Dest \( \rightarrow \) Dest = Dest \& Src
  - and\( l \) Src, Dest \( \rightarrow \) Dest = Dest \& Src
  - or\( l \) Src, Dest \( \rightarrow \) Dest = Dest | Src
  - Also called cl\( l \)
  - Arithmetic
  - Logical

- No distinction between signed and unsigned int (why?)
Some Arithmetic Operations

- One Operand Instructions
  - inc1 Dest: Dest = Dest + 1
  - dec1 Dest: Dest = Dest - 1
  - neg1 Dest: Dest = -Dest
  - not1 Dest: Dest = -Dest

- See book for more instructions

Using leal for Arithmetic Expressions

```c
int arith
(int x, int y, int z)
{
    int t1 = x+y;
    int t2 = z1;
    int t3 = x4;
    int t4 = y + 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

Set Up

push $ebp
movl $esp, $ebp
movl $8($ebp), $eax
movl $12($ebp), $edx
leal ($edx,$eax), %ecx
leal ($edx,$edx,2), %edx
call $4,$edx
addl $16($ebp), $ecx
lea 4($edx,$eax), %eax
imull $ecx,$eax
movl $eax,$ebp
popl $ebp
ret

Body

Finish

Understanding arith

```
int arith
(int x, int y, int z)
{
    int t1 = x+y;
    int t2 = z1;
    int t3 = x4;
    int t4 = y + 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

What does each of these instructions mean?

```
movi $8($ebp), %eax
movl $12($ebp), %edx
leal (%edx,%eax), %ecx
leal (%edx,%dx,2), %edx
call $4,%edx
addl $16($ebp), %ecx
lea 4(%edx,%eax), %eax
imull % ecx,% eax
movl % eax,% eax
```

Understanding arith

```
int arith
(int x, int y, int z)
{
    int t1 = x+y;
    int t2 = z1;
    int t3 = x4;
    int t4 = y + 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

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

Understanding arith

```
int arith
(int x, int y, int z)
{
    int t1 = x+y;
    int t2 = z1;
    int t3 = x4;
    int t4 = y + 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

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

Understanding arith

```
int arith
(int x, int y, int z)
{
    int t1 = x+y;
    int t2 = z1;
    int t3 = x4;
    int t4 = y + 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

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

6
Conditionals and Control Flow
- A test/conditional branch is sufficient to implement most control flow constructs offered in higher level languages
  - if (condition) then [...] else [...]
  - while(condition) [...] do [...] while (condition)
  - for(initialization; condition; i[...])
- (Unconditional branches implemented some related control flow constructs)
  - break, continue

Jumping
- JX instructions
  - Jump to different part of code depending on condition codes

<table>
<thead>
<tr>
<th>JX</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3w</td>
<td>1</td>
<td>Unconditional</td>
</tr>
<tr>
<td>3w</td>
<td>SF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>3w</td>
<td>SF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>3w</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>3w</td>
<td>SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>3w</td>
<td>SF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>3w</td>
<td>SF</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>3w</td>
<td>SF</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>3w</td>
<td>SF</td>
<td>Absolute (unsigned)</td>
</tr>
<tr>
<td>3w</td>
<td>SF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>

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` 32-bit/64-bit: `t := a + b`
  - CF set if carry out from most significant bit (unsigned comparison)
  - 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)`
- Not set by `lea` instruction (beware!)
- Full documentation (IA32)

Condition Codes (Explicit Setting: Compare)
- Explicit Setting by Compare Instruction
  - `cmp lcmpq` 32-bit, 64-bit
  - `%eax`, `%ecx`, `%edx`, `%ebx`, `%esi`, `%edi`
  - Current stack frame
  - Current stack top
  - General purpose registers
  - Instruction pointer
- Explicit Setting by Compare Instruction
  - `%eax`, `%ecx`, `%edx`, `%ebx`, `%esi`, `%edi`
  - 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` 32-bit, 64-bit
  - `%edx`, `%edi`
  - Like computing `a&b` without setting destination
  - Sets condition codes based on value of `a&b`
  - Useful to have one of the operands be a mask
  - ZF set when `a&b = 0`
  - SF set when `a&b < 0`
- Explicit Setting by Test Instruction
  - `%eax`, `%ecx`, `%edx`, `%ebx`, `%esi`, `%edi`
  - Like computing `a& b` without setting destination
  - Sets SF and ZF, check if eax is 0.
Reading Condition Codes

SetX Instructions:
- Set a single byte based on combinations of condition codes

<table>
<thead>
<tr>
<th>SetX</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sete</td>
<td>SF</td>
<td>Equal/Zero</td>
</tr>
<tr>
<td>setle</td>
<td>SF</td>
<td>Not Equal/Net Zero</td>
</tr>
<tr>
<td>sets</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>setg</td>
<td>SF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>setge</td>
<td>SF</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>setl</td>
<td>SF</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>setle</td>
<td>SF</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>setc</td>
<td>CF</td>
<td>Above (Unsigned)</td>
</tr>
<tr>
<td>setb</td>
<td>CF</td>
<td>Below (Unsigned)</td>
</tr>
</tbody>
</table>

Reading Condition Codes (Cont.)

- SetX instructions based on combination of condition codes
- One of 8 addressable byte registers
  - Does not alter remaining 3 bytes
  - Typically use movb1 to finish job

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

Body

```c
movl 12(%ebp), %eax
cmpl %eax, %eax
setl %al
movb %al, %eax
```

Note: Inverted ordering!

Jumping

JX Instructions:
- Jump to different part of code depending on condition codes

<table>
<thead>
<tr>
<th>JX</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jae</td>
<td>SF</td>
<td>Equal/Zero</td>
</tr>
<tr>
<td>jae</td>
<td>SF</td>
<td>Not Equal/Net Zero</td>
</tr>
<tr>
<td>jg</td>
<td>SF</td>
<td>Greater</td>
</tr>
<tr>
<td>jge</td>
<td>SF</td>
<td>Greater or Equal</td>
</tr>
<tr>
<td>jl</td>
<td>SF</td>
<td>Less</td>
</tr>
<tr>
<td>jle</td>
<td>SF</td>
<td>Less or Equal</td>
</tr>
<tr>
<td>jx</td>
<td>CF</td>
<td>Above (Unsigned)</td>
</tr>
<tr>
<td>jx</td>
<td>CF</td>
<td>Below (Unsigned)</td>
</tr>
</tbody>
</table>

Conditional Branch Example

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

```
abdiff:
    push %ebp
    movl %esp, %ebp
    movl 8(%ebp), %edx
    movl 12(%ebp), %eax
    cmpl %eax, %edx
    jle L7
    subl %edx, %eax
    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;
}
```

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

- Callsite "goto" as means of transferring control
- Closer to machine-level programming style
- Generally considered bad coding style
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;
}
```

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;
}
```

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;
}
```

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;
}
```

General Conditional Expression Translation

**C Code**

```
val = Test ? Then-Exp : Else-Exp;
val = Any ? A1 : A2;
```

**Goto Version**

```
if (Test) goto Else;
val = Then-Exp;
else;
val = Else-Exp;
```

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

- How would you make this efficient?

Conditionals: x86-64

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

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

- 0x100  cmp  r2, r3  0x1000
- 0x102  je  0x70  0x1002
- 0x104  -  0x1004
- 0x172  add  r3, r4  0x1072

- PC relative branches are relocatable
- Absolute branches are not

Machine Programming II: Instructions (cont’d)

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

Compiling Loops

- How to compile other loops should be clear to you
  - The only slightly tricky part is to be sure where the conditional branch occurs: top or bottom of the loop
  - Q: How is for (i=0; i<100; i++) implemented?
  - Q: How are break and continue implemented?

“Do-While” Loop Example

C Code
```c
int fact_do(int n) {
    int result = 1;
    do { 
        result *= n;
        n -= 1;
    } while (n > 1);
    return result;
}
```

Goto Version
```c
int fact_goto(int n) {
    int result = 1;
    loop: 
    result *= n;
    n -= 1;
    if (n > 1) goto loop;
    return result;
}
```

- Use backward branch to continue looping
- Only take branch when “while” condition holds

“Do-While” Loop Compilation

Goto Version
```c
int fact_goto(int n) {
    int result = 1;
    loop: 
    result *= n;
    n -= 1;
    if (n > 1) goto loop;
    return result;
}
```

Assembly
```assembly
fact_goto:
push ebp
mov esp, ebp
mov $1, eax
mov $0, (%esp), %edx
L11:
    imul %edx, %eax
decl %eax
jg L11
mov %ebp, %esp
popl %ebp
ret
```

Translation?

Goto Version
```c
int fact_goto(int n) {
    int result = 1;
    loop: 
    result *= n;
    n -= 1;
    if (n > 1) goto loop;
    return result;
}
```

Assembly
```assembly
fact_goto:
push ebp
mov esp, ebp
mov $1, eax
mov $0, (%esp), %edx
L11:
    imul %edx, %eax
decl %eax
jg L11
mov %ebp, %esp
popl %ebp
ret
```

Registers:
- %edx
- x
- result

Goto Version
```c
int fact_goto(int n) {
    int result = 1;
    loop: 
    result *= n;
    n -= 1;
    if (n > 1) goto loop;
    return result;
}
```

Assembly
```assembly
fact_goto:
push ebp
mov esp, ebp
mov $1, eax
mov $0, (%esp), %edx
L11:
    imul %edx, %eax
decl %eax
jg L11
mov %ebp, %esp
popl %ebp
ret
```

Registers:
- %edx
- x
- result

Goto Version
```c
int fact_goto(int n) {
    int result = 1;
    loop: 
    result *= n;
    n -= 1;
    if (n > 1) goto loop;
    return result;
}
```

Assembly
```assembly
fact_goto:
push ebp
mov esp, ebp
mov $1, eax
mov $0, (%esp), %edx
L11:
    imul %edx, %eax
decl %eax
jg L11
mov %ebp, %esp
popl %ebp
ret
```

Registers:
- %edx
- x
- result

Goto Version
```c
int fact_goto(int n) {
    int result = 1;
    loop: 
    result *= n;
    n -= 1;
    if (n > 1) goto loop;
    return result;
}
```

Assembly
```assembly
fact_goto:
push ebp
mov esp, ebp
mov $1, eax
mov $0, (%esp), %edx
L11:
    imul %edx, %eax
decl %eax
jg L11
mov %ebp, %esp
popl %ebp
ret
```

Registers:
- %edx
- x
- result

11
General “Do-While” Translation

C Code
```c
do
  Body
while (Test);
```

Goto Version
```c
Loop:
  Body
if (Test) goto loop
```

- **Body**: { Statement;
  Statement;
  Statement;
}
- **Test** returns integer:
  - 0 interpreted as false
  - # interpreted as true

“While” Loop Example

C Code
```c
int fact_while(int n) {
  int result = 1;
  while (n > 1) {
    result *= n;
    n = n-1;
  }
  return result;
}
```

Goto Version #1
```c
int fact_while_goto1(int n) {
  int result = 1;
  Loop:
  if (! (n > 1))
    goto done;
  result *= n;
  n = n-1;
  goto Loop;
  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
```c
int fact_while(int n) {
  int result = 1;
  while (n > 1) {
    result *= n;
    n = n-1;
  }
  return result;
}
```

Goto Version #2
```c
int fact_while_goto2(int n) {
  int result = 1;
  Loop:
  if (! (n > 1))
    goto done;
  result *= n;
  n = n-1;
  goto Loop;
  return result;
}
```

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

General “While” Translation

C Code
```c
while (Test) {
  Body
}
```

Goto Version
```c
while (Test)
  Body
```

- **Do-While Version**
  ```c
  if (! (Test))
    goto done;
  Body
  while (Test);
  done:
  ```

- **Goto Version**
  ```c
  if (! (Test))
    goto done;
  Body
  while (Test);
  done:
  ```

New Style “While” Loop Translation

C Code
```c
int fact_while(int n) {
  int result = 1;
  while (n > 1) {
    result *= n;
    n = n-1;
  }
  return result;
}
```

Goto Version #3
```c
int fact_while_goto3(int n) {
  int result = 1;
  goto middle;
  if (! (n > 1))
    goto done;
  middle:
  result *= n;
  n = n-1;
  goto middle;
  return result;
}
```

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

Jump-to-Middle While Translation

C Code
```c
while (Test) {
  Body
}
```

Goto Version
```c
while (Test)
  Body
```

- **Goto Version**
  ```c
  goto middle;
  Body
  middle:
  if (Test)
    goto loop;
  ```

- **Goto (Previous) Version**
  ```c
  if (! (Test))
    goto done;
  Body
  if (Test)
    goto middle;
  ```

- Avoids duplicating test code
- Unconditional goto incurs no performance penalty
- For loops compiled in similar fashion
Jump-to-Middle Example

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

Quick Review

- **Complete memory addressing mode**
  - `%eax`, `%ebx`, `%ecx`, `%edx`, ...

- **Arithmetic operations that do set condition codes**
  - `%subl %eax, %ecx`, `%addl %esi, %edx`
  - `%cmpxchg %eax, %edi`
  - `%imul %ecx, %eax`
  - `%addl %edx, %esi`

- **Arithmetic operations that do NOT set condition codes**
  - `%isal %edx, %eax`
  - `%addl %edx, %esi`

Quick Review

- x86-64 vs. IA32
  - Integer registers: `%eax`, `%ecx`, `%edx`, `%ebx`, `%esi`
  - `%movq`, `%addq`, `%subq`, `%mulq`, `%imulq` better support for passing function arguments in registers

- Control
  - Condition code registers
  - `%cmpxchg %eax, %edi`
  - `%test %eax, %edi`

“For” Loop Example: Square-and-Multiply

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

Algorithm

- Explicit bit representation: 
  - \[ p = p_0 + 2^1 p_1 + 2^2 p_2 + \ldots + 2^n p_n \]
- Gives \[ x^p = x^{p_0} \cdot x^{2^1 p_1} \cdot \ldots \cdot x^{2^n p_n} \]
- \[ p_i = 0 \text{ when } p_i = 0 \]
- \[ p_i = x \text{ when } p_i = 1 \]
- Complexity \( O(\log p) \)

ipwr Computation

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

```
<table>
<thead>
<tr>
<th>Iteration</th>
<th>Result</th>
<th>( x^2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```
IA32 Object Code

- Jump Table
  - Doesn't show up in disassembled code
  - Can inspect using GDB
    gdb asm-cntl
    (gdb) x/1w $esp+0x0804888dc
    - Examine 16-bit hexadecimal format "words" (4-bytes each)
    - Use command "help x" to get format documentation

Disassembled Object Code

0x08048610 <switch_sp>:
0x08048622: ff 99 dc 88 04 08       jmp *0x080488b8d1, 4
0x08048624:

Disassembled Targets

Matching Disassembled Targets

Summarizing

- C Control
  - if-else, do-while, while, for

- Assembler Control
  - Conditional jump
  - Indirect jump
  - Compiler
    - Must generate assembly code to implement more complex control