# More on NASM and Assembly Language

W. H. K. Bester

Computer Science 252, University of Stellenbosch, 2008

**Assembly language** You try to shoot yourself in the foot, only to discover you first have to invent the gun, the bullet, and your foot. If you know what you are doing, in only 7 bytes, you blow off your entire leg using a mere 2 CPU cycles. Else, when you pull the trigger, the gun beeps and crashes the system. Then the system administrator arrives and shoots you in the foot. After a moment of contemplation, he shoots himself in the foot and then hops rabidly around the lab, shooting everyone else in the head.

**C** You shoot yourself in the foot.

**C++** You accidentally create a dozen instances of yourself and shoot them all in the foot. Providing emergency medical assistance is impossible since you can't tell which are bitwise copies and which are just pointing at others and saying, "That's me, over there."

**C#** You stab yourself in the foot with something sharp.

**Java** You locate the `Gun` class, but discover that the `Bullet` class is abstract, so you extend it and write the missing part of the implementation. Then you implement the *`ShootAble`* interface for your foot, and recompile the `Foot` class. The interface lets the bullet call the `doDamage()` method on the `Foot`, so the `Foot` can damage itself in the most effective way. Now you run the program, and call the `doShoot()` method on the instance of the `Gun` class. First the `Gun` creates an instance of `Bullet`, which calls the `doFire()` method on the `Gun`. The `Gun` calls the `hit(Bullet)` method on the `Foot`, and the instance of `Bullet` is passed to the `Foot`. But this causes an `IllegalHitByBullet` exception to be thrown, and you die.

**Pascal** The compiler won't let you shoot yourself in the foot.

**Python** You try to shoot yourself in the foot, only to realise that there's no need, since Guido thoughtfully shot you in the foot years ago.

**Ruby** If you can find enough documentation, your foot is ready to be shot in roughly five minutes, but you just can't find anywhere to shoot it.

**SQL** You type: SELECT @ammo:=bullet FROM gun WHERE trigger = 'PULLED'; INSERT INTO leg (foot) VALUES (@ammo);—and then you realise something didn't match a field type, and your foot was converted into a watermelon.

**VisualBasic** Shoot yourself in the foot with a water pistol. On Vista, continue until your entire lower body is waterlogged.

## Contents

```
%define SYS_exit  1
%define SYS_write 4
%define STDOUT    1

; %% Read-only data %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    section .rodata

hello:     db "Hello,␣World!", 10
hello_len: equ $-hello

; %% Code section %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    section .text
    global _start

_start:
    ; gdb doesn't like to stop at the entry point address, so we
    ; put a nop here for convenience
    nop

main:
    mov  eax, SYS_write
    mov  ebx, STDOUT
    mov  ecx, hello
    mov  edx, hello_len
    int  80h

    xor  ebx, ebx
    mov  eax, SYS_exit
    int  80h
```

Figure 1: Assembly source to display "Hello, World!" on screen.

## 1 "Hello World!" in Linux Assembly Language

Consider the assembly code in figure 1. As a matter of fact, don't only consider it, but copy the code into your favourite editor `vim`, and save it as `hello.asm`. Then do the following:

1. Assemble the program with NASM.

   ```
   [whkbester@knuth rw252]$ nasm -f elf hello.asm
   ```

   This creates the unlinked object file `hello.o`.

2. Next, using `ld`, link the object file produced by NASM into an executable.

   ```
   [whkbester@knuth rw252]$ ld -o hello hello.o
   ```

   This produces the executable file `hello`.

3. Run the executable thus:

   ```
   [whkbester@knuth rw252]$ ./hello
   Hello, World!
   ```

Now, let us examine the assembly source in figure 1 line by line. A more detailed discussion follows in the next section.

**Lines 1 to 3** defines three single-line macros. The NASM preprocessor directive `%define` functions similarly to the C preprocessor directive `#define`. For example, `%define SYS_write 4` will expand `SYS_write` in line 21 to `4`.

**Line 6** announces that the section for initialised read-only data follows.

**Line 8** defines a string constant that is labelled `hello` and that ends with the LINEFEED character, which is given as decimal 10. Since we don't involve C, we don't need to follow the C string convention, and ending the string with null character is unnecessary.[1]

**Line 9** defines the numeric constant labelled `hello_len` to be the length of the string constant defined on the previous line.

**Line 12** announces that the section for code follows.

**Line 13** declares the global symbol `_start`, which labels the program entry point where execution will begin when the assembled and linked code is run. If `_start` is not specified, the linker `ld` gives a warning.

**Lines 21 to 24** set up the required registers for the `write` system call.

**Lines 25** invokes the system call interrupt so that the `write` is executed. For now, we ignore the return value in register `EAX`.

**Lines 27 to 29** set up the registers for and execute the `exit` system call.

## 2 NASM in more detail

### 2.1 Command-Line Options

For all the NASM command-line options, refer to chapter 2 of the NASM manual. The options of particular interest to us are:

`-f` specifies the output file format.[2] In our setup, we invariably follow this switch with `elf` to specify the Executable and Linking Format (ELF). NASM is capable of generating output in other formats, including Microsoft OMF and Win32 object files, as well as the older Linux `a.out` format. For more information on ELF, type the following at the console:

```
[whkbester@knuth rw252]$ man 5 elf
```

[1]*Null*, derived from the Latin word *nullus*, means "nothing". For example, a null pointer intentionally does not point to an object, the null string is the empty string, and the null character has neither value nor control meaning, but may be used to delimit strings. In C, both the null pointer symbol `NULL` and the null character `\0` may be written as a literal zero `0`; doing so, however, is considered bad style.

[2]A file containing code that has already been assembled or compiled is called an *object file* (as opposed to *source file*). We tend to use "output file format" and "object file format" interchangeably.

```
 1                                     %define SYS_exit  1
 2                                     %define SYS_write 4
 3                                     %define STDOUT    1
 4
 6                                         section .rodata
 8 00000000 48656C6C6F2C20576F-        hello:     db "Hello,␣World!", 10
 9 00000009 726C64210A
10                                     hello_len: equ $-hello
11
13                                         section .text
15                                         global _start
16
17                                     _start:
20 00000000 90                             nop
21
22                                     main:
23 00000001 B9[00000000]                   mov  ecx, hello
24 00000006 BA0E000000                     mov  edx, hello_len
25 0000000B BB01000000                     mov  ebx, STDOUT
26 00000010 B804000000                     mov  eax, SYS_write
27 00000015 CD80                           int  80h
28
29 00000017 31DB                           xor  ebx, ebx
30 00000019 B801000000                     mov  eax, SYS_exit
31 0000001E CD80                           int  80h
```

Figure 2: The listing file produced by NASM for `hello.asm`.

**-l** (followed by a file name) lets NASM generate a listing file, which contains address offsets and the generated code on the left, and the actual source code, with multi-line macros expanded, on the right. The following, for example, produces a listing file for `hello.asm`, called `hello.lst` and given in figure 2:

```
[whkbester@knuth rw252]$ nasm -f elf -l hello.lst hello.asm
```

Note that, for the sake of brevity, some empty and all comment lines have been removed in figure 2 although they appear in the actual file.

**-v** displays version information.

## 2.2 The Layout of a Source Line

Each NASM source line contains some combination of the following four fields:

```
label:    instruction  operands      ; comment
```

Most of these fields are optional; refer to chapter 2 of the NASM manual. The presence or absence of any combination of a label, an instruction and comment is allowed. The operand field is either required or forbidden by the presence and nature of the instruction field. The applicable instructions may be prefaced by REP, REPE, etc. in the usual way.

Valid labels start with a letter, "`.`", "`_`" or "`?`", followed by any of these characters, numbers, "`$`", "`#`", "`@`", or "`~`". Labels beginning with a single period have special meaning. They are treated as local labels, which means that are associated with the previous non-local label. This implies that they can be reused. For example, we can use the local label `.loop` (once) in each of the scopes of non-local labels.

Labels should not be confused with the stronger concept of a variable. A variable may take on values, but a label only marks a position. In high-level languages such as C and Java, variables are supported in the compiler by a symbol table that associates type, scope, size, visibility, and other information with a variable identifier. Labels, on the other hand, are simply devices of convenience, allowing the assembly programmer to use identifiers to refer to positions in the code, thereby leaving the calculation of jump offsets and subroutine entry points to the assembler.

Also, there are no primitive data types *per se* in assembly language. The type of the data follows implicitly from its context, i.e., which instruction is used to operate on it, whether it is used as a memory address that points to other data, etc. For example, the use of `div` or `idiv` specifies whether the operands are to be treated as unsigned or signed integers, respectively.

## 2.3 Pseudo-Instructions

Pseudo-instructions, although they are not real Intel instructions, may used in the instruction field of a source line. The pseudo-instructions supported by NASM are `db`, `dw`, `dd`, `dq`, `resb`, `resw`, `resd`, `resq`, `rest`, the `incbin` and `equ` commands, and the `times` prefix.

The `dx` instructions are used to declare initialised data in the output file (`x` is `b` for byte, `w` for word, etc.), while the `resx` instructions are used in the BSS section to declare uninitialised storage space. `incbin` includes a file verbatim into the output file, and `equ` defines a symbol to be a given constant value. `times` causes an instruction to be assembled multiple times. Refer to chapter 3 of the NASM manual for examples and a more detailed discussion.

## 2.4 Expressions

Expressions in NASM have a similar syntax to those of C. Regarding integers, NASM guarantees the same as C: There will always be *at least* 32 bits per integer. In increasing order of precedence, the arithmetic operators provided by NASM are:

1. “`|`”, the bitwise OR operator;

2. “`^`”, the bitwise XOR operator;

3. “`&`”, the bitwise AND operator;

4. “`<<`” and “`>>`”, the left and right bit shifts—in NASM right bit shifts are always unsigned, i.e., the bit positions that are shifted in on the left are zero-filled;

5. “`+`” and “`-`”, the ordinary addition and subtraction operators;

6. “`*`”, “`/`”, “`//`”, “`%`”, and “`%%`”, the mutiplication, division, and modulo operators, where a single division or modulo operator is unsigned, and a double one is signed; and

```
%define SYS_read  3
%define SYS_write 4
%define SYS_open  5
%define SYS_close 6
%define SYS_creat 8
```

Figure 3: Using macros to name magic numbers for Linux system calls.

7. the unary operators "-" (which negates its operand), "+" (which does nothing and is only for symmetry with the unary minus), "~" (which computes the one's complement of its operand), and "SEG" (which gives the segment address of its operand).

Additionally, NASM provides two special tokens for use in expression: `$` evaluates to the assembly position at the beginning of the line containing the expression, and `$$` evaluates to the beginning of the current section. For example, the expression `$-$$` calculates how far a line is into its section.

Another example of a NASM expression is line 9 of `hello.asm` in figure 1: The label `hello_len` is defined to have the value of the operand after the pseudo-instruction `equ`. The operand `$-hello` evaluates to the difference of the assembly positions at the current line (i.e., the line beginning with the label `hello_len`) and the line beginning with the label `hello`, which is the length of the string labelled `hello`. Therefore, during NASM preprocessing, the symbol `hello_len` in line 24 is replaced with the number 14. So, the generated code for line 24 of figure 2 makes sense, because `EB` is the opcode for an immediate `mov` to `EDX`, and `0E000000` is the little-endian hexadecimal representation of 14.

## 2.5 Macros

Within the context of preprocessing, for assembly as well as C, a *macro* is a rule that specifies how a sequence of input tokens is mapped to output characters. Such a mapping process is known as *macro expansion*. In the preprocessors for assembly and C, macro expansion works by simple textual search-and-replace.

You should, by now, be familiar with using macros to define so-called *magic numbers* in C. Magic numbers are the constants, array sizes, character positions, conversion factors, and other literal values in programs. As a rule of thumb, any number other than 0 or 1 is likely to be a magic number. Using a descriptive string rather than a number leads to code that is more readable, and that is also easier to modify and debug.

In a program that uses some system calls frequently, it makes sense to name the system call function numbers as was done in figure 3. Then, instead of writing "`mov eax, 3`", we write "`mov eax, SYS_read`", which is not only more readable, but as far as mistakes with function numbers go, has a single point of failure, namely where the magic number itself was defined.

In both assembly language and C, we may also use function macros, i.e., macros that expand not only to a number, but to a piece of code. For example, make the following C macro definition:

```
#define BOOLEAN_VALUE(x) ((x) ? "TRUE" : "FALSE")
```

Then it is legal to write:

```
    int b = 0;
    printf("b␣is␣%s\n", BOOLEAN_VALUE(b));
```

Remember that a macro works by a textual search-and-replace. So, the macro defined above is not a true function, and after the preprocessor has run, the previous code will be:

```
    int b = 0;
    printf("b␣is␣%s\n", ((x) ? "TRUE" : "FALSE"));
```

In C, function macros should be used sparingly. When C was first defined—in a time of slow machines with expensive function calls—function macros were a handy way of avoiding the cost of a function call for short computations that are executed frequently. Today we do not suffer from such limited resources, so function macros provide little benefit.

An example of warranted use is conditional compilation of debug information reporting routines. Consider the following definition:

```
#ifdef DEBUG_OUTPUT
    void DebugInfo(const char *fmt, ...);
    #define DINFO(x, ...) DebugInfo(x, ## __VA_ARGS__)
#else
    #define DINFO(x, ...)
#endif
```

If the DEBUG_OUTPUT macro switch is flipped on for GCC invocation, the function prototype is included in the preprocessed file, and the DINFO(x, ...) macro definition is made; if the switch was not given, DINFO(x, ...) will be replaced with the empty string in the source file. To flip the switch, use the following GCC invocation for the file app.c:

```
[whkbester@knuth rw252]$ gcc -DDEBUG_OUTPUT -o app app.c
```

Similarly, NASM macros may expand to operands and instruction snippets. The NASM manual gives the following example:

```
%define ctrl       0x1F &
%define param(a,b) ((a)+(a)*(b))

    mov  byte [param(2,ebx)], ctrl 'D'
```

The last line will expand to:

```
    mov  byte [(2)+(2)*(ebx)], 0x1F & 'D'
```

NASM also allows multi-line macros. Consider the following code snippet, which expands to a function prologue that follows the C calling convention and gets the stack space to be reserved for local variables as a macro argument.

```
%macro  prologue 1

        push  ebp
        mov   ebp, esp
        sub   esp, %1

%endmacro
```

Macros can be overloaded, so the following code snippet defines a macro (with no arguments) that allocates no local stack space, and that can be included in the same source file as the previous code snippet:

```
%macro  prologue 0

        push  ebp
        mov   ebp, esp

%endmacro
```

## 2.6 Sections

A section directive changes which section of the output the code will be assembled into. The Unix object formats, including ELF, all support the standard section names `.text`, `.data`, and `.bss`, for code, and initialised data, respectively. Additionally, ELF supports `.rodata` for read-only initialised data. In more detail:

**.text** contains the instructions that is executed. It is the only section that may be executed, and is treated as read-only data once the program is run.

**.data** contains initialised data. It cannot be executed, but may be read from and written to when the program is run. The `dx` pseudo-instructions, for example, are used to declare initialised data:

```
    section .data
num:    dd 0x12345678          ; 0x78 0x56 0x34 0x12
str:    db 'Hello!', 10, 0     ; the string "Hello" with linefeed
                               ; and terminator
```

**.bss** contains uninitialised data. It cannot be executed, but may be read from and written to when the program is run. The `resx` pseudo-instructions, for example, are used to reserve uninitialised storage space:

```
    section .bss
buffer:     resb   64          ; reserve 64 bytes
wordvar:    resw   1           ; reserve a word
```

**.rodata** contains initialised read-only data. It may not be written to. This section is illustrated in `hello.asm` in figure 1.

# 3 Linking with GCC

Thus far, we have only looked at linking with `ld`. However, from previous experience with C source code divided into separate modules, we know that GCC is quite willing to link object code files into an executable. It stands to reason that we would try to "compile" the object code file `hello.o`, assembled from `hello.asm` by NASM, as follows:

```
[whkbester@knuth rw252]$ gcc -o hello hello.o
```

However, when we run GCC like this, the following error message is displayed:

```
hello.o: In function '_start':
hello.asm:(.text+0x0): multiple definition of '_start'
/usr/lib/gcc/i386-redhat-linux/4.1.2/../../../crt1.o:(.text+0x0):
```

```
%define SYS_write 4
%define STDOUT    1

    section .rodata
hello:      db "Hello,␣World!", 10
hello_len: equ $-hello

    section .text
    global main
main:
    push ebp
    mov  ebp, esp
    push ebx

    mov  eax, SYS_write
    mov  ebx, STDOUT
    mov  ecx, hello
    mov  edx, hello_len
    int  80h

    pop  ebx
    mov  esp, ebp
    pop  ebp
    ret
```

Figure 4: `hello.asm` rewritten for use with GCC.

```
 first defined here
/usr/lib/gcc/i386-redhat-linux/4.1.2/../../../crt1.o: In
 function ‘_start’:
(.text+0x18): undefined reference to ‘main’
collect2: ld returned 1 exit status
```

This error is the result of C mandating a function called `main()`, which is the program entry point for a program compiled from C source. As our source file in figure 1 currently stands, there is no global symbol `main`, hence the complaint by `ld`.

We already have a label `main`, so if we add a global declaration for this label, GCC should be happy. (Remember to reassemble `hello.asm` with NASM.) Unfortunately, GCC complains again:

```
hello.o: In function ‘_start’:
hello.asm:(.text+0x0): multiple definition of ‘_start’
/usr/lib/gcc/i386-redhat-linux/4.1.2/../../../crt1.o:(.text+0x0):
 first defined here
collect2: ld returned 1 exit status
```

To get things working, we need to remove the global declaration of the symbol `_start`. Also, since GCC treats `main` as a function, we should follow the C function calling convention. This yields the code in figure 4. Note that, just as if we were writing any other function in assembler for use with C source code, as the callee we must preserve the registers `EBX`, `EDI`, and `ESI` if we plan to use them. We used `EBX` for the system call `write`, so it is preserved on the stack.

If we use `ld` instead of GCC to link, we must point `ld` to all the relevant libraries to link against. Therefore, if we want to use functions such as `printf()`

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    printf("Hello,␣world!\n");
}
```

Figure 5: C source code to display “Hello, world!”.

```
.file   "hello.c"
    .section    .rodata
.LC0:
    .string "Hello,␣world!"
    .text
.globl main
    .type   main, @function
main:
    leal    4(%esp), %ecx
    andl    $-16, %esp
    pushl   -4(%ecx)
    pushl   %ebp
    movl    %esp, %ebp
    pushl   %ecx
    subl    $4, %esp
    movl    $.LC0, (%esp)
    call    puts
    addl    $4, %esp
    popl    %ecx
    popl    %ebp
    leal    -4(%ecx), %esp
    ret
    .size   main, .-main
    .ident  "GCC:␣(GNU)␣4.1.2␣20070925␣(Red␣Hat␣4.1.2-33)"
    .section    .note.GNU-stack,"",@progbits
```

Figure 6: The assembler listing for figure 5 produced by GCC.

from the standard C libraries, it is easier to use GCC itself for linking.

GCC will produce assembler output that may be assembled with the GNU assembler GAS. If we compile the program `hello.c` in figure 5 with the `S` switch, GCC will produce `hello.s`, given in figure 6:

```
[whkbester@knuth rw252]$ gcc -S hello.c
```

By default, GAS uses the AT&T-style syntax, where the source operand appears before the destination operand. So, “`movl %esp, %ebp`” is interpreted the same as the NASM code “`mov ebp, esp`”. (NASM uses the Intel syntax where the destination operand is before the source operand.) Note, also, that the operand size is specified as a suffix to the instruction name. So, the suffix `b` specifies byte-sized operands, and `w` and `l` specifies word-sized (16-bit) and long (32-bit) operands, respectively.

Until recently, GAS only supported the AT&T syntax. This implies that inline assembly—where assembly is included directly into a C source file—is written in the AT&T syntax. GAS now also supports the Intel syntax through marking

| x | NOT y |
|---|---|
| 0 | 1 |
| 1 | 0 |

| x | y | x AND y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

| x | y | x OR y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

| x | y | x XOR y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Figure 7: Truth tables for NOT, AND, OR, and XOR.

| $x$ | $y$ | $\neg x$ | $\neg y$ | $x \wedge \neg y$ | $\neg x \wedge y$ | $(x \wedge \neg y) \vee (\neg x \wedge y)$ |
|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 0 | 0 | 1 |

Figure 8: The derivation of XOR from AND, OR, and NOT.

the assembly source with a special directive. It is, however, a new and poorly documented feature, so we stick with the AT&T syntax for inline assembly.

## 4 Logic and Boolean Algebra

We now digress briefly into the areas of logic and boolean algebra. It is tempting here to delve into digital logic, a subject right on the edge between computer science and electronic engineering. Suffice it, however, merely to note some of the basics.

The word *digital* refers to information being represented as discrete values. So a decimal digital computer, for example, would need a way of representing 10 values, the numbers 0 to 9. In practice, because of physical restrictions on components, it is easier to deal with binary digital logic. In these systems, the numbers 0 and 1 are represented by a low and a high voltage interval.

In binary logic computers, the binary values are manipulated by logic circuits called *gates*. These gates correspond to the operators of *boolean algebra*, which is a binary symbolic logic, and is named after its inventor George Boole.[3]

### 4.1 The Basic Logic Operations

Only three boolean functions are necessary to run the whole gamut of boolean algebra involving several variables. These are the familiar operators AND, OR, and NOT, which are defined by the first three truth tables in figure 7. The derivation of XOR ("exclusive or", the last table in figure 7) is given in figure 8 to illustrate how the basic operations are used in constructing another boolean operator.

If we interpret the 1 and 0 in figure 7 as TRUE and FALSE, respectively, then AND, OR, and NOT correspond to the logical connectives *conjunction* ("$\wedge$"), *disjunction* ("$\vee$"), and *negation* ("$\neg$"), respectively, from formal mathematical

[3]It seems that the biggest honour mathematicians can bestow on a fellow practitioner of the dark arts, is to create an eponymous adjective sans capital letter. Hence, "abelian" in stead of "Abelian", "boolean" in stead of "Boolean", and so on.

$$
\begin{array}{rcr}
1A = A & 0 + A = A & \text{(Identity law)} \\
0A = 0 & 1 + A = 1 & \text{(Null law)} \\
AA = A & A + A = A & \text{(Idempotent law)} \\
A\overline{A} = 0 & A + \overline{A} = 1 & \text{(Inverse law)} \\
AB = BA & A + B = B + A & \text{(Commutative law)} \\
(AB)C = A(BC) & (A + B) + C = A + (B + C) & \text{(Associative law)} \\
A + BC = (A + B)(A + C) & A(B + C) = AB + AC & \text{(Distributive law)} \\
A(A + B) = A & A + AB = A & \text{(Absorption law)} \\
\overline{AB} = \overline{A} + \overline{B} & \overline{A + B} = \overline{A}\,\overline{B} & \text{(De Morgan's law)}
\end{array}
$$

Figure 9: Some identities in boolean algebra.

| | | | |
|---|---|---|---|
| & | bitwise AND | \| | bitwise inclusive OR |
| ^ | bitwise exclusive OR | ~ | one's complement (unary) |

Figure 10: C operators for logical bitwise operations.

logic. By convention, in boolean algebra we use implied multiplication or a dot to mean AND, a plus to mean OR, and an over-bar to mean NOT.

Some important identities of boolean algebra are given in figure 9. Be careful where the notation seems to contradict "normal" mathematical interpretation, for example, in the distributive law where OR is said to distribute over AND. When in doubt, draw up a truth table!

At this point, you may think that this all is a lot of academic hogwash. However, De Morgan's laws, for example, may be especially helpful when reasoning about the guard statements in loops, particularly if you are converting the structured loops of a high-level programming language to the goto paradigm of assembler, and at the same time, trying to use as few logic instructions as possible.

## 4.2 Bit vectors

**Note:** In this subsection, we use the C notation for bitwise operations, given in figure 10.

It sometimes necessary to represent binary states—i.e., on–off, busy–available, true–false, and yes–no—succinctly in code. To this end, bit vectors may be used. In assembly language, a bit vector is the just the value of a register or memory area, except that we interpret this value as a string of 1s and 0s, each string position storing the on–off or busy–available state of some device, the yes–no or true–false result of boolean computation, etc. In other words, we don't interpret the register or memory value as an integer, a floating-point number, a character, etc.

We cannot, however, address bits directly. So, how do we use such a bit vector? An example will best illustrate what we do. Say we have 8 I/O devices, numbered 0 to 7, attached to a computer. We may now use a byte value as a bit vector to represent the busy or available states of the devices, where 1 indicates that a device is busy, and 0 that it is not. Then the bit vector 00101100 means that devices 0, 1, 4, 6, and 7 are available, and that devices 2, 3, and 5 are busy.

To work with a single bit, we use the concept of a mask. The mask for bit b is simply (the binary representation of) $2^b$. The mask for bit 0 is therefore $2^0$ = 00000001, for bit 1 it is $2^1$ = 00000010, for bit 2 it is $2^2$ = 00000100, and so on.

To see whether a device b is busy, we compute the bitwise AND of the device vector with the mask for the bit b, and then zero-fill shift the result b bits to the right. For example, to check the state of device 2, and given the device vector 00101100, we compute

```
          00101100
&         00000100
------------------
          00000100
>> 2      00000001
------------------
```

to see that device 2 is busy. Note that, since the mask is a power of 2, we may create the mask itself by bitshifts to the left.

Similarly, we may check the state of device 7 by

```
          00101100
&         10000000
------------------
          00000000
>> 7      00000000
------------------
```

to see that it is available. Intuitively, since k & 1 = k and k & 0 = 0, the effect of the AND-operation is to "mask off" the other device bits, leaving us only with the bit of interest.

To set bit b to 1, we compute the bitwise OR of the given vector with the mask for bit b. Since k | 1 = 1 and k | 0 = k, the effect of the OR-operation is to "mask on" the bit, regardless of what its value was previously, and leaving the other bits unchanged. If bit b were 1, it stays 1, and if it were 0, it becomes 1. The following two computations illustrate the point:

```
     00101100          00101100
|    00010000     |    00000100
-------------     -------------
     00111100          00101100
-------------     -------------
```

To set bit b to 0, we compute the bitwise AND of the given vector with the bitwise complement of the mask for bit b. This sets bit b to 0, regardless of its previous value, but leaves the other bits unchanged. To set bits 4 and 2 to 0, we do, respectively:

```
     00101100          00101100
&    11101111     &    11111011
-------------     -------------
     00101100          00101000
-------------     -------------
```

To flip bit b, we compute the XOR of the given vector with the mask for bit b. Since k ^ 1 = ~k and k ^ 0 = k, the effect is to keep every bit in the original vector aligned with a 0 the same, but to flip the bit aligned with the 1 in the mask. This is illustrated in the following two bitflips, for bits 4 and 2, respectively:

```
     00101100          00101100
^    00010000     ^    00000100
-------------     -------------
     00111100          00101000
-------------     -------------
```