Introduction to Machine-Level Representation of C Programs[1]

Annex to Version 1[2]

Edited by Alberto José Proença
Dep. Informática, Universidade do Minho, Portugal

Contents

1. *Arithmetic and Logical Operations, 1*
2. *Instruction Flow Control, 5*

1. Arithmetic and Logical Operations[3]

Bit-Level Operations in C

One useful feature of C is that it supports bit-wise Boolean operations. In fact, the symbols we use for the Boolean operations are exactly those used by C: `|` for `OR`, `&` for `AND`, `~` for `NOT`, and `^` for `EXCLUSIVE-OR`. These can be applied to any "integral" data type, that is, one declared as type `char` or `int`, with or without qualifiers such as `short`, `long`, or `unsigned`. Here are some example expression evaluations:

C Expression	Binary Expression	Binary Result	C Result
`~0x41`	`~[01000001]`	`[10111110]`	`0xBE`
`~0x00`	`~[00000000]`	`[11111111]`	`0xFF`
`0x69 & 0x55`	`[01101001] & [01010101]`	`[01000001]`	`0x41`
`0x69 | 0x55`	`[01101001] | [01010101]`	`[01111101]`	`0x7D`

[1] Compiled and adapted from the Beta Draft version book of Randal E. Bryant and David R. O'Hallaron *Computer Systems: A Programmer's Perspective* (Prentice Hall, 2002, final version not yet available). These edited notes aim to complement the 1999 authors' handouts, which introduce their students in Carnegie-Mellon University to *Machine-Level Programs on Linux/IA32*; they are complementary teaching notes for laboratory classes in Computer Architecture courses lectured at Dep. Informatics, University of Minho (November 2001).
[2] This annex only contains the additional text that was introduced in Version 1 which generated Version 2.

[3] Editor's note: Most of this material is covered in the author's handouts in *Machine-Level Programs on Linux/IA32*. Some additional comments are here included to better understand the shift operations (taken from Chapter 2 of the book), namely bit-level, logical and shift operations in C. Also, from the same Chapter, some material is presented related to signed *versus* unsigned in C. This section is Section 9 in Version 2.

As our examples show, the best way to determine the effect of a bit-level expression is to expand the hexadecimal arguments to their binary representations, perform the operations in binary, and then convert back to hexadecimal.

One common use of bit-level operations is to implement *masking* operations, where a mask is a bit pattern that indicates a selected set of bits within a word. As an example, the mask `0xFF` (having 1s for the least significant eight bits) indicates the low-order byte of a word. The bit-level operation `x & 0xFF` yields a value consisting of the least significant byte of x, but with all other bytes set to 0. For example, with `x = 0x89ABCDEF`, the expression would yield `0x000000EF`. The expression `~0` will yield a mask of all 1s, regardless of the word size of the machine. Although the same mask can be written `0xFFFFFFFF` for a 32-bit machine, such code is not as portable.

Logical Operations in C

C also provides a set of *logical* operators `||`, `&&`, and `!`, which correspond to the OR, AND, and NOT operations of propositional logic. These can easily be confused with the bit-level operations, but their function is quite different. The logical operations treat any nonzero argument as representing TRUE and argument 0 as representing FALSE. They return either 1 or 0 indicating a result of either TRUE or FALSE, respectively. Here are some example expression evaluations:

Expression	Result
`!0x41`	`0x00`
`!0x00`	`0x01`
`!!0x41`	`0x01`
`0x69 && 0x55`	`0x01`
`0x69 || 0x55`	`0x01`

Observe that a bit-wise operation will have behavior matching that of its logical counterpart only in the special case where the arguments are restricted to be either 0 or 1.

A second important distinction between the logical operators `&&` and `||`, versus their bit-level counterparts `&` and `|` is that the logical operators do not evaluate their second argument if the result of the expression can be determined by evaluating the first argument. Thus, for example, the expression `a && 5/a` will never cause a division by zero, and the expression `p && *p++` will never cause the dereferencing of a null pointer.

Shift Operations in C

C also provides a set of *shift* operations for shifting bit patterns to the left and to the right. For an operand x having bit representation $[x_{n-1}, x_{n-2}, \ldots, x_0]$, the C expression `x << k` yields a value with bit representation $[x_{n-k-1}, x_{n-k-2}, \ldots, x_0, 0, \ldots, 0]$. That is, x is shifted k bits to the left, dropping off the most significant bits and filling the left end with k 0s. The shift amount should be a value between 0 and n-1. Shift operations group from left to right, so `x << j << k` is equivalent to `(x << j) << k`. Be careful about operator precedence: `1<<5 - 1` is evaluated as `1 << (5-1)`, not as `(1<<5) - 1`.

There is a corresponding right shift operation `x >> k`, but it has a slightly subtle behavior. Generally, machines support two forms of right shift: *logical* and *arithmetic*. A logical right shift fills the left end with 0s, giving a result $[0, \ldots, 0, x_{n-1}, x_{n-2}, \ldots, x_k]$. An arithmetic right shift fills the left end with repetitions of the most significant bit, giving a result $[x_{n-1}, \ldots, x_{n-1}, x_{n-1}, x_{n-2}, \ldots, x_k]$. This convention might seem peculiar, but as we will see it is useful for operating on signed integer data.

The C standard does not precisely define which type of right shift should be used. For unsigned data (i.e., integral objects declared with the qualifier `unsigned`), right shifts must be logical. For signed data (the default), either arithmetic or logical shifts may be used. This unfortunately means that any code assuming

one form or the other will potentially encounter portability problems. In practice, however, almost all compiler/machine combinations use arithmetic right shifts for signed data, and many programmers assume this to be the case.

Signed *versus* Unsigned in C

The C standard does not require signed integers to be represented in two's complement form, but nearly all machines do so. To keep code portable, one should not assume any particular range of representable values or how they are represented, beyond the ranges indicated in Table 1 in *Machine-Level Programs on Linux/IA32*.

The C library file `<limits.h>` defines a set of constants delimiting the ranges of the different integer data types for the particular machine on which the compiler is running. For example, it defines constants `INT_MAX`, `INT_MIN`, and `UINT_MAX` describing the ranges of signed and unsigned integers. For a two's complement machine where data type `int` has w bits, these constants correspond to the values of $TMax_w$, $TMin_w$, and $UMax_w$ (see table below).

Quantity	Word Size w			
	8	16	32	64
$UMax_w$	0xFF 255	0xFFFF 65,535	0xFFFFFFFF 4,294,967,295	0xFFFFFFFFFFFFFFFF 18,446,744,073,709,551,615
$TMax_w$	0x7F 127	0x7FFF 32,767	0x7FFFFFFF 2,147,483,647	0x7FFFFFFFFFFFFFFF 9,223,372,036,854,775,807
$TMin_w$	0x80 128	0x8000 32,768	0x80000000 2,147,483,648	0x8000000000000000 9,223,372,036,854,775,808
-1	0xFF	0xFFFF	0xFFFFFFFF	0xFFFFFFFFFFFFFFFF
0	0x00	0x0000	0x00000000	0x0000000000000000

Consider the function $U2T_w(x)$ that takes a number between 0 and $2^w - 1$ and yields a number between -2^{w-1} and $2^{w-1} - 1$, where the two numbers have identical bit representations, except that the argument is unsigned, while the result has a two's complement representation. This behavior is illustrated in Figure 4, below. For small ($<2^{w-1}$) numbers, the conversion from unsigned to signed preserves the numeric value. For large ($>=2^{w-1}$) the number is converted to a negative value.

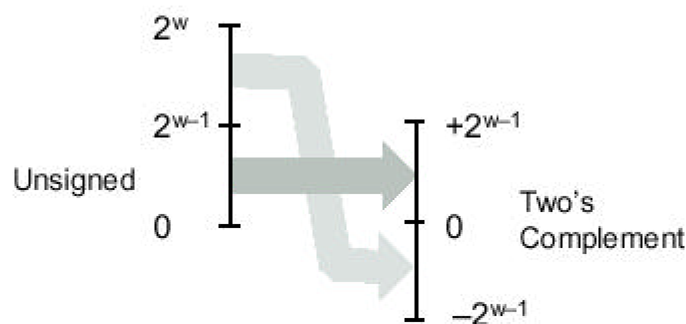

Figure 4: **Conversion from Unsigned to Two's complement** (Function $U2T_w(x)$)

Conversely, the function $T2U_w(x)$ yields the unsigned number having the same bit representation as the two's complement value of x. Figure 5 below illustrates the behavior of this function: when mapping a signed number to its unsigned counterpart, negative numbers are converted to large positive numbers, while nonnegative numbers remain unchanged.

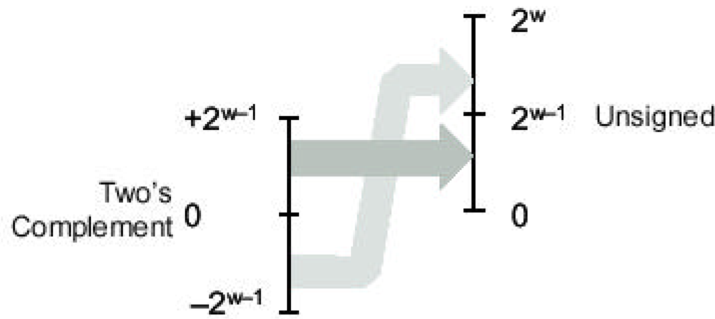

Figure 5: **Conversion from Two's complement to Unsigned** (Function $T2U_w(x)$)

These two functions might seem purely of academic interest, but they actually have great practical importance. They formally define the effect of casting between signed and unsigned values in C. For example, consider executing the following code on a two's complement machine:

```
1  int x = -1;
2  unsigned ux = (unsigned) x;
```

This code will set `ux` to $UMax_w$, where w is the number of bits in data type `int`, since in the table in the previous page we can see that the w bit two's complement representation of `-1` has the same bit representation as $UMax_w$. In general, casting from a signed value `x` to unsigned value `(unsigned) x` is equivalent to applying function $T2U_w$. The cast does not change the bit representation of the argument, just how these bits are interpreted as a number. Similarly, casting from unsigned value `u` to signed value `(int) u` is equivalent to applying function $U2T_w$.

As stated before, C supports both signed and unsigned arithmetic for all of its integer data types. Although the C standard does not specify a particular representation of signed numbers, almost all machines use two's complement. Generally, most numbers are signed by default. For example, when declaring a constant such as `12345` or `0x1A2B`, the value is considered signed. To create an unsigned constant, the character '`U`' or '`u`' must be added as suffix, e.g., `12345U` or `0x1A2Bu`.

C allows conversion between unsigned and signed. The rule is that the underlying bit representation is not changed. Thus, on a two's complement machine, the effect is to apply the function $U2T_w$ when converting from unsigned to signed, and $T2U_w$ when converting from signed to unsigned, where w is the number of bits for the data type.

Conversions can happen due to explicit casting, such as in the code:

```
1  int tx, ty;
2  unsigned ux, uy;
3
4  tx = (int) ux;
5  uy = (unsigned) ty;
```

or implicitly when an expression of one type is assigned to a variable of another, as in the code:

```
1  int tx, ty;
2  unsigned ux, uy;
3
4  tx = ux; /* Cast to signed */
5  uy = ty; /* Cast to unsigned */
```

When printing numeric values with `printf`, the directives `%d`, `%u`, and `%x` should be used to print a number as a signed decimal, an unsigned decimal, and in hexadecimal format, respectively. Note that `printf` does not make use of any type information, and so it is possible to print a value of type `int` with directive `%u` and a value of type `unsigned` with directive `%d`. For example, consider the following code:

```
1  int x = -1;
2  unsigned u = 2147483648; /* 2 to the 31st */
3
4  printf("x = %u = %d\n", x, x);
5  printf("u = %u = %d\n", u, u);
```

When run on a 32-bit machine it prints the following:

```
x = 4294967295 = -1
u = 2147483648 = -2147483648
```

In both cases, `printf` prints the word first as if it represented an unsigned number and second as if it represented a signed number.

Some peculiar behavior arises due to C's handling of expressions containing combinations of signed and unsigned quantities. When an operation is performed where one operand is signed and the other is unsigned, C implicitly casts the signed argument to unsigned and performs the operations assuming the numbers are nonnegative. As we will see, this convention makes little difference for standard arithmetic operations, but it leads to nonintuitive results for relational operators such as < and >.
The figure below shows some sample relational expressions and their resulting evaluations, assuming a 32-bit machine using two's complement representation. The nonintuitive cases are marked by '*'. Consider the comparison `-1 < 0U`. Since the second operand is unsigned, the first one is implicitly cast to unsigned, and hence the expression is equivalent to the comparison `4294967295U < 0U` (recall that $T2U_w(x) = UMax_w$), which of course is false. The other cases can be understood by similar analyses.

Expression	Type	Evaluation
`0 == 0U`	unsigned	1
`-1 < 0`	signed	1
`-1 < 0U`	unsigned	0 *
`2147483647 > -2147483648`	signed	1
`2147483647U > -2147483648`	unsigned	0 *
`2147483647 > (int) 2147483648U`	signed	1 *
`-1 > -2`	signed	1
`(unsigned) -1 > -2`	unsigned	0 *

10. Instruction Flow Control[4]

Encoding Jump Instructions[5]

Under normal execution, instructions follow each other in the order they are listed. A jump instruction can cause the execution to switch to a completely new position in the program. These jump destinations are generally indicated by a *label*.

---

[4] Editor's note: The introductory and relevant material was introduced in the author's handouts in *Machine-Level Programs on Linux/IA32*, namely in Sections 5 (Control) and 6 (Procedures). However, some additional information may be helpful to better understand how jump instructions are encoded and how conditional branches are translated into assembly, which are addressed in this section, while loops and case statements are further analysed in the following two sections.
[5] Editor's note: A short version of this section can be found in Section 5.3 in *Machine-Level Programs on Linux/IA32*. However, several relevant details are missing, and Table 7 (related to `jump` instructions) is incomplete and with a typing mistake in line 2. To improve readability, this section will duplicate the material in those handouts.

Instruction	Jump Condition	Description
`jmp` *Label*	`Always`	Direct Jump
`jmp` **Operand*	`Always`	Indirect Jump
`je` *Label*	`ZF`	Equal / Zero
`jne` *Label*	`~ZF`	Not Equal / Not Zero
`js` *Label*	`SF`	Negative
`jns` *Label*	`~SF`	Nonnegative
`jg` *Label*	`~(SF^OF)&~ZF`	Greater (Signed >)
`jge` *Label*	`~(SF^OF)`	Greater or Equal (Signed >=)
`jl` *Label*	`SF^OF`	Less (Signed <)
`jle` *Label*	`(SF^OF)|ZF`	Less or Equal (Signed <=)
`ja` *Label*	`~CF&~ZF`	Above (Unsigned >)
`jae` *Label*	`~CF`	Above or Equal (Unsigned >=)
`jb` *Label*	`CF`	Below (Unsigned <)
`jbe` *Label*	`CF&~ZF`	Below or Equal (Unsigned <=)

Consider the following assembly code sequence:

```
1     xorl %eax,%eax            Set %eax to 0
2     jmp .L1                   Goto .L1
3     movl (%eax),%edx          Null pointer dereference
4  .L1:
5     popl %edx
```

The instruction `jmp .L1` will cause the program to skip over the `movl` instruction and instead resume execution with the `popl` instruction. In generating the object code file, the assembler determines the addresses of all labeled instructions and encodes the *jump targets* (the addresses of the destination instructions) as part of the jump instructions.

The `jmp` instruction jumps unconditionally. It can be either a *direct* jump, where the jump target is encoded as part of the instruction, or an *indirect* jump, where the jump target is read from a register or a memory location. Direct jumps are written in assembly by giving a label as the jump target, e.g., the label "`.L1`" in the code above. Indirect jumps are written using '`*`' followed by an operand specifier using the same syntax as used for the `movl` instruction. As examples, the instruction

```
jmp *%eax
```

uses the value in register `%eax` as the jump target, while

```
jmp *(%eax)
```

reads the jump target from memory, using the value in `%eax` as the read address.

The other jump instructions either jump or continue executing at the next instruction in the code sequence depending on some combination of the condition codes. Note that the names of these instructions and the conditions under which they jump match those of the `set` instructions. As with the `set` instructions, some of the underlying machine instructions have multiple names. Conditional jumps can only be direct.

Although we will not concern ourselves with the detailed format of object code, understanding how the targets of jump instructions are encoded will become important when we study linking in Chapter 7. In addition, it helps when interpreting the output of a disassembler. In assembly code, jump targets are written using symbolic labels. The assembler, and later the linker, generate the proper encodings of the jump targets. There are several different encodings for jumps, but some of the most commonly used ones are *PC-relative*. That is, they encode the difference between the address of the target instruction and the address of the instruction immediately following the jump. These offsets can be encoded using one, two, or four bytes. A

second encoding method is to give an "absolute" address, using four bytes to directly specify the target. The assembler and linker select the appropriate encodings of the jump destinations.

As an example, the following fragment of assembly code was generated by compiling a file `silly.c`. It contains two jumps: the `jle` instruction on line 1 jumps forward to a higher address, while the `jg` instruction on line 8 jumps back to a lower one.

```
1    jle .L4                      If <=, goto dest2
2    .p2align 4,,7                Aligns next instruction to multiple of 8
3 .L5:                           dest1:
4    movl %edx,%eax
5    sarl $1,%eax
6    subl %eax,%edx
7    testl %edx,%edx
8    jg .L5                       If >, goto dest1
9 .L4:                           dest2:
10   movl %edx,%eax
```

Note that line 2 is a directive to the assembler that causes the address of the following instruction to begin on a multiple of 16, but leaving a maximum of 7 wasted bytes. This directive is intended to allow the processor to make optimal use of the instruction cache memory.

The disassembled version of the "`.o`" format generated by the assembler is as follows:

```
1    8:   7e 11                   jle    1b <silly+0x1b>     Target = dest2
2    a:   8d b6 00 00 00 00       lea    0x0(%esi),%esi      Added nops
3   10:   89 d0                   mov    %edx,%eax         dest1:
4   12:   c1 f8 01                sar    $0x1,%eax
5   15:   29 c2                   sub    %eax,%edx
6   17:   85 d2                   test   %edx,%edx
7   19:   7f f5                   jg     10 <silly+0x10>     Target = dest1
8   1b:   89 d0                   mov    %edx,%eax         dest2:
```

The "`lea 0x0(%esi),%esi`" instruction in line 2 has no real effect. It serves as a 6-byte `nop` so that the next instruction (line 3) has a starting address that is a multiple of 16.

In the annotations generated by the disassembler on the right, the jump targets are indicated explicitly as `0x1b` for instruction 1 and `0x10` for instruction 7. Looking at the byte encodings of the instructions, however, we see that the target of jump instruction 1 is encoded (in the second byte) as `0x11` (decimal 17). Adding this to `0xa` (decimal 10), the address of the following instruction, we get jump target address `0x1b` (decimal 27), the address of instruction 8.

Similarly, the target of jump instruction 7 is encoded as `0xf5` (decimal -11) using a single-byte, two's complement representation. Adding this to `0x1b` (decimal 27), the address of instruction 8, we get `0x10` (decimal 16), the address of instruction 3.

The following shows the disassembled version of the program after linking:

```
1 80483c8:   7e 11                  jle   80483db <silly+0x1b>
2 80483ca:   8d b6 00 00 00 00      lea   0x0(%esi),%esi
3 80483d0:   89 d0                  mov   %edx,%eax
4 80483d2:   c1 f8 01               sar   $0x1,%eax
5 80483d5:   29 c2                  sub   %eax,%edx
6 80483d7:   85 d2                  test  %edx,%edx
7 80483d9:   7f f5                  jg    80483d0 <silly+0x10>
8 80483db:   89 d0                  mov   %edx,%eax
```

The instructions have been relocated to different addresses, but the encodings of the jump targets in lines 1 and 7 remain unchanged. By using a PC-relative encoding of the jump targets, the instructions can be

compactly encoded (requiring just two bytes), and the object code can be shifted to different positions in memory without alteration.

To implement the control constructs of C, the compiler must use the different types of jump instructions we have just seen. We will go through the most common constructs, starting from simple conditional branches, and then considering loops and switch statements.