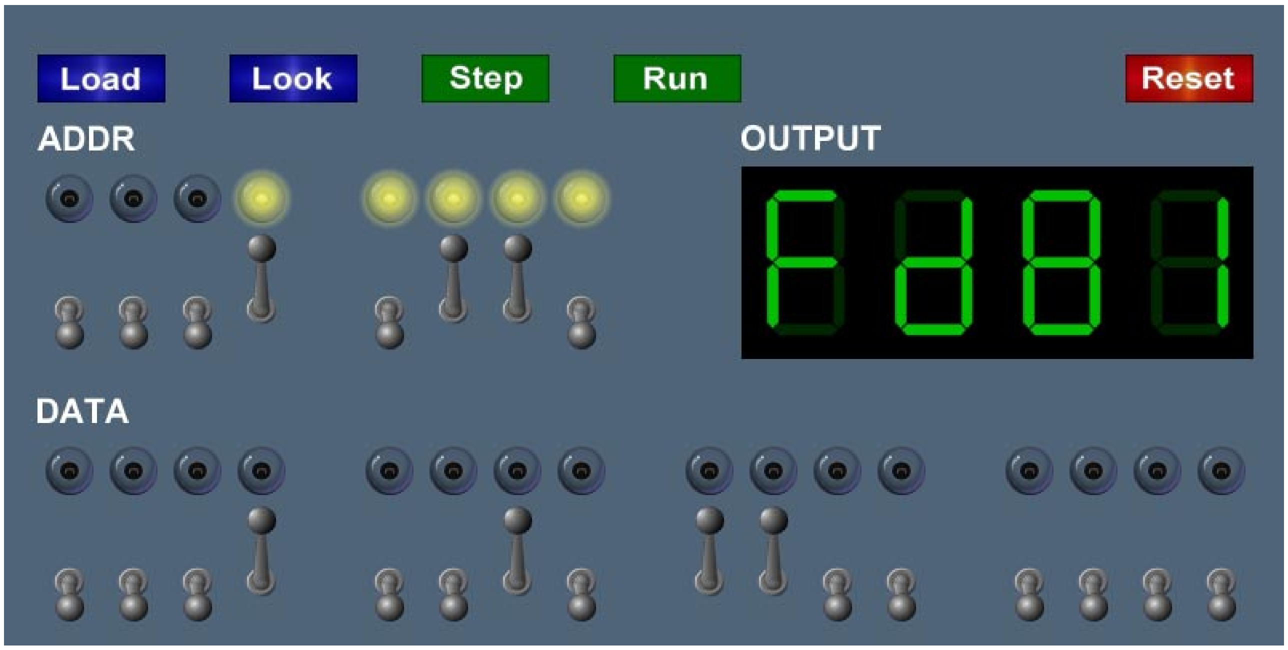
The X-TOY Machine*

February 6, 2002

Abstract

The following document supplements the COS 126 course material on X-TOY, including Lectures A1 and A2. X-TOY is an imaginary machine (created at Princeton) that is very similar to ancient computers. We study it today because it shares the essential characteristics of modern day microprocessors. Also, it demonstrates that simple computational models can perform useful and nontrivial calculations. In this document we describe how to use and program the X-TOY machine. The companion document *Building the X-TOY Machine* describes how to build such a machine in hardware.



*Copyright © 2001, Kevin Wayne. Based on lecture notes by Bob Sedgewick.

Contents

1 Inside the X-TOY Machine

The X-TOY machine consists of an arithmetic logic unit, memory, registers, a program counter, switches, lights, and a few buttons (LOAD, GO, STEP, LOOK). We now describe the function of each of these components. Then, we will describe how the use these components to write X-TOY machine language programs.

1.1 Word size

The X-TOY machine has two types of storage: main memory and registers. Each entity stores one *word* of information. On the X-TOY machine, a word is a sequence of 16 bits. Typically, we interpret these 16 bits as a hexadecimal integer in the range `0000 - FFFF`. Using *two's complement notation*, we can also interpret it as a decimal integer in the range $-32,768$ to $+32,767$. See Appendix A for a refresher on number representations and two's complement integers.

1.2 Main memory

The X-TOY machine has 256 words of *main memory*. Each memory location is labeled with a unique *memory address* – by convention, we use the 256 hexadecimal integers in the range `00 - FF`. Think of a memory location as a mailbox, and a memory address as a postal address. Main memory is used to store instructions and data.

1.3 Registers

The X-TOY machine has 16 *registers*, indexed from `0 - F`. Registers are much like main memory: each register stores one 16-bit word. However, registers provide a faster form of storage than main memory. Registers are used as scratch space during computation and play the role of variables in the X-TOY language. Register 0 is a special register whose output value is always 0.

1.4 Program counter

The *program counter* (`pc`) is an extra register that keeps track of the next instruction to be executed. It stores 8 bits, corresponding to a hexadecimal integer in the range `00 - FF`. This integer stores the memory address of the next instruction to execute.

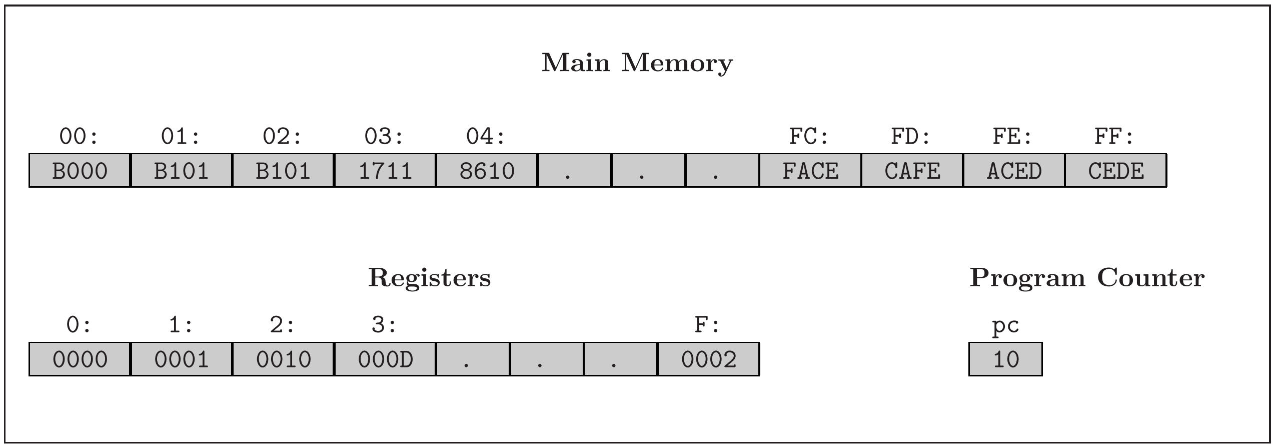

Main Memory
00: 01: 02: 03: 04: FC: FD: FE: FF:
B000 B101 B101 1711 8610 . . . FACE CAFE ACED CEDE
Registers
0: 1: 2: 3: F:
0000 0001 0010 000D . . . 0002
Program Counter
pc
10

Figure 1: The contents of memory, registers, and the program counter characterize the state of the X-TOY machine, and completely determine what the machine will do.

1.5 Input

The *switches* and LOAD button are used to enter instructions and data into the machine. The switches behave just like ordinary light switches: they are either on or off. There are 8 *memory address switches* to select one of the $2^8 = 256$ possible memory addresses. There are also 16 *data switches* to select a 16 bit integer to load into the corresponding memory location. To enter data into memory, you set the appropriate memory and data switches, and then press the LOAD button. This tedious process is repeated for each memory location. All registers and memory locations are initially `0000`; the `pc` is initially `10`.

1.6 Output

The *memory address switches*, *lights* and LOOK button are used to display the address and contents of one word of main memory, as the program is being executed. To select which word of main memory to view, you set the 8 memory address switches and press the LOOK button. Now, the 8 address lights display the chosen memory address (which at the moment is the same as the address switches). The 16 data lights display the contents of that memory location. Old programmers could often know what part of the program was being executed by staring at the pattern of memory lights.

1.7 Running the Machine

To execute an X-TOY program, you first enter the program and data into main memory, one word at a time, as described in Section 1.5. Then, you set the initial value of the program counter: to do this, set the memory address switches to the desired value (typically `10`). Now, you can press the GO or STEP button to initiate the computation. From this point on, the X-TOY machine executes instructions in a specific, well-defined way. First it checks the value of the `pc` and fetches the contents of this memory location. Next, it increments the `pc` by 1. (For example, if `pc = 10`, then it will get incremented to `11`.) Finally, it interprets this data as an instruction and executes it according to the rules in Section 1.9. Each instruction can modify the contents of various registers, main memory, or even the program counter itself. It may also output integers to the standard input and standard output. After executing the instruction, the whole fetch-execute cycle is repeated, using the new value of the program counter to obtain the next instruction. This continues forever, or until the machine executes a halt instruction. As with C, it is possible to write programs that go into infinite loops. It is always possible to stop the X-TOY machine by pulling the plug.

1.8 von Neumann Machine

One of the essential characteristics of the X-TOY machine is that it stores computer programs as numbers, and both data and programs are stored in the *same* main memory. In 1945, Princeton scholar John von Neumann first popularized this *stored-program model* or *von Neumann machine*.[1] It enables computers to perform any type of computation, without requiring the user to physically alter or reconfigure the hardware. In contrast, to program the ENIAC computer, the operator had to manually plug in cables and set switches. This was quite tedious and time consuming. This simple but fundamental idea of the stored-program machine has been incorporated into all modern digital computers.

Since the program and data share the same space, the machine can modify its data or the program itself while it is executing. That is, code and data are the same, or at least they can be. The memory is interpreted as an instruction when the `pc` references it, and as data when an instruction references it. The ability to treat the program as data is crucial. Consider what happens when you want to download a program from a remote location, e.g., `download.com`. It is no different from receiving email, or any other data. Compilers and debuggers are also programs that read in other programs as input data. Treating programs as data is not without its perils. Computer viruses are (malicious) programs that propagate by writing new programs or modifying existing ones.

In hindsight von Neumann's idea may seem obvious. However, it is not obvious whether a computer built around a stored program model can be as powerful as a computer that can be rewired and reconfigured. In fact, the ability to physically reconfigure a computer does not enable you to solve more problems, so long as basic instruction set is rich enough (as is the case with the X-TOY machine). This is a consequence of previous work by Alan Turing on "Turing machines." We will study this later in the course.

[1]In 1945, von Neumann circulated a memo that described the stored-program model. Although it contained only his name, historians believe that Eckert and Mauchly are also deserving of credit. However, the idea of the stored-program computer was explicit in Alan Turing's work on what was later to become known on universal Turing machines.

1.9 Instruction Set Architecture

The *instruction set architecture* (ISA) is the interface between the X-TOY programming langauge and the physical hardware that will execute the program. The ISA specifies the size of main memory, number of registers, and number of bits per instruction. It also specifies exactly which instructions the machine is capable of performing and how each of the instruction bits is interpreted.

The X-TOY ISA. As discussed above, the X-TOY machine has 256 words of main memory, 16 registers, and 16-bit instructions. There are 16 different instruction types; each one is designated by one of the *opcodes* `0 - F`. Each instruction manipulates the contents of memory, registers, or the program counter in a completely specified manner. The 16 X-TOY instructions are organized into three categories: arithmetic-logic, transfer between memory and registers, and flow control. The table below gives a brief summary. We will describe them in more detail in Section 2.

Opcode	Description	Format	Type	Pseudocode
`1`	add	1		`R[d] <- R[s] +  R[t]`
`2`	subtract	1	arithmetic	`R[d] <- R[s] -  R[t]`
`3`	and	1	logic	`R[d] <- R[s] &  R[t]`
`4`	xor	1		`R[d] <- R[s] ^  R[t]`
`5`	shift left	1		`R[d] <- R[s] << R[t]`
`6`	shift right	1		`R[d] <- R[s] >> R[t]`
`7`	load address	2		`R[d] <- addr`
`8`	load	2		`R[d] <- mem[addr]`
`9`	store	2	transfer	`mem[addr] <- R[d]`
`A`	load indirect	1		`R[d] <- mem[R[t]]`
`B`	store indirect	1		`mem[R[t]] <- R[d]`
`0`	halt	1		`halt`
`C`	branch zero	2		`if (R[d] == 0) pc <- addr`
`D`	branch positive	2	flow control	`if (R[d] > 0) pc <- addr`
`E`	jump register	2		`pc <- R[d]`
`F`	jump and link	2		`R[d] <- pc; pc <- addr`

Each X-TOY instruction consists of 4 hex digits (16 bits). The leading (left-most) hex digit encodes one of the 16 opcodes. The second (from the left) hex digit refers to one of the 16 registers, which we call the *destination register* and denote by `d`. The interpretation of the two rightmost hex digits depends on the opcode. With *Format 1* opcodes, the third and fourth hex digits are each interpreted as the index of a register, which we call the two *source registers* and denote by `s` and `t`. For example, the instruction `1462` adds the contents of registers `s = 6` and `t = 2` and puts the result into register `d = 4`. With *Format 2* opcodes, the third and fourth hex digits (the rightmost 8 bits) are interpreted as a memory address, which we denote by `addr`. For example, the instruction `9462` stores the contents of register `d = 4` into memory location `addr = 62`. Note that there is no ambiguity between Format 1 and Format 2 instruction since each opcode has a unique format.

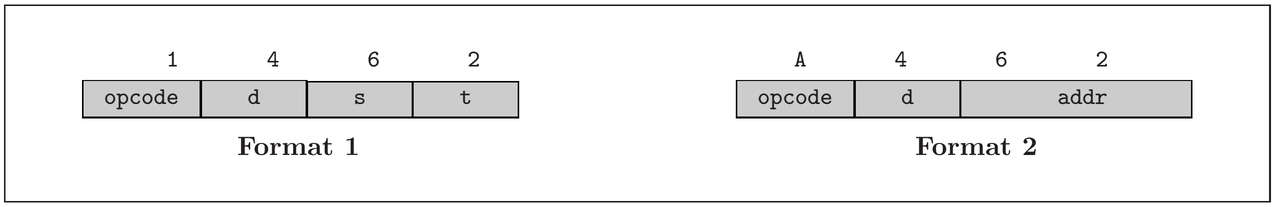


Figure 2: Parsing an X-TOY instruction.

Modern day microprocessors and ISAs. Today, there is a wide variety of ISAs used on modern day microprocessors: IA-32 (Intel, AMD), PowerPC (IBM, Motorola), Alpha (Compaq), PA-RISC (Hewlett-Packard), and SPARC (SUN Microsystems). These ISAs typically access millions of words of main memory, each of which is 32 or 64 bits wide. The number of instruction types varies greatly from half a dozen to several hundred. The ISA is chosen in order to make it easy to build the underlying hardware and compilers, while striving to maximize performance and minimize cost. Unfortunately, sometimes both goals are sacrificed to maintain backward compatibility with obsolete hardware. There are always tradeoffs.

The X-TOY machine possesses all of the essential features of modern day microprocessors. However, it is much easier to understand because it has only 16 instructions. In contrast, the IA-32 (the one used on Intel PC's) has more than 100 instruction types and over a dozen different instruction formats. As a programming language, X-TOY is also much simpler than the C programming language. This makes it easier to fully understand, but not to write code or debug. A *C compiler* (e.g., cc, gcc, lcc) is a program that automatically converts C code into the ISA of the computer on which it is to be executed. You will soon appreciate the convenience of working in a higher-level language like C.

2 X-TOY Programming

Although the X-TOY machine language contains only 16 different instruction types, it is possible to perform a variety of interesting computations. In fact, any computation that can be done in the C programming language on your PC (or in any programming language on any supercomputer) can also be done in X-TOY (provided you give X-TOY enough main memory and time). This may come as quite a surprising fact; we will justify it later in the course. Below, we describe features of the X-TOY language.

2.1 Load Address

The *load address* instruction (opcode 7) is the most primitive type of assignment statement in the X-TOY language. For now, you should think of it as an instruction that loads a specified integer into a register. [2] As an example, Program 2.1 initializes register 5 to the value `002F`. Recall that all values in X-TOY are expressed in hexadecimal notation, so this corresponds to the decimal integer 47.

Program 2.1 This program stores the value `002F` into register 5.

```
    10: 752F    R[5] <- 002F
    11: 0000    halt
```

Note that opcode `7` only allows you to assign 8 bit integers (`00 - FF`) to a register, even though registers are capable of storing 16 bit integers.

2.2 Add and Subtract

The *add* and *subtract* instructions (opcodes 1 and 2) perform the conventional arithmetic operations. Remember that all operations involve 16 bit two's complement integers, as described in Appendix A. Program 2.2 stores the values `0029` and `0078` into registers 1 and 2, respectively. Then, it computers their sum, and stores the result in register 3.

Program 2.2 This program computes `0029` + `0078` = `00A1`.

```
    10: 7129    R[1] <- 0029
    11: 7278    R[2] <- 0078
    12: 1312    R[3] <- R[1] + R[2]
    13: 0000    halt
```

[2] You will understand why the instruction is called load *address* rather than load *constant* when we discuss arrays.

Upon termination of this program, register 3 contains the value 00A1. If you computed the result 107, start getting adjusted to working with hexadecimal integers. Analogously, Program 2.3 computes 0029 - 0078 = FFB1. Review Appendix A if you do not understand the answer FFB1, which is the decimal equivalent of -79.

Program 2.3 This program computes 0029 - 0078 = FFB1.

```
    10: 7129    R[1] <- 0029
    11: 7278    R[2] <- 0078
    12: 2312    R[3] <- R[1] - R[2]
    13: 0000    halt
```

The curious reader might wonder what happens if the result of the arithmetic operations is too large to fit into a 16 bit register. Such overflow is handled by disregarding everything except the rightmost 4 hex digits. So, the result of adding EFFF and 1005 is 0004, since EFFF + 1005 = 10004 in hex and we discard the leading digit.

2.3 Load and Store

To transfer data between registers and main memory, we use the *load* (opcode 8) and *store* (opcode 9) instructions. This type of operation is convenient because it is not possible to perform arithmetic operations directly on the contents of main memory. Instead, the data must be first transferred to registers. There are many circumstances where it is not possible to maintain all of our program's variables simultaneously in registers, e.g., if we need to store more than 16 quantities. We overcome the 16 register limitation by storing variables in main memory, and transferring them back and forth to registers using the load and store instructions.

In Program 2.4, think of memory addresses 00, 01, and 02 as storing variables a, b, and c, respectively. The program calculates c = a + b by loading the variables a and b into temporary registers 1 and 2, then storing the sum in register 3, and finally transferring the contents of register 3 back to memory address 02.

Program 2.4 This program adds the two integers stored in memory locations 00 and 01, and stores the result in memory location 02.

```
    00: 0029    a
    01: 0078    b
    02: 0000    c

    10: 8A00    R[A] <- mem[00]          load a
    11: 8B01    R[B] <- mem[01]          load b
    12: 1CAB    R[C] <- R[A] + R[B]      a + b
    13: 9C02    mem[02] <- R[C]          store c
    14: 0000    halt
```

Upon termination, memory location 02 contains the value 00A1. Notice the difference between load and load address; this is a common source of confusion. It is important to note that memory locations 00 - 02 do not get executed; they are treated as data.

The load and store instructions are also used to access *standard input* and *standard output*. The distinguished memory address FF is intercepted by an X-TOY interrupt: instead of loading or storing information in memory location FF, the data is received from the keyboard or is sent to the screen. To illustrate the process, Program 2.5 reads two integers from standard input, and writes the sum to standard output.

Program 2.5 This program reads in two integers from standard input, computes their sum, and writes the result to standard output.

```
    10: 8AFF    read R[A]
    11: 8BFF    read R[B]
    12: 1CAB    R[C] <- R[A] + R[B]
    13: 9CFF    write R[C]
    14: 0000    halt
```

When the program is executed, it pauses until the user types in two integers. As usual, the integers are specified as 4 hexadecimal digits. Then it computes their sum, and prints it to the screen.

2.4 Branch Statements

The *branch if zero* and *branch if positive* instructions (opcodes C and D) statements are useful for performing `while` loops and `if-else` conditional statements. We will illustrate how they are used by writing a program that multiplies two integers.

Conspicuously absent from the X-TOY instruction set is a multiply instruction. To achieve the same effect in software, we describe and implement an algorithm to multiply two integers. The brute force way to compute `c = a * b` is to set `c = 0`, and then add `a` to `c`, `b` times. This suggests having a loop that repeats `b` times. We accomplish this by making a counter variable `i` that we initialize to `b`, and then decrement it by one until it reaches 0. We use the branch if positive instruction to detect this event.

Program 2.6 This program reads in two integers from standard input, computes their product using the brute-force algorithm, and writes the result to standard output.

```
10: 8AFF   read R[A]
11: 8BFF   read R[B]

12: 7C00   R[C] <- 0000                c = 0
13: 7101   R[1] <- 0001                always 1

14: CA18   if (R[A] == 0) goto 18      while (a != 0) {
15: 1CCB   R[C] <- R[C] + R[B]            c += b
16: 2AA1   R[A] <- R[A] - R[1]            a--
17: C014   goto 14                     }

18: 9CFF   write R[C]
19: 0000   halt
```

The astute reader might notice that our algorithm suffers from a serious performance flaw. The burte force algorithm is *extremely* inefficient if the values are large. The loop iterates `b` times, and since `b` is a 16-bit integer, it can be as large as 32,767. This issue would be much more pronounced on a 64-bit machine where the loop might require a mind-boggling 9,223,372,036,854,775,807 iterations! Fortunately, we can incorporate better algorithmic ideas (see Program 2.11) to rescue this otherwise hopeless task.

2.5 X-TOY Idioms

There are several common idioms or pseudo-instructions in X-TOY that can be used for common programming tasks. Many of these tricks rely on the fact that register 0 always stores the value `0000`.

Register-To-Register Transfer. Suppose you want to make register 2 have the same value as register 1. There is no built-in instruction to do this. Relying on the fact that register 0 always contains `0000`, we can use the addition instruction to sum up `R[0]` and `R[1]` and put the result in `R[2]`.

```
14: 1201    R[2] <- R[1]
```

Another way to achieve the same effect is with the bitwise AND instruction (opcode 3).

```
14: 3211    R[2] <- R[1] & R[1]
```

This works because ANDing an integer with itself produces the original integer. As in most machine languages, there are often many ways to achieve the same goal.

Nothing Statement (no-op). In a structured programming language like C (with `for` and `while` loops), inserting extra code is easy. In an unstructured language like X-TOY (where there are line numbers and goto statements), you must be careful about inserting code. A branch statement hardwires in the memory address to jump to; if you insert code, the line numbers of your program may change. To avoid some of this awkwardness, machine language programmers often find it convenient to fill in the program with "useless" statements to act as placeholders. The instruction `1000` is ideal for this purpose since register 0 is always 0 anyway. [3]

Goto. There is no instruction that directly changes the program counter to the `addr`. However, it is easy to use the branch if zero instruction with register 0 to achieve the same effect. For example, the instruction `C0F0` changes the program counter to `F0` since register 0 is always 0.

2.6 Functions

In C it is quite useful to divide a program up into smaller functions. We can do the same in X-TOY. Below is an X-TOY program that "calls" a multiply function with two arguments and computers their product. Since all of the variables (registers) are global, we need to agree upon a protocol for calling our function. We'll assume that we want to multiply the integers stored in registers `A` and `B`, and store their product in register `C`. The program below works by using two branch if zero statements, one in the "main program" starting at `10`, and one in the "function" starting at `F0`.

```
10: 7A05    R[A] <- 0005
11: 7B06    R[B] <- 0006
12: C0F0    goto F0
13: 9CFF    print R[C]
14: 0000    halt

F0: B101    R[1] <- 0001             R[1] always 1 in this function
F1: 8C00    R[C] <- 0                store result
F2: 120B    R[2] <- R[B]             i = b
                                     do {
F3: 1CCA    R[C] <- R[C] + R[A]         c += a
F4: 2221    R[2] <- R[2] - R[1]         i--
F5: D2F3    if (R2 > 0) goto F3      } while(i > 0)
F6: C013    goto 13
```

The above function is a bit unsatisfying, since it hardwires in the memory address `13` in the return statement `f6: C013`. This means that we can only call the function from line `12`; otherwise the function will return to the wrong place. We write a better version that calls the multiply function twice to compute $x \times y \times z$, once to compute $x \times y$, then again to compute $(x \times y) \times z$. We use the *jump and link* and *jump register* instructions that are especially designed for this purpose.

[3]Just so there's no confusion, the instruction `1012` would also be a no-op because register 0 always contains 0 regardless of how you might try to change it.

Program 2.7 contains the complete code. Instructions `11` and `14` store the value of the `pc` in register `R3`, before jumping to the function located at `F0`. This makes it possible to return back to the main program, without hardwiring in the address into the program.

Program 2.7 This program reads in three integers x, y, and z from standard input, and prints out $x \times y \times z$. It packages up the multiplication program into an X-TOY function that begins at line `30`.

```
10: 82FF   read R[2]                 x
11: 83FF   read R[3]                 y
12: 84FF   read R[4]                 z

13: 1A20   R[A] <- R[2]              x
14: 1B30   R[B] <- R[3]              y
15: FF30   R[F] <- pc; goto 30       x * y

16: 1AC0   R[A] <- R[C]              x * y
17: 1B40   R[B] <- R[4]              z
18: FF30   R[F] <- pc; goto 30       (x * y) * z

19: 9CFF   write R[C]
1A: 0000   halt


30: 7C00   R[C] <- 0000
31: 7101   R[1] <- 0001
32: CA36   if (R[A] == 0) goto 36
33: 1CCB   R[C] <- R[C] + R[B]
34: 2AA1   R[A] <- R[A] - R[1]
35: C032   goto 32
36: EF00   goto R[F]
```

Now, every time the program counter is reset to `F0`, the old program counter is saved away in register F for future use. Instruction `F5` returns from the function by restting the program counter to the value stored in register F. Note also, that the program counter is incremented *before* the instruction is executed. Thus, during the first function call `R[F] = 16`, not `15`.

Be very careful about which variables you are using when writing machine language functions. There is no such thing as a "local variable." Had we continued to use register 2 as the loop counter in the multiplication function, this would have overwritten register 2 in the main program, which was being used to store the quantity b.

2.7 Horner's method.

We can use the multiplication function to *evaluate polynomials*: given integer coefficients $a_n, \ldots, a_2, a_1, a_0$, and an integer x, evaluate $p(x) = a_n x^n + \cdots + a_2 x^2 + a_1 x + a_0$ at the integer x. Polynomial evaluation was one *raison d'être* for early machines. It has many applications including studying ballistic motion and converting an integer from its decimal representation to hexadecimal.

The brute force algorithm for polynomial evaluation is to sum up the $n+1$ terms: term i is the product of a_i and x^i. To compute x^i we could write a power function that multiplies x by itself $i-1$ times.

Horner's method is a clever alternative that is more efficient and easier to code. The basic idea is to judiciously sequence the way in which terms are multiplied. We can rewrite an order 3 polynomial

$$\begin{aligned} p(x) &= a_3 x^3 + a_2 x^2 + a_1 x + a_0 \\ &= (((a_3)x + a_2)x + a_1)x + a_0. \end{aligned}$$

Similarly, we can rewrite an order 5 polynomial

$$\begin{aligned} p(x) &= a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0 \\ &= (((((a_5)x + a_4)x + a_3)x + a_2)x + a_1)x + a_0. \end{aligned}$$

Using Horner's method, only n multiplications are required to evaluate an order n polynomial. Moreover, we can translate the method directly into C or X-TOY code.

Program 2.8 This program reads in a sequence of integers x, n, and $a_n, \ldots, a_2, a_1, a_0$ from standard input and prints out $p(x) = a_nx^n + \cdots + a_2x^2 + a_1x + a_0$.

```
10: 7C00   R[C] <- 0000                 result c
11: 7101   R[1] <- 0001                 always 1
12: 82FF   read R[2]                    read x
13: 83FF   read R[3]                    read n

                                        do {
14: 84FF   read R[4]                        read a_i
15: 1A20   R[A] <- R[2]
16: 1BC0   R[B] <- R[C]                     c *= x
17: FF30   R[F] <- pc; goto 30
18: 1CC4   R[C] <- R[C] + R[4]              c += a_i
19: C31C   if (R[3] == 0) goto 1C
1A: 2331   R[3] <- R[3] - R[1]           } while (i-- >= 0)
1B: C014   goto 14

1C: 9CFF   write R[C]
1D: 0000   halt

// brute force multiply function
30: 7C00   R[C] <- 0000
31: 7101   R[1] <- 0001
32: CA36   if (R[A] == 0) goto 36
33: 1CCB   R[C] <- R[C] + R[B]
34: 2AA1   R[A] <- R[A] - R[1]
35: C032   goto 32
36: EF00   goto R[F]
```

We can use Program 2.8 to convert a decimal integer to its hexadecimal representation. To convert 765_{10} to hex, we set the input $x = A$, $n = 3$, $a_2 = 7$, $a_1 = 6$, and $a_0 = 5$. Since all arithmetic is performed in hex, the program computes the hexadecimal equivalent of $7 \times 10^2 + 6 \times 10 + 5 = 2FD_{16}$.

2.8 Arrays

Arrays are not directly built into the X-TOY language, but it is possible to achieve the same functionality using the *load indirect* and *store indirect* instructions. We illustrate this technique with two examples. First, we will consider a program that reads in a sequence of integers and prints them in reverse order. Then, we will consider a more sophisticated application of arrays that performs the classic insertion sort algorithm on a sequence of integers.

Reverse. The following program reads from standard input a sequence of positive integers followed by the integer 0000. It stores them in main memory, starting at address 30. We use the load address instruction to store the address 30 into register A.[4] Then, it marches through the elements in reverse order, printing them out to standard input. We use register B to keep track of the number of elements read in. We arrange

[4]Now it should make sense why opcode 7 is referred to as load address.

it so that register 6 contains the memory location of the array element that we are currently reading or writing. To write and read an array element, we use the *load indirect* (opcode A) and *store indirect* (opcode B) instructions, respectively.

Program 2.9 This program reads in a sequence of positive integers and prints them out in reverse order.

```
10: 7101   R[1] <- 0001                 R[1] always 1
11: 7A30   R[A] <- 0030                 memory address of array a[]
12: 7B00   R[B] <- 0000                 # elements in array = n

// read in sequence of positive integers
13: 8CFF   read R[C]                    while (read R[C]) {
14: CC19   if (R[C] == 0) goto 19          if (c == 0) break
15: 16AB   R[6] <- R[A] + R[B]             a + n
16: BC06   mem[R[6]] <- R[C]               a[n] = c
17: 1BB1   R[B] <- R[B] + R[1]             n++
18: C013   goto 13                      }

// print out results in reverse order
19: CB20   if (R[B] == 0) goto 20       while (n != 0) {
1A: 16AB   R[6] <- R[A] + R[B]             a + n
1B: 2661   R[6] <- R[6] - R[1]             a + n - 1
1C: AC06   R[C] <- mem[R[6]]               c = a[n-1]
1D: 9CFF   write R[C]                      print c
1E: 2BB1   R[B] <- R[B] - R[1]             n--
1F: C019   goto 19                      }
20: 0000   halt
```

The program suffers from one important flaw. Since the X-TOY machine has only 256 memory locations, it is not possible to store or reverse a list that contains too many elements. In the example above, after the program fills up memory locations `30 - FF`, it will wrap around and start writing into memory locations `0 - F`. Pretty soon, it will start overwriting the lines of the original program `10 - 20`. A devious user could exploit this *buffer overflow* and input integers in such a way that the integers from standard input get interpreted as instructions rather than data. Viruses can be spread by such buffer overflow attacks.

Insertion sort. The following program reads in a sequence of positive integers from standard input and insertion sorts them. The program terminates upon reading in a nonpositive integer.

Program 2.10 This program reads in a sequence of positive integers and insertion sorts them.

```
// R[1] always 1
10: 7101   R[1] <- 0001

// R[A] = memory address of array element 0 = a[]
11: 7A40   R[A] <- 0040

// R[B] = # elements in array = n
12: 7B00   R[B] <- 0000

// read integer from stdin into R[C] = x until "EOF"
13: 8CFF   read R[C]
14: CC25   if (R[C] == 0) pc <- 25

// a[n] = x
15: 16AB   R[6] <- R[A] + R[B]
16: BC06   mem[R[6]] <- R[C]

// i = n
17: 12B0   R[2] <- R[B]
18: C223   if (R[2] == 0) pc <- 23

// compare a[i] and a[i-1]
19: 16A2   R[6] <- R[A] + R[2]        // a + i
1A: A706   R[7] <- mem[R[6]]          // a[i]
1B: 2861   R[8] <- R[6] - 1           // a + i - 1
1C: A908   R[9] <- mem[R[8]]          // a[i-1]
1D: 2379   R[3] <- R[7] - R[9]        // a[i] - a[i-1]
1E: D321   if (R[3] > 0) pc <- 21
// swap
1F: B906   mem[R[6]] <- R[9]
20: B708   mem[R[8]] <- R[7]

// decrement inner loop
21: 2221   R[2]--
22: D219   if (R[2] > 0) pc <- 19

// increment outer loop
23: 1BB1   R[B] <- R[B] + R[1]
24: C013   pc <- 13

25: 0000   halt
```

2.9 Logical Operators

The logical operations (AND, XOR, shift left, shift right) work just like the analogous ones in C. Because you may not have used these operators in C, we review them here.

AND and XOR. The bitwise AND and bitwise XOR functions take two 16 bit integers, and apply the corresponding Boolean operator to each of the 16 pairs of bits. For example, if registers 1 and 2 contain the values `00B4` and `00E3`, then the instruction `3312` assigns the value `00A0` to register 3. In order to see why, look at the binary representation of registers 1 and 2, and take the AND of each corresponding bit.

```
R[1] = 0000 0000 1011 0100 (binary) = 00B4 (hex)
```

```
R[2] = 0000 0000 1110 0011 (binary) = 00E3 (hex)
R[3] = 0000 0000 1010 0000 (binary) = 00A0 (hex)
```

Shifting. The *left shift* operator shifts the bits over a certain number of places to the left, padding 0's on the right. For example, if register 2 has the value `00B4` and register 3 has the value `0002`, then the instruction `5423` assigns the value `02D0` to register 4. To see why, look at the binary representation.

```
R[2]      = 0000 0000 1011 0100 (binary) = 00B4 (hex)  = 180 (dec)
R[3] << 2 = 0000 0010 1101 0000 (binary) = 02D0 (hex)  = 720 (dec)
```

Note that left shifting by one bit is equivalent to multiplication by 2; left shifting by i bits is equivalent to multiplying by 2^i.

The *right shift* operator is similar, but the bits get shifted to the right. Leading 0's or 1's are padded on the left, according to the sign bit (the leftmost bit). For example, if register 2 has the value `00B4` and register 3 has the value `0002`, then the instruction `6423` assigns the value `02D0` to register 4. To see why, look at the binary representation.

```
R[2]      = 0000 0000 1011 0100 (binary) = 00B4 (hex)  =  180 (dec)
R[2] >> 2 = 0000 0000 0010 1101 (binary) = 002D (hex)  =   45 (dec)
```

The value is register 2 is nonnegative so 0's are padded on the left.

If register 2 has the value `FF4C` instead of `00B4`, then the result of the right shifting is `FFD3`. In this case

```
R[2]      = 1111 1111 0100 1100 (binary) = FF4C (hex)  = -180 (dec)
R[2] >> 2 = 1111 1111 1101 0011 (binary) = FFD3 (hex)  = - 45 (dec)
```

In this case, the value in register 2 is negative so 1's are padded on the left.

Note that right shifting an integer by 1 bit is equivalent to dividing the integer by 2 and throwing away the remainder. This is true regardless of the sign of the orginal integer. In general, right shifting an integer by i bits is equivalent to dividing it by 2^i and throwing away any remainder. This type of shifting is called an *arithmetic shift*: it preserves the sign for two's complement integers. In contrast, a *logical right shift* always pads 0's on the left.

Efficient multiplication. Using the bitwise operators, we provide an efficient implementation of the multiplication function from Section 2.4. Recall, in Section 2.4, we computed $c = a \times b$ by setting $c = 0$, then adding a to c, b times. This severely handicaps performance when b is large because the loop iterates b times.

To multiply two 16-bit integers a and b, we let b_i denote the ith bit of b. That is,

$$b \quad = \quad b_{15} \times 2^{15} \ + \cdots + \ b_2 \times 2^2 \ + \ b_1 \times 2 \ + \ b_0 \times 2^0.$$

By distributivity, we obtain:

$$a \times b \quad = \quad (a \times b_{15} \times 2^{15}) \ + \cdots + \ (a \times b_2 \times 2^2) \ + \ (a \times b_1 \times 2) \ + \ (a \times b_0 \times 2^0).$$

Thus, to compute $a \times b$, it suffices to add the above 16 terms. Naively, this appears to reduce the problem of performing one multiplication to 32 multiplication, two for each of the 16 terms. Fortunately, each of these 32 multiplications are of a very special type. First, observe that $a \times 2^i$ is the same as left shifting a by i bits. Second, note that b_i is either 0 or 1; thus term i is either `a << i` or 0. The following X-TOY function loops 16 times. In iteration i, it computes the ith term and adds it to the running total stored in register C. To gain some perspective, recall the standard grade school algorithm for multiplying two decimal integers. The bitwise procedure we just described is really just the grade school algorithm applied to binary integers.

Program 2.11 This program reads in two integers from standard output and writes their product to standard output. It uses the grade school (binary) multiplication algorithm.

```
10: 8AFF   read R[A]
11: 8BFF   read R[B]
12: FF30   R[F] <- pc; goto 30
13: 9CFF   write R[C]
14: 0000

30: 7101   R[1] <- 0001
31: 7210   R[2] <- 0010             i = 16
32: 7C00   R[C] <- 0000             result

33: C23B   if (R[2] == 0) goto 3B   while (i > 0) {
34: 2221   R[2]--                      i--
35: 53A2   R[3] <- R[A] << R[2]        a * 2^i
36: 64B2   R[4] <- R[B] >> R[2]
37: 3441   R[4] <- R[4] & R[1]         b_i = ith bit of b

38: C43A   if (R[4] == 0) goto 3A
39: 1CC3   R[C] += R[3]                c += b_i * a * 2^i

3A: C033   goto 33                  }

3B: EF00   goto R[F]                return
```

3 The X-TOY Simulator

Suppose we are interested in designing a new machine or microprocessor. We could test it out by building a prototype machine and run various programs on it. A different approach would be to write a program on an existing machine that would simulate the behavior of the new machine. This has two main advantages. First it would be easy to extend the simulator to add extra flexibility, e.g., add debugging tools or experiment with a different ISA. Second, it is much cheaper than building a prototype for a new machine.

Designing a simulator can have other benefits. Suppose our X-TOY machine becomes obsolete by the more powerful NOTATOY machine. But, we have written thousands of programs in the X-TOY language and are reluctant to rewrite them for NOTATOY. Instead, we could build a simulator for the X-TOY machine on NOTATOY. Now, we can run all of our existing programs on NOTATOY by running them on our X-TOY simulator. As you might suspect, this extra layer of simulation will slow down our code somewhat. Many ancient programs are currently still running on today's computers, under several layers of simulation. For example, it is still possible to run the Apple IIe game Lode Runner under Microsoft Windows NT.

3.1 An X-TOY Simulator in C

Figure 3.1 gives a C program that reads in an X-TOY program and *simulates* the behavior of the X-TOY machine.[5] Remarkably, it fits on one page. The C program uses arrays to store the registers and main memory. It also stores the `pc`. The C program reads in the X-TOY program and alters the appropriate register, memory, and `pc` contents in exactly the same way that the X-TOY machine would modify its registers, memory, and `pc`. Any program written for the X-TOY machine could ultimately be used on the

[5]For simplicity, we have ignored dealing with standard input and standard output, although these changes would be fairly straightforward. We also note that the C language type `short` is not required to be a 16-bit two's complement integer, although on many systems it is.

X-TOY simulator, and any program written for the X-TOY simulator could be used on the X-TOY machine, were it to be built. A Java simulator with a graphical interface is available from the course web page.

Program 3.1 Simulating the X-TOY machine in C. We use arrays `R[]` and `mem[]` to store the registers and main memory. We assume that `short` is a two's complement 16-bit integer and `unsigned char` is an 8-bit integer.

```
int main(void) {
   short R[16] = {0};                       // 16 16-bit registers
   short mem[256] = {0};                    // 256 16-bit memory locations
   unsigned char pc = 0x10;                 // 8-bit program counter
   int inst;                                // current instruction
   int op, addr, d, s, t;


   // read X-TOY program from a file???
   while(scanf("%2hX: %4hX", &addr, &inst) != EOF)
      mem[addr] = inst;

   // fetch-execute cycle
   do {
     inst = mem[pc++];                     // fetch next instruction
     op   = (inst >> 12) &  15;            // get opcode (bits 12 - 15)
     d    = (inst >>  8) &  15;            // get d      (bits  8 - 11)
     s    = (inst >>  4) &  15;            // get s      (bits  4 -  7)
     t    = (inst >>  0) &  15;            // get t      (bits  0 -  3)
     addr = (inst >>  0) & 255;            // get addr   (bits  0 -  7)

     switch (op) {
        case  0:                            break;    // halt
        case  1: R[d] = R[s] +  R[t];       break;    // add
        case  2: R[d] = R[s] -  R[t];       break;    // subtract
        case  3: R[d] = R[s] &  R[t];       break;    // bitwise and
        case  4: R[d] = R[s] ^  R[t];       break;    // bitwise xor
        case  5: R[d] = R[s] << R[t];       break;    // shift left
        case  6: R[d] = R[s] >> R[t];       break;    // shift right
        case  7: R[d] = addr;               break;    // load address
        case  8: R[d] = mem[addr];          break;    // load
        case  9: mem[addr] = R[d];          break;    // store
        case 10: R[d] = mem[R[t]];          break;    // load indirect
        case 11: mem[R[t]] = R[d];          break;    // store indirect
        case 12: if (R[d] == 0) pc = addr;  break;    // branch if zero
        case 13: if (R[d] > 0) pc = addr;   break;    // branch if positive
        case 14: pc = R[d];                 break;    // jump register
        case 15: R[d] = pc; pc = addr;      break;    // jump and link
     }
     R[0] = 0;  // register 0 always outputs 0

   } while (op != 0);

   return 0;
}
```

3.2 Translator

It is possible to *translate* any particular TOY program into a C program. That is, given an X-TOY program (e.g., insertion sort), write a corresponding C program that does the same thing. This is analogous to translating a book from French into Spanish. Note that simulation is not the same as translation. A simulator mimics the behavior of the original machine exactly line-by-line, whereas a translator needs to generate code that produces the same output given the same input. In principle, it is also possible, but somewhat tedious, to translate a C program into X-TOY.

3.3 Bootstrapping

We now consider a mind-bending idea with great practical significance. Since it is always possible to translate a C program into X-TOY, let's translate our X-TOY simulator (written in C) into a program written in the X-TOY language! That is, we create an X-TOY program that simulates the X-TOY machine itself. Now, we could modify the X-TOY simulator, e.g., to add debugging tools, or to simulate variants of the X-TOY machine. The resulting X-TOY simulator program is "more powerful" than the original X-TOY machine. This idea is called *bootstrapping*, where once we build one machine, we can use it to simulate "more powerful" machines. This fundamental idea is now used in the design of all new computers.[6]

[6] According to Apple's web site "Seymour Cray, founder of Cray Research and father of several generations of supercomputers, heard that Apple had bought a Cray to simulate computer design. Cray was amused, remarking, Funny, I am using an Apple to simulate the Cray-3."

A Representing Integers

In this appendix, we review how to represent integers in binary, decimal, and hexadecimal. We also review how to convert between bases and do arithmetic. Finally, we describe how to represent negative integers using "two's complement notation."

A.1 Number Systems

There are many ways to represent integers: the number of days in the month of October can be represented as 31 in decimal, 11111 in binary, 1F in hexadecimal, or XXXI in Roman Numerals. It is important to remember than an integer is an integer, no matter whether it is represented in decimal or with Roman Numerals. We are most familiar with performing arithmetic with the *decimal* (base 10) number system. This number system has been universally adopted in large part because we have 10 fingers. Clocks are based on the *sexagecimal* (base 60) number system, mainly because 60 has lots of divisors. Computers are based on the *binary* (base 2) number system because each wire can be in one of two states (on or off). The *hexadecimal* (base 16) number system is often used in machine languages, including X-TOY, as a shorthand for binary. Base 16 is useful because 16 is a power of 2 with a reasonable number of digits.

A sequence of digits $x = x_n\, x_{n-1}\, \ldots\, x_1\, x_0$ in base b represents the integer

$$x \;=\; x_n\, b^n + x_{n-1}\, b^{n-1} \;+\; \ldots \;+\; x_1\, b^1 + x_0\, b^0. \tag{1}$$

This representation is called *positional notation.* The x_i are the *positional digits*, and each digit is required to be an integer between 0 and $b-1$. In binary, the two digits (also referred to as bits) are 0 and 1; in decimal, the ten digits are 0 through 9; in hexadecimal, the sixteen digits are 0 through 9 and the letters A through F. Every positive integer can be expressed using positional notation, and the representation is unique modulo an arbitrary number of the leading 0's. As an example the number of days in a leap year is:

$$\begin{aligned}
366 \text{ (decimal)} &= \mathbf{3} \times 10^2 + \mathbf{6} \times 10^1 + \mathbf{6} \times 10^0 \\
&= \mathbf{1} \times 2^8 + \mathbf{0} \times 2^7 + \mathbf{1} \times 2^6 + \mathbf{1} \times 2^5 + \mathbf{0} \times 2^4 + \mathbf{1} \times 2^3 + \mathbf{1} \times 2^2 + \mathbf{1} \times 2^1 + \mathbf{0} \times 2^0 \\
&= 101101110 \text{ (binary)} \\
&= \mathbf{1} \times 16^2 + \mathbf{6} \times 16^1 + \mathbf{14} \times 16^0 \\
&= \text{16E (hexadecimal).}
\end{aligned}$$

The following table gives the binary, decimal, and hexadecimal representations of the first 48 integers.

Bin	Dec	Hex	Bin	Dec	Hex	Bin	Dec	Hex
0	0	0	10000	16	10	100000	32	20
1	1	1	10001	17	11	100001	33	21
10	2	2	10010	18	12	100010	34	22
11	3	3	10011	19	13	100011	35	23
100	4	4	10100	20	14	100100	36	24
101	5	5	10101	21	15	100101	37	25
110	6	6	10110	22	16	100110	38	26
111	7	7	10111	23	17	100111	39	27
1000	8	8	11000	24	18	101000	40	28
1001	9	9	11001	25	19	101001	41	29
1010	10	A	11010	26	1A	101010	42	2A
1011	11	B	11011	27	1B	101011	43	2B
1100	12	C	11100	38	1C	101100	44	2C
1101	13	D	11101	29	1D	101101	45	2D
1110	14	E	11110	30	1E	101110	46	2E
1111	15	F	11111	31	1F	101111	47	2F

A.2 Number Conversion

In this section we describe a few techniques for converting among various representations, including binary, decimal, and hexadecimal.

Converting to Decimal. It is straightforward to convert an integer represented in base b to an integer represented in decimal: use Formula (1), performing ordinary decimal arithmetic. As an example, to convert the binary number 101101110 to decimal, compute:

$$\begin{aligned} 101101110_2 &= \mathbf{1}\times 2^8 + \mathbf{0}\times 2^7 + \mathbf{1}\times 2^6 + \mathbf{1}\times 2^5 + \mathbf{0}\times 2^4 + \mathbf{1}\times 2^3 + \mathbf{1}\times 2^2 + \mathbf{1}\times 2^1 + \mathbf{0}\times 2^0 \\ &= 256+64+32+8+4+2 \\ &= 366_{10}. \end{aligned}$$

Converting from Decimal. It is slightly more difficult to convert an integer represented in decimal to one in base b because we are accustomed to performing arithmetic in base 10. One way to do the conversion is to apply Formula (1), using *base b arithmetic.* An alternate method that uses only decimal arithmetic is to repeatedly divide by the base b, and read the remainder upwards, as in Figure 3. This is usually the best way to do it by hand.

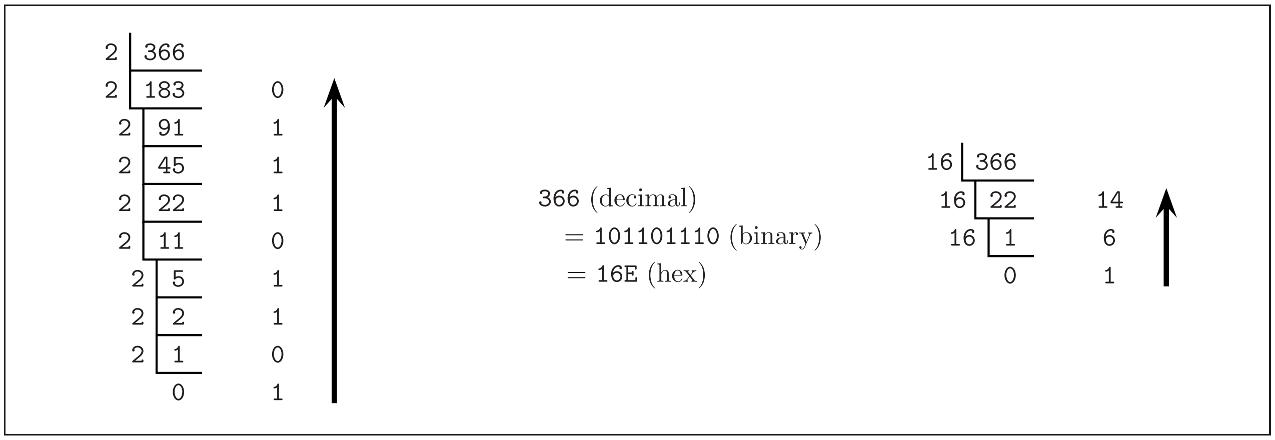


Figure 3: To convert from base b to decimal, repeatedly divide by b and read the remainders upwards.

Converting between Binary and Hexadecimal. We describe a fast way to convert from the binary to hexadecimal representation of an integer. In principle, we could convert from binary to decimal, and then from decimal to hexadecimal. The following approach is simpler. To convert from binary to hexadecimal: first, group the digits 4 at a time starting from the right; then convert each group to a single hexadecimal digit, padding 0's to the very last group if necessary.

$$111010111001110001 \text{ (binary)} = \underbrace{0011}_{3}\ \underbrace{1010}_{\text{A}}\ \underbrace{1110}_{\text{E}}\ \underbrace{0111}_{7}\ \underbrace{0001}_{1} = \text{3AE71 (hex)}.$$

To convert from hexadecimal to binary: convert each hexadecimal digit individually into its corresponding 4 digit binary number, removing any leading 0's.

$$\text{9F03 (hex)} = \underbrace{9}_{1001}\ \underbrace{\text{F}}_{1111}\ \underbrace{0}_{0000}\ \underbrace{3}_{0011} = 1001111100000011 \text{ (binary)}.$$

These simple procedures work because one base is a power of the other. Likewise, it would be easy to convert between the base 125 and base 5 representations.

A.3 Arithmetic in Other Number Systems

Addition. In grade school you learned how to add two decimal integers: add the two least significant digits (rightmost digits); if the sum is more than 10, then carry a 1 and right down the sum modulo 10. Repeat with the next digit, but this time include the carry bit in the addition. The same procedure can be generalized to base b by replacing the 10 with the base b. For example, if you are working in base 16 and the two summand digits are 7 and E, then you should carry a 1 and write down a 6 because $7 + E = 16_{16}$. Below, we compute $4567_{10} + 366_{10} = 4933_{10}$ in binary (left), decimal (middle) and hexadecimal (right).

$$
\begin{array}{ccccccccccccccc}
 & & & & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & & \\
 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 \\
+ & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\
\hline
 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1
\end{array}
\qquad
\begin{array}{ccccc}
 & & {\scriptstyle 1} & {\scriptstyle 1} & \\
 & 4 & 5 & 6 & 7 \\
+ & & 3 & 6 & 6 \\
\hline
 & 4 & 9 & 3 & 3
\end{array}
\qquad
\begin{array}{ccccc}
 & & {\scriptstyle 1} & {\scriptstyle 1} & \\
 & 1 & 1 & D & 7 \\
+ & & 1 & 6 & E \\
\hline
 & 1 & 3 & 4 & 5
\end{array}
$$

Multiplication. One compelling reason to use positional number systems is to facilitate multiplication. Multiplying two Roman Numerals is awkward and slow. In contrast, the grade school algorithm for multiplying two decimal integers is straightforward and reasonably efficient. As with addition, it easily generalizes to handle base b integers. All intermediate single-digit multiplications and additions are done in base b. Below, we multiply the decimal integers 4,567 and 366, and then the same integers represented in hex.

$$
\begin{array}{ccccccc}
 & & & 4 & 5 & 6 & 7 \\
 & & \times & & 3 & 6 & 6 \\
\hline
 & & 2 & 7 & 4 & 0 & 2 \\
 & 2 & 7 & 4 & 0 & 2 & \\
1 & 3 & 7 & 0 & 1 & & \\
\hline
1 & 6 & 7 & 1 & 5 & 2 & 2
\end{array}
\qquad\qquad
\begin{array}{cccccc}
 & & 1 & 1 & D & 7 \\
 & & \times & 1 & 6 & E \\
\hline
 & & F & 9 & C & 2 \\
 & 6 & B & 0 & A & \\
1 & 1 & D & 7 & & \\
\hline
1 & 9 & 8 & 1 & 6 & 2
\end{array}
$$

A.4 Signed and Unsigned Integers

On a machine with 16-bit words, there are $2^{16} = 65,536$ possible integers that can be stored in one word of memory. By interpreting the 16 bits as a binary number, we obtain an *unsigned integer* in the range 0 – 65,535. Instead, we can interpret the leading bit as the sign of the number, using *two's complement notation*. This allows us to interpret the 16 bits as a *signed integer* in the range $-32,768$ to $+32,767$, as described in the table below. As with binary integers, it is often convenient to express 16-bit two's complement integers using hexadecimal notation.

Binary	Hex	Decimal
0000 0000 0000 0000	0000	0
0000 0000 0000 0001	0001	+1
0000 0000 0000 0010	0002	+2
0000 0000 0000 0011	0003	+3
...		
0111 1111 1111 1110	7FFE	+32,766
0111 1111 1111 1111	7FFF	+32,767
1000 0000 0000 0000	8000	-32,768
1000 0000 0000 0001	8001	-32,767
1000 0000 0000 0010	8002	-32,766
...		
1111 1111 1111 1101	FFFD	-3
1111 1111 1111 1110	FFFE	-2
1111 1111 1111 1111	FFFF	-1

Negating a Two's Complement Integer. To *negate* a two's complement integer, first complement all of the bits, then add 1. By *complement*, we mean replace all of the 0's with 1's, and the 1's with 0's. The table below illustrates a few examples.

	Integer	Complement	Increment	
`+3`	`0000 0000 0000 0011`	`1111 1111 1111 1100`	`1111 1111 1111 1101`	`-3`
`-3`	`1111 1111 1111 1101`	`0000 0000 0000 0010`	`0000 0000 0010 0011`	`+3`
`+40`	`0000 0000 0010 1000`	`1111 1111 1101 0111`	`1111 1111 1101 1000`	`-40`
`+366`	`0000 0001 0110 1110`	`1111 1110 1001 0001`	`1111 1110 1001 0010`	`-366`
`-40`	`1111 1111 1101 1000`	`0000 0000 0010 0111`	`0000 0000 0010 1000`	`+40`
`0`	`0000 0000 0000 0000`	`1111 1111 1111 1111`	`0000 0000 0000 0000`	`0`
`-32,768`	`1000 0000 0000 0000`	`0111 1111 1111 1111`	`1000 0000 0000 0000`	`-32,768`

Number Conversions. We describe how to convert between the decimal and two's complement representations of an integer. To convert the 16-bit two's complement integer `FE92` into decimal, we start by writing down its binary representation: `1111 1110 1001 0010`. We recognize it as a negative integer since the most significant bit is `1`. Then, we negate it (flip bits and add 1) to obtain: `0000 0001 0110 1110`. As in Section A.2, we can convert 101101110 (binary) to 366 (decimal). After putting back in the negative sign, we obtain the final answer of -366 (decimal).

To covert from the decimal integer -366 to its 16-bit two's complement representation, we can apply the above steps in reverse. First we convert 366 (decimal) into 101101110 (binary), as in Figure 3. Next, we negate it (flip bits and add one) to obtain `1111 1110 1001 0010`. It is important that we fill in all 16 bits. If desired, we can convert this to hexadecimal: `FE92`.

Adding Two's Complement Integers. Adding two's complement integers is straightforward: add the numbers as if they were unsigned integers, ignoring any overflow. Below we compute $4567_{10} + (-366_{10}) = 4201_{10}$ using 16-bit two's complement integers. Note that the second binary integer represents a negative integer using two's complement notation.

$$
\begin{array}{ccccc}
 & & {\scriptstyle 1} & {\scriptstyle 1} & \\
 & 4 & 5 & 6 & 7 \\
+ & & \text{-3} & 6 & 6 \\
\hline
 & 4 & 2 & 0 & 1
\end{array}
\qquad
\begin{array}{ccccccccccccccccc}
 & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & & {\scriptstyle 1} & & {\scriptstyle 1} & {\scriptstyle 1} & & \\
 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 \\
+ & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\
\hline
 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1
\end{array}
$$

In the example above, we carry a one into the most significant bit (leftmost bit), and we carry a one out, which we subsequently discard. Despite the apparent overflow, we are left with the correct result. There is one situation where this will produce an incorrect answer: if we do not carry a one into the most significant bit, but do carry a one out. Below we compute $-32,766_{10} + (-366_{10}) = -33,132_{10}$ using 16-bit two's complement integers. At first, we might be surprised to see that the result of adding two negative integers is a positive integer (to see this quickly, look at the most significant bit of the result). This occurs because there is carry out of the most significant digit, but no carry in to it. In hindsight, we should not be surprised because the true answer -33,132 cannot be represented as a 16-bit two's complement integer.

$$
\begin{array}{cccccc}
 & \text{-3} & 2 & 7 & 6 & 6 \\
+ & & & \text{-3} & 6 & 6 \\
\hline
 & \text{-3} & 3 & 1 & 3 & 2
\end{array}
\qquad
\begin{array}{cccccccccccccccccc}
 & {\scriptstyle 1} & & & & & & & & & & & & & & & & \\
 & & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
+ & & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\
\hline
 & & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0
\end{array}
$$