CS270 classnotes

Raphael Finkel

February 6, 2018

1 Intro

1. Handout 1 — My names
2. Plagiarism — read aloud
3. E-mail list: cs270@cs.uky.edu
4. Labs: five throughout the semester, typically on Fridays. The first one is this Friday. Labs count toward your grade; you must do them during the lab session.
5. Projects are available at the course website, https://www.cs.uky.edu/~raphael/courses/CS270.html. First project — Review of the C language
6. Accounts in MultiLab if you want; every student has a virtual machine as well, at name@name.netlab.uky, where name is your LinkBlue account name. The first lab will acquaint you with this facility.

2 A brief introduction to systems programming

2. Programs are written in C, which can do low-level manipulation of data, but is error-prone.
3. Files: \texttt{open()}, \texttt{read()}/\texttt{write()}, \texttt{close()}

4. Memory: \texttt{malloc()} gives a pointer to a character array.

3 Software tools

<table>
<thead>
<tr>
<th>Use (client)</th>
<th>Spec</th>
<th>Programmer</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementation</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Class 2, 1/12/2018 Lab 1

4 Abstraction and reality

1. Class 3, 1/17/2018

2. Most CS and CE courses emphasize abstraction; it matches how we think, and it lets us hide implementation details and complexity.

3. But hardware has limits that our abstractions lack (maximum size of an integer, for instance). If we hide implementation details, we are at risk of inefficiency and inability to cooperate with other components.

4. Examples of hardware limits

   a) C \texttt{int} is not an integer: \(50000 \times 50000 = 2500000000\), but the \texttt{int} has value \(-1794967296\).

   b) C \texttt{float} is not real: \(1 \times 10^20 + 3.14 - 1 \times 10^20 = 3.14\), but the float result is 0.0.

   c) Programming languages hide the instructions that are executed.

   d) Layout in memory affects performance (caches, pages). Example:
#define BIG 10000
int from[BIG*BIG], to[BIG*BIG];

void copyij(int src[BIG][BIG], int dst[BIG][BIG]) {
    int row, col;
    for (row = 0; row < BIG; row += 1) // reorder?
        for (col = 0; col < BIG; col += 1) // reorder?
            dst[row][col] = src[row][col];
} // copy

int main() {
    copyij(from, to);
    return (0);
} // main

One experiment shows that in the order given, user time is 0.3 seconds; with interchanged order, user time is 1.4 seconds.

5. We no longer teach assembler-language programming, because compilers are much better and more patient than assembler-language programmers.

6. But you need to understand computation at the assembler level.
   (a) When your program has a bug, high-level models can fail.
   (b) To improve performance, you need to understand what optimizations the compiler can and cannot do.
   (c) When you write system software, what actually runs is machine code.
   (d) Operating systems need to deal with the intricacies of machine code as they manipulate processes (keeping track of floating point registers, the program counter, memory mapping tables)
   (e) Malware is often written in x86 assembler.

5 Memory-referencing bugs

C (and C++) are subject to memory-referencing bugs.

1. Array references out of bounds. Example (the actual result is architecture-specific; the results shown are on the x86_64)
```c
void fun(int index) {
    printf("fun(%d):\n", index);
    double d[1] = {3.14};
    long int a[2];
    a[index] = 9223372036854775803L;
    printf("%lf\n", d[0]);
}
```

```c
int main() {
    fun(0);  // 3.14
    fun(1);  // 3.14
    fun(2);  // 3.14
    fun(-1); // 3.14
    fun(-2); // nan (not a number)
    fun(-3); // 3.14
    fun(3);  // 3.14 then fault during the return
              // *** stack smashing detected ***: ./fun terminated
    return(0);
}
```

Reason: a and d are adjacent on the stack; d at a lower address than a. When you go before the start of a, you might modify d; when you go beyond the end of a, you might access saved state on the stack, ruining the return address.

2. Dereferencing invalid pointers
3. Improper allocating and deallocating memory regions
4. Unfortunately, the symptoms may be unrelated to the causes, and the effect might be visible only long after it is generated.
5. Some languages prevent such errors: Java, JavaScript, Python, Perl, Ruby, but they are not usually used for systems programming.
6. There are tools to help you detect referencing errors (such as valgrind).

6 Binary representation

1. Bits are represented by two voltages, one for 0 and another for 1. A typical representation would be 0.3V for 0 and 3.0V for 1, but every architecture bases the values on the particular kind of transistors it uses. When a voltage changes from one value to the other, there is
an intermediate time at which its value is indeterminate; hardware carefully avoids inspecting the voltage then.

2. We usually think of numbers in decimal (base 10), but for systems programming, we sometimes need to use binary (base 2) or hexadecimal (base 16) representation.

(a) Base 2 uses only digits 0 and 1. Represent, for instance, 5.25 as $101.01_2$. Some numbers can be represented exactly in decimal but not in binary: $5.3 = 101.0100110011..._2$.

(b) Base 16 uses digits 0 . . . 9, A, B, C, D, E, F. The letters are usually written in capital letters. Each hex digit corresponds to four bits. $285.3 = 11D.4CCCCCCC..._{16}$

(c) I used the bc program to compute these values:

```
bc
scale = 10
obase = 16
285.3000
```

3. A byte is usually 8 bits. (The official name, used in computer networks, is octet, but we’ll just say “byte”). When treated as an unsigned integer, a byte has values ranging from 0 to 255 (or $11111111_2 = FF_{16}$).

4. Signed integers using $n$ bits can store numbers in the range $-2^{n-1}...2^{n-1} - 1$. For $n = 32$, the range is $-2147483648...2147483647$ or $(-80000000_{16}...7FFFFFFF_{16})$.

5. [Class 4, 1/19/2018]: Lab 2

6. [Class 5, 1/22/2018]

7. Modern computers represent signed integers in a format called 2’s complement.

(a) In unsigned form, these range from 0 (represented as all 0 bits) to $2^n - 1$ (represented by all 1 bits).

(b) In signed form, if the high bit is 0, the number is positive, in the range 0 . . . $2^{n-1} - 1$.

(c) In signed form, if the high bit is 1, the number is negative, in the range $-2^{n-1}... - 1$.

(d) To negate a number:
i. flip all the bits (0 becomes 1, 1 becomes 0)
ii. add 1 (ignoring any carry out of the most significant bit).

(e) Example \((n = 4)\): \(6_{10} = 0110_2\). Flipped: 1001. Final result: 1010.
(f) To negate again: \(-6_{10} = 1010_2\). Flipped: 0101. Final result: 0110.
(g) Negating 0 gives 0.
(h) The most negative number (when \(n = 4\)) is \(-8_{10} = 1000_2\).
     Negating this number leaves it unchanged.

8. It’s easy to see how many distinct values you can store in \(n\) bits. Since every bit can be 0 or 1, there are \(2^n\) possibilities. Luckily, \(2^{10} \approx 10^3\), so \(2^{32} = 2^2 \times 2^{30} = 4 \times (2^{10})^3 \approx 4 \times (10^3)^3 = 4 \times 10^9 = 4\) billion. Or just remember that
   \(2^{10} = 1024 \approx 10^3 = 1\) thousand (kilo or K);
   \(2^{20} = 1048576 \approx 10^6 = 1\) million (mega or M);
   \(2^{30} = 1073741824 \approx 10^9 = 1\) billion (giga or G);
   \(2^{40} \approx 10^{12} = 1\) trillion (tera or T);
   \(2^{50} \approx 10^{15} = 1\) quadrillion (peta or P);
   So \(2^{32} = 4\)G.

9. What is the largest signed integer you can store in 16 bits? (Answer: \(2^{15} - 1 = 32767\))

10. How many bits do you need to store 4893? It’s slightly more than \(4 \times 10^3\), so slightly more than 12 bits. (The right answer is 13 bits, but in a signed representation, at least 14 bits.)

7 C types and their sizes

1. Unfortunately, C declarations are machine-specific. Here is the size in bytes of various declarations.

<table>
<thead>
<tr>
<th>C declaration</th>
<th>x86</th>
<th>x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long long</td>
<td>12</td>
<td>16</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>12</td>
<td>16</td>
</tr>
<tr>
<td>pointer</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>
8 Byte ordering

1. If a word has more than one byte, what is the order?
2. Example: 19088743 = 01234567\text{16}
3. Big-endian: Least significant byte has the highest address. (Sun, PPC, Mac, Internet). The bytes, in order, are 0x01, 0x23, 0x45, 0x67.
4. Little-endian: Least significant byte has the lowest address. (x86). The bytes, in order, are 0x67, 0x45, 0x23, 0x01.
5. You can use the \texttt{od} program to show a file in bytes, characters, integers, ... For instance:

```
    od -t x4 -t o1 /etc/hosts | head
```

6. You can also use a program:

```
1 typedef unsigned char *pointer;
2 void show_bytes(pointer start, int len){
3     int i;
4     for (i = 0; i < len; i += 1) {
5         printf("%p	0x%.2x\n", start+i, start[i]);
6     }
7     printf("\n");
8 } // show_bytes
9 int main() {
10    int a = 15213;
11    printf("int a = %d (0x%x)\n", a, a);
12    show_bytes((pointer) &a, sizeof(int));
13 } // main
```

The output is:

```
int a = 15213 (0x00003b6d);
0x7fffdf0d 0x00003b6d
0x7fffdf0e 0x00003b6c
0x7fffdf0f 0x00003b6b
0x7fffdf10 0x00003b6a
0x7fffdf11 0x00003b69
```
9 Memory organization

1. We usually address memory in **bytes**, although older computers used to measure in “words”, which could be of any length (PDP-10: 36 bits per word).

2. When a program is running, we call it a **process**.

3. From a process’s point of view, memory looks like a long array, starting at byte address 0 and going to some limit determined by the operating system. (On Linux for x86_32 memory is limited to 3GB; on our machine, the x86_64, it is limited to 128TB.)

4. The operating system creates a separate address space for each process. We say that process address spaces are **virtual**, because when a process refers to address \( n \), it is very likely not at physical address \( n \).

5. Because processes get individual address spaces, they cannot read or write in each other’s address spaces, although the operating system can also arrange for some sharing.

6. The operating system allocates **physical** space, which also looks like an array ranging from address 0 to a limit determined by how much physical memory the machine has.

7. [Class 6, 1/24/2018]

8. Within a process, the program uses memory for various purposes. The compiler decides where in memory to put various items, including the instructions, initialized data, uninitialized data, stack, and heap.

9. A 32-bit architecture generally means that integers are contained in 32 bits, and that virtual addresses use 32 bits (unsigned). The maximum address is therefore 4G-1. That memory size is too small for some applications.

10. A 64-bit architecture generally means that integers are contained in 64 bits, and that virtual addresses use 64 bits (unsigned). The maximum address is therefore about \( 1.8 \times 10^{19} \). The x86_64 architecture supports only 48-bit addresses, which gives 256TB.

11. Architectures generally support multiple data formats. So a 64-bit architecture might be able to manipulate 8-bit, 16-bit, 32-bit, 64-bit, and 128-bit integers.
10 Strings and Buffers

1. A C string is an array of bytes, each representing a single character, terminated by a null (zero) byte.

2. Declaration

(a) char *myString;
(b) char myString[];
(c) char myString[200];

3. The representation is typically 7-bit ASCII.

4. Some representations, such as UTF-8, might use several bytes for a single Unicode character. So the length of the array is not necessarily the number of characters.

5. There is no need to worry about byte ordering; the start of the string always has the lowest address in memory.

6. A buffer is also an array of bytes, typically used to hold data subject to I/O. The bytes hold arbitrary binary values, not necessarily printable values.

7. Declaration

(a) char *myBuffer;
(b) char myBuffer[];
(c) char myBuffer[4096];

8. Buffers are not null-terminated; you need a separate variable to remember how much data is in the buffer.

11 Boolean algebra

1. Named after George Boole (1815–1864).

2. A computer’s circuitry uses pieces that accomplish Boolean functions in order to build both combinatorial and sequential circuits.

3. We are familiar with the truth tables for and (in C: &), or (|), not (˜).
   We might not be familiar with exclusive or (xor, ^).

4. When one operates on bytes (or larger chunks such as integers) with Boolean functions, they are applied bitwise. Examples:
5. [Class 7, 1/26/2018] Lab 3

6. [Class 8, 1/29/2018]

7. Instead of interpreting 32 bits as an integer, we can interpret it as a subset of \{0, \ldots, 31\}. Each 1 bit represents a number in that range that is in the set; every 0 bit represents a number that is not in the set. So 1001 represents the set 0, 3. Then:

(a) & is intersection.
(b) | is union.
(c) ^ is symmetric difference.
(d) ~ is complement.

8. One can use the Boolean operators in C and apply them to any integral data type: \texttt{char}, \texttt{short}, \texttt{int}, \texttt{long}, \texttt{long long}.

9. Don’t confuse these operators with logical operators \&\&, ||, and !. In C, 0 is understood to mean false, and any other value is true. The logical operators always return either 0 or 1. They use short-circuit semantics.

12 Shifting operators

1. Left shift: \texttt{x \ll y} Left-shifts bits in \texttt{x} by \texttt{y} positions; new positions on the right are filled with 0. \textbf{Warning:} In C (and Java), if \texttt{y} is equal to or greater than the number of bits \texttt{n} in the type of \texttt{x}, the shift distance is \texttt{y \mod n}. So shifting a 32-bit integer by 34 bits only shifts it \texttt{34 \mod 32 = 2} bits.

2. Right shift: \texttt{x \gg y} Right-shifts bits in \texttt{x} by \texttt{y} positions; new positions on the left are filled with the sign bit (for signed types only). The same warning applies.
13 Compilation and disassembly in Linux


2. Command-line options are by Unix convention marked with `-`.

   - `o filename` put the output of compilation in `filename`
   - `E` Don’t compile; just run the preprocessor. (result goes to standard out)
   - `S` Compile but don’t assemble; result is `filename.s`
   - `c` Compile and assemble, but don’t link; result is `filename.o`
   - `g` Add debugging information to the result.
   - `-On` Turn on optimization level `n` (from 0 to 3, also `s` for size, also `fast`)

14 Tools to inspect compiled code

1. `objdump -d filename`: disassembles `filename`

2. `gdb filename`: runs the debugger on `filename`; can disassemble

3. `nm filename`: shows location and type of identifiers in `filename`

4. `strings filename`: shows all the ASCII strings in `filename`

5. `od filename`: displays `filename` in numeric or character format.


7. `dissy filename`: graphical tool to inspect `filename`. You can install `dissy` by using `apt-get`.

15 Machine basics

1. An architecture, also called an instruction-set architecture (ISA), is the part of a processor design that you need to know to read/write assembler code. It includes the instruction set and the characteristics of registers. Examples: x86 (also called x86_32, IA-32), IPF (also called IA-64: Itanium), x86_64.

2. The microarchitecture describes how the architecture is implemented. It includes the sizes of caches and the frequency at which the machine operates. It is not important for programming in assembler.

3. Components of importance to the assembler programmer
(a) **Program counter** (PC): a register containing the address of the next instruction. Called EIP (x86) or RIP (x86_64)

(b) **Registers**, used for heavily-accessed data, with names specific to the architecture. The set of all registers is sometimes called the **register file**.

(c) **Condition codes** store information about the most recent arithmetic operation, such as “greater than zero”, useful for conditional branch instructions.

(d) **Memory**, addressed by bytes, containing code (also called “text” in Unix), data, and a stack (to support procedures).

16 **Steps in converting C to object code**

1. Say the code is in two files: **p1.c** and **p2.c**

2. To compile: **gcc p1.c p2.c -o p**, which puts the compiled program in a file called **p**.

3. The **gcc** compiler first creates assembler files (stored in `/tmp`, but we can imagine they are called **p1.s** and **p2.s**).

4. It then runs the **as** assembler on those files, creating **p1.o** and **p2.o**.

5. It then runs the **ld** linker to combine those files with libraries (primarily the C library **libc**) to create an executable file **p**. Libraries provide code for **malloc**, **printf**, and others.

6. Some libraries are **dynamically linked** when the program starts to execute, saving space in the executable file and allowing the operating system to share code among processes.

7. Sample code:

   ```
   int sum(int x, int y)
   {
   int t = x+y;
   return t;
   }
   ```

8. Output of objdump -d on compiled (-O0) file:

   ```
   4004ed: 55 push %rbp
   4004ee: 48 89 e5 mov %rsp,%rbp
   ```
9. Interpretation: x is in %edi, then -0x14(%rbp), then %edx; y is in %esi, then -0x18(%rbp), then %eax; t is in %eax, then -4(%rbp), then %eax.

10. Same thing with gdb, using command “disassemble sum”:

```
0x00000000004004ed <+0>: push %rbp
0x00000000004004ee <+1>: mov %rsp,%rbp
0x00000000004004f1 <+4>: mov %edi,-0x14(%rbp)
0x00000000004004f4 <+7>: mov %esi,-0x18(%rbp)
0x00000000004004f7 <+10>: mov -0x18(%rbp),%eax
0x00000000004004fa <+13>: mov -0x14(%rbp),%edx
0x00000000004004fd <+16>: add %edx,%eax
0x00000000004004ff <+18>: mov %eax,-0x4(%rbp)
0x0000000000400502 <+21>: mov -0x4(%rbp),%eax
0x0000000000400505 <+24>: pop %rbp
0x0000000000400506 <+25>: retq
```

11. Same thing with gdb, using command “x/25xb sum”:

```
0x4004ed <sum>:    0x55 0x48 0x89 0xe5 0x89 0x7d 0xec 0x89
0x4004f5 <sum+8>:  0x75 0xe8 0x8b 0x45 0xe8 0x8b 0x55 0xec
0x4004fd <sum+16>: 0x01 0xd0 0x89 0x45 0xfc 0x8b 0x45 0xfc
0x400505 <sum+24>: 0x5d 0xc3
```

12. Compiling with -O1:

```
4004ed: 8d 04 37   lea   (%rdi,%rsi,1),%eax
4004f0: c3        retq
```

13. **Class 9, 1/31/2018**

14. One can even disassemble .EXE files (from Win32 compilations) with objdump
17 Intel/AMD architectures

1. Architecture: what a programmer needs to write assembler/machine code.
   
   (a) Machine code: byte-level programs that the CPU executes
   (b) Assembler code: a text representation of the machine code
   (c) x86 32-bit architecture (since 1985; 2nd edition of our textbook)
   (d) You can compile for 32 bits even on an x86_64 with the \texttt{-m32} flag.
   (e) x86-64: 64-bit architecture (since 2003; 3rd edition of our textbook)
   (f) Others: ARM (used in mobile phones), MIPS, Sparc, ...

2. Registers

   \textbf{non-volatile} means the callee must preserve the value.

   \begin{tabular}{cccccc}
     64-bit & 32-bit & 16-bit & 8-bit & original purpose & C \\
     rax & eax & ax & ah/al & accumulator & return value \\
     rbx & ebx & bx & bh/bl & base & non-volatile \\
     rcx & ecx & cx & ch/cl & counter & 4th parameter \\
     rdx & edx & dx & dh/dl & data & 3rd parameter \\
     rsi & esi & sil & source index & 2nd parameter \\
     rdi & edi & dil & destination index & 1st parameter \\
     rsp & esp & sp & spl & stack pointer & stack pointer \\
     rbp & ebp & bp & bpl & base & base; non-volatile \\
     r8 & r8d & & & general purpose & 5th parameter \\
     r9 & r9d & & & general purpose & 6th parameter \\
     r10 & r9d & & & general purpose \\
     r11 & r9d & & & general purpose \\
     r12 & r9d & & & general purpose & non-volatile \\
     r13 & r9d & & & general purpose & non-volatile \\
     r14 & r9d & & & general purpose & non-volatile \\
     r15 & r15d & & & general purpose & non-volatile \\
     rip & eip & ip & & instruction pointer \\
   \end{tabular}

3. Moving data: \texttt{movq source, dest} The q means “quad”, which means 64 bits. One can also use \texttt{movl}, where l means “long”, which means 32 bits.

4. Operand types
(a) **Immediate**: integer constant, such as $0x400$ or $-533$ The actual constant is represented in 1, 2, or 4 bytes, depending on size; the assembler chooses the right representation. The source may be immediate, but not the destination.

(b) **Register**: any of the 16 integer registers, such as: `%rcx` (although `%rsp` and `%rbp` have special purposes). Can also be the lower 4, 2, or 1 bytes of a register (using names above). Either source or destination or both may be register.

(c) **Memory**: 8 bytes of memory, whose first byte is addressed by any register, such as `(,%rax)` (note the parentheses). Either source or destination, but not both, may be memory.

(d) **Displacement**: 8 bytes of memory whose first byte is addressed by any register plus some constant, such as `8(,%eax)`. Either source or destination, but not both, may be memory or displacement.

5. Example in C: Swap

```c
void swap(long xs[2])
{
    long t0 = xs[0]; // or *xs
    long t1 = xs[1];
    xs[0] = t1;
    xs[1] = t0;
}
```

6. Same thing in assembler

```asm
movq (%rdi),%rax  # t0 = xs[0]
movq 0x8(%rdi),%rdx # t1 = xs[1]
movq %rax,0x8(%rdi) # xs[1] = t0
movq %rdx,(%rdi)  # xs[0] = t1
retq  # return
```

18 More complex memory-addressing modes

1. We saw memory and displacement.

2. This is the most general form: $D(Rb,Ri,S)$

   (a) $D$ is the displacement, in bytes, such as 1, 2, 80.
(b) \( R_b \) is the base register: any of the 8 integer registers.
(c) \( R_i \) is the index register, any register by %esp, and you most likely don’t want to use %ebp, either
(d) \( S \) is a scale, which is any of 1, 2, 4, or 8.

3. The value it references is \( \text{Mem}[\text{Reg}[R_b] + S \cdot \text{Reg}[R_i] + D] \).

4. Example (using 32-bit registers)
   (a) %edx: 0xf000
   (b) %ecx: 0x0100
   (c) 0x8(%edx): 0xf000 + 0x8 = 0xf008
   (d) (%edx, %ecx): 0xf000 + 0x0100 = 0xf100
   (e) (%edx, %ecx, 4): 0xf000 + 4 \cdot 0x0100 = 0xf400
   (f) 0x80(,%edx,2): 2 \cdot 0xf000 + 0x80 = 0x1e080

19 Address computation without referencing

1. [Class 10, 2/2/2018]
2. One can compute an address and save it without actually referencing it.
3. \texttt{lea src, dest}
4. Load Effective Address of \( src \) and put it in \( dest \).
5. Purpose: translate \( p = \&x[i] \)
6. Purpose: compute arithmetic expressions like \( x + k \cdot y \) where \( k \) is 1, 2, 4, or 8.
7. Example: \( x \cdot 12 \)

   1 leal (%eax,%eax,2), %eax  # x = x+2x
   2 sall $2, %eax  # x = x \cdot 4
20 Arithmetic operations

1. Two-operand instructions

<table>
<thead>
<tr>
<th>instruction</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>addq</td>
<td>dest = dest + src</td>
</tr>
<tr>
<td>subq</td>
<td>dest = dest − src</td>
</tr>
<tr>
<td>imulq</td>
<td>dest = dest × src</td>
</tr>
<tr>
<td>salq</td>
<td>dest = dest &lt;&lt; src</td>
</tr>
<tr>
<td>sarq</td>
<td>dest = dest &gt;&gt; src (arithmetic)</td>
</tr>
<tr>
<td>shrq</td>
<td>dest = dest &gt;&gt; src (logical)</td>
</tr>
<tr>
<td>xorq</td>
<td>dest = dest ⊕ src (bitwise)</td>
</tr>
<tr>
<td>andq</td>
<td>dest = dest ∧ src (bitwise)</td>
</tr>
<tr>
<td>orq</td>
<td>dest = dest ∨ src (bitwise)</td>
</tr>
</tbody>
</table>

2. Be careful of parameter order for asymmetric operations.

3. There is no distinction between signed and unsigned integers.

4. One-operand instructions

<table>
<thead>
<tr>
<th>instruction</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>incl</td>
<td>dest = dest + 1</td>
</tr>
<tr>
<td>decl</td>
<td>dest = dest − 1</td>
</tr>
<tr>
<td>negl</td>
<td>dest = − dest</td>
</tr>
<tr>
<td>notl</td>
<td>dest = ¬ dest (bitwise)</td>
</tr>
</tbody>
</table>

5. Example

```c
1    int arith(int x, int y, int z)
2    { 
3        int t1 = x+y;
4        int t2 = z+t1;
5        int t3 = x+4;
6        int t4 = y * 48;
7        int t5 = t3 + t4;
8        int rval = t2 * t5;
9        return rval;
10    }
```

6. Result of compilation (-O1)
1. lea (%rdi, %rsi, 1), %eax  # $a = t1 = x + y
2. add %edx, %eax  # $a = t2 = x + y + z
3. lea (%rsi, %rsi, 2), %edx  # $d = t4 = 3y
4. shl $0x4, %edx  # $d = t4 = 16 * (3y) = 48y
5. lea 0x4(%rdi, %rdx, 1), %ecx  # $c = t5 = x + 48y + 4
6. imul %ecx, %eax  # $a = rval = t5 * t2
7. retq  # return rval

7. Optimization converts multiple expressions to a single statement, and a single expression might require multiple instructions.

8. The compiler generates similar code for \((x + y + z) \times (x + 4 + 48y)\).

9. Example

```c
1 int logical(int x, int y)
2 {
3    int t1 = x^y;
4    int t2 = t1 >> 17;
5    int mask = (1<<13) - 7; // 0x1FF9
6    int rval = t2 & mask;
7    return rval;
8 }
```

10. Result of compilation (-O1)

```asm
1 xor %esi, %edi  # x = t1 = x^y
2 sar $0x11, %edi  # x = t2 = t1 >> 17
3 mov %edi, %eax  # $a = t2 = t1 >> 17
4 and $0x1ff9, %eax  # $a = rval = t2 & mask
5 retq  # return rval
```

21 Control based on condition codes

1. There are four Boolean condition codes

(a) \textbf{CF}: Carry flag
(b) \textbf{ZF}: Zero flag
(c) \textbf{SF}: Sign flag
(d) \textbf{OF}: Overflow flag (for signed integers)
2. Arithmetic operations implicitly set these flags, but the \texttt{lea} instruction does not set them.

3. Example: addition $t = a + b$
   
   (a) sets $\textbf{CF}$ if there is a carry from most significant bit
   (b) sets $\textbf{ZF}$ if the result in $t$ is zero
   (c) sets $\textbf{SF}$ if the top bit of $t$ is set ($t < 0$ if we treat $t$ as a signed integer)
   (d) sets $\textbf{OF}$ if there is a two's complement overflow:
      \[(a > 0 \land b > 0 \land t < 0) \lor (a < 0 \land b < 0 \land t > 0)\]

4. The compare instruction (\texttt{cmp b, a}) also sets the flags; it’s like computing $a - b$ without modifying the destination.
   
   (a) sets $\textbf{CF}$ if there is a carry from most significant bit
   (b) sets $\textbf{ZF}$ if $a = b$
   (c) sets $\textbf{SF}$ if $a - b < 0$
   (d) sets $\textbf{OF}$ if there is a two’s complement overflow:
      \[(a > 0 \land b < 0 \land (a - b) < 0) \lor (a < 0 \land b > 0 \land (a - b) > 0)\]

5. \textit{Class 11, 2/5/2018}

6. The test instruction (\texttt{test b, a}) also sets the flags; it’s like computing $a \& b$ without modifying the destination. Usually, one of the two operands is a mask.
   
   (a) sets $\textbf{ZF}$ if $a \land b = 0$
   (b) sets $\textbf{SF}$ if $a \land b < 0$

7. Many instructions in the \texttt{setXX dest} family test the condition codes and set the destination (a single byte) to 0 or 1 based on the result.

\begin{verbatim}
sete  ZF       Equal/Zero
setne ¬ZF    Not Equal / Not Zero
sets  SF      Negative
setsns ¬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)
\end{verbatim}
8. Many instructions in the \texttt{jXX dest} family jump depending on the condition codes.

\begin{itemize}
  \item \texttt{jmp} \quad \textbf{true} \quad \text{Unconditional}
  \item \texttt{je} \quad \textbf{ZF} \quad \text{Equal/Zero}
  \item \texttt{jne} \quad \neg \textbf{ZF} \quad \text{Not Equal / Not Zero}
  \item \texttt{js} \quad \textbf{SF} \quad \text{Negative}
  \item \texttt{jns} \quad \neg \textbf{SF} \quad \text{Nonnegative}
  \item \texttt{jg} \quad \neg (\textbf{SF} \oplus \textbf{OF}) \land \neg \textbf{ZF} \quad \text{Greater (Signed)}
  \item \texttt{jge} \quad \neg (\textbf{SF} \oplus \textbf{OF}) \quad \text{Greater or Equal (Signed)}
  \item \texttt{jl} \quad (\textbf{SF} \oplus \textbf{OF}) \quad \text{Less (Signed)}
  \item \texttt{jle} \quad (\textbf{SF} \oplus \textbf{OF}) \lor \textbf{ZF} \quad \text{Less or Equal (Signed)}
  \item \texttt{ja} \quad \neg \textbf{CF} \land \neg \textbf{ZF} \quad \text{Above (unsigned)}
  \item \texttt{jb} \quad \textbf{CF} \quad \text{Below (unsigned)}
\end{itemize}

9. Many instructions in the \texttt{cmovXX src, dest} family move data depending on the condition codes.

10. Example

\begin{verbatim}
1 int absdiff(int x, int y)
2 {
3   int result;
4   if (x > y) {
5     result = x-y;
6   } else {
7     result = y-x;
8   }
9   return result;
10 }
\end{verbatim}

\begin{verbatim}
1 mov  %edi,%eax  # $a = x
2 sub  %esi,%eax  # $a = x-y
3 mov  %esi,%edx  # $d = y
4 sub  %edi,%edx  # $d = y-x
5 cmp  %esi,%edi  # flags for x:y
6 cmovle %edx,%eax  # if x \leq y then $a=$d
7 retq
\end{verbatim}

11. C can sometimes use a single \texttt{conditional expression} to handle such cases:

\begin{verbatim}
val = x>y ? x-y : y-x;
\end{verbatim}

\texttt{gcc -O1} generates the same output for this one line as it does for
22 do while loops

1. C code to count how many 1’s in a parameter

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

2. Assembler listing

```assembly
0:   mov $0x0,%eax          # $a = result = 0
5:   mov %edi,%edx          # $d = x
7:   and $0x1,%edx          # $d = x & 0x1
a:   add %edx,%eax          # $a += (x & 0x1)
c:   shr %edi               # x >>= 1
e:   jne 5 <countOnes+0x5>  # if x != 0, goto 5
10:  repz retq              # return result (ignore repz)
```

3. for loops are very similar

4. The compiler can often generate better code by replacing the unconditional jump at the end of the loop with a conditional jump.

23 Procedures

1. In order to handle recursion, languages are compiled to use a stack.

2. Each invocation of a procedure pushes a new frame on the stack.

3. When the procedure returns, the frame is popped and the space it occupied is available for another procedure.

4. The frame contains storage private to this instance of the procedure. From the bottom (at higher addresses):
   
   (a) parameters 7 ... (if necessary; later parameters first)
(b) return address
(c) saved registers (saved by callee)
(d) local variables
(e) temporary locations (that don’t fit in registers)
(f) parameters for the next call (“argument build”), last parameter first