Computer Science 126 and 128	Spring 1996

Handout # 3: The Toy Machine

by G.Humphreys, A. LaPaugh, and S. Arora *Feb. 20, 1996.*

1 Introduction

The Toy machine is a ficticious computer designed for COS 126/128. It is a simplifed processor that captures the basics of how modern microprocessors work. While it is less complex than the microprocessors inside today's personal computers and workstations, it has the same fundamental organization. Understanding it will give you the foundations of understanding microprocessor operation, and also a pretty good idea of what really happens when a program in a high-level language (such as C) executes inside the computer.

As in all existing processors, information (i.e., programs and data) are represented inside the Toy machine using *bits*: 0's and 1's. For this reason we first remind you of the binary number system and the related hexadecimal ("hex") number system.

1.1 Binary and Hex Number Systems

The number system we use everyday is the *decimal* number system, which means that it represents integers using an "alphabet" of 10 digits: $0, 1, \ldots, 9$. Thus the sequence of digits 20395 represents the following integer:

$$2 \times 10^4 + 0 \times 10^3 + 3 \times 10^2 + 9 \times 10^1 + 5 \times 10^0.$$

In the binary number system, the alphabet consists of only $0, 1$. A sequence of binary digits (aka "bits") is interpreted similarly as in the decimal case, except we use powers of 2 instead of powers of 10. Thus the sequence of bits 10111 represents the following integer:

$$1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0.$$

Finally, the hexadecimal number system uses an alphabet of 16 digits:

$$0, 1, 2, \ldots, 9, A, B, C, D, E, F,$$

where $A, B, \ldots, F$ stand for the values from 10 to 15. A sequence of hexadecimal digits is interpreted similarly as in the decimal case, except we use powers of 16 instead of powers of 10. Thus the sequence of letters $A10D$ represents the following integer:

$$10 \times 16^3 + 1 \times 16^2 + 0 \times 16^1 + 13 \times 16^0.$$

Note that $16 = 2^4$, so 1 "hex" digit $\equiv$ 4 bits. Thus the hex number $A10D$ is represented in binary as 1010000100001101. In C, hexadecimal constants are written using the prefix "0x" or "0X" , e.g. 0xA10D. We will use this convention below when there might be confusion as to whether a constant is in decimal or hexadecimal.

2 The Machine

Toy consists of a central processing unit (CPU), registers, and memory. There is an interface to the outside world through a terminal.

The memory and the registers can be used to store machine instructions and data, which are encoded in binary. Each register and each memory location can hold 16 bits, and the machine can only manipulate (i.e., operate upon) chunks of 16 bits at a time. (To use standard computer jargon, the Toy machine uses a 16-bit *word*. Modern microprocessors have 32-bit or 64-bit words.)

Registers

A *register* is a place to temporarily hold one word of data. Registers provide fast access to data (in a real processor, they are built into the CPU), but they are expensive, so there are not very many of them. The Toy machine has 8 registers. This means that you can only store 8 words in registers at a time. (If you need to store more than 8 pieces of data, you store it in memory and move each word from memory to a register when you are about to use it in a computation. You can keep a few frequently used words in registers throughout the program execution.) We'll call the registers R_0, R_1, $\ldots$, R_7. Since $8 = 2^3$, we see that 3 bits are needed to write the number identifying a register.

Memory

Memory provides the main storage to hold both programs and data; "data" here refers to both initial data and intermediate results. Modern microprocessors have large amounts of memory, but Toy is much more primitive: it only has 256 words of memory. Since $256 = 2^8$, the *address* of each location in memory is represented by 2 hex digits (= 8 bits). Thus the following are addresses of memory locations: $A4$, AF, 08. We use bracket notation to denote a memory location: M[08] denotes the contents of (i.e., the value in) the memory location with address 08. Likewise, M[A4] denotes the contents of the memory location with address 0xA4 (=164). (Thus M can be thought of as an array of 256 locations.)

Program Execution

A *program* for the Toy machine is a sequence of *instructions*. (Instructions are described below.) The instructions are stored in memory. A special register called the *program counter* (PC) holds the address of the next instruction to execute. To execute a program, you first load the program in memory using external input, then externally set the value in the PC to be the address of the first instruction to execute, then tell the machine to "go". Once execution is started, the Toy machine repeats the following sequence of actions until it encounters a HALT operation.

- **fetch** from memory the instruction at the memory location in the PC;
- **increment** the PC;
- **execute** the instruction just fetched.

In other words, the Toy machine goes through the instructions in memory and executes them one by one. However, there are a few exceptions to this sequential behavior: certain instructions can modify the address stored in the PC, which means that at the next step the machine will look at ("jump to") that new address for its next instruction.

3 The instructions

An *instruction* tells the machine *what* operation to do and *which* data to perform it on (this data is called the *operand* of the instruction). There are 16 different types of instructions. Each instuction consists of 4 hex digits (i.e., 16 bits), like `6B07`. The first digit (in this case, 6), is called the *opcode*. It specifies which of the sixteen operations is to be done. The other three hexadecimal digits represent the operands; they specify registers or constant values depending on the operation to be done. Details will be given below.

We make some general remark about the notation used below. We use register names similarly to the way variables are used in the C language. Thus an instruction that adds the current *value* of R_2 and R_3 and stores the sum in R_6 is abbreviated as

$$R_6 \leftarrow R_3 + R_2.$$

Some instructions (namely, the ones with opcodes 9 and A) use the current value of a register as a memory address. (Note: The value stored in a register has 16 bits whereas a memory address has 8 bits. It is an error to try to access a memory location with an address larger than $0xFF$ and the machine will halt.) Thus the notation M[R_4] denotes the contents of the memory location whose address is the value of R_4.

Table 3 gives a summary of the 16 opcodes and how instructions containing them are executed.

0 – HALT This opcode tells the Toy machine to quit. The other three digits are ignored entirely.

1 – ADD The ADD opcode does just that – it adds the values stored in two registers, and puts the result in a third. The destination register is the second digit, and the two registers to add are the third and fourth digit. So, for example, the instruction `1632` would take the contents of R_3 and R_2, and put their sum in R_6, i.e. $R_6 \leftarrow R_3 + R_2$.

2 – SUBTRACT The SUBTRACT opcode is similar to the ADD opcode; it subtracts the values in two registers and puts the result in a third. The destination register is again the second digit, and the two registers to subtract are the third and fourth digits. So, for example, `2632` executes the command $R_6 \leftarrow R_3 - R_2$.

OPCODE	NAME	FORMAT	ACTION
0	HALT	$0xyz$	Stop Execution
1	ADD	$1xyz$	$R_x \leftarrow R_y + R_z$
2	SUBTRACT	$2xyz$	$R_x \leftarrow R_y - R_z$
3	MULTIPLY	$3xyz$	$R_x \leftarrow R_y \star R_z$
4	SYSTEM CALL	$4xyz$	if $(yz == 2)$ print R_x
5	JUMP	$5xyz$	PC $\leftarrow yz$
6	JUMP$_{<0}$	$6xyz$	if $(R_x < 0)$ PC $\leftarrow yz$
7	JUMP INDIRECT	$7xyz$	PC $\leftarrow R_x$
8	JUMP & LINK	$8xyz$	$R_x \leftarrow$ PC; PC $\leftarrow yz$
9	LOAD	$9xyz$	$R_x \leftarrow M[R_y + z]$
A	STORE	$Axyz$	$M[R_y + z] \leftarrow R_x$
B	LOAD IMMEDIATE	$Bxyz$	$R_x \leftarrow yz$
C	EXCLUSIVE-OR	$Cxyz$	$R_x \leftarrow R_y \mathbin{\hat{}} R_z$
D	AND	$Dxyz$	$R_x \leftarrow R_y \& R_z$
E	SHIFT RIGHT	$Exyz$	$R_x \leftarrow R_y >> R_z$
F	SHIFT LEFT	$Fxyz$	$R_x \leftarrow R_y << R_z$

Table 1: The opcodes, the format of instructions that use those opcodes, and the machine's action on executing those instructions. Note that x, y, z denote any hexadecimal digits. If any of x, y, z is meant to be the index of a register, it has to take a value from 0 to 7, otherwise the machine produces an *error*. Thus the instruction $1A23$ produces an error, since there is no register numbered A.

3 – MULTIPLY The MULTIPLY opcode works in exactly the same way as the SUBTRACT and the ADD ones work. So, `3632` would execute $R_6 \leftarrow R_3 \times R_2$.

4 – SYSTEM CALL The SYSTEM CALL command is used to perform tasks related to the computer itself. (In a "real" computer, these would be tasks that require the intervention of the operating system.) The only system command available on the Toy machine is *print*. The register to print is the second digit. The third and fourth digits must be 02 to designate printing. (Some day Toy might be able to do other system calls, and these two digits would code what system action needs to be taken.) For example, the command `4402` would print the value in register 4 on the screen. The printed value is always followed by a carriage return.

5 – JUMP The JUMP opcode allows you to hop around in your code non-sequentially. The new next instruction is in the memory location specified by the third and fourth digits of the instruction; the second digit is ignored. This memory address is put in the PC so that at the next instruction fetch, the new instruction location is used. The instruction `502C` would make the next instruction executed be at memory location 2C. We write this as $PC \leftarrow 2C$.

6 – JUMP IF $<$ 0 This opcode provides a way to *conditionally* jump to some location in memory. This means that, in addition to specifying a memory location to jump to, you also specify a register. If the contents of that register is less than zero, you jump (change the PC). Otherwise, you simply continue executing in the normal way. This provides a way to do loops.

7 – JUMP INDIRECT The JUMP INDIRECT opcode is exactly the same as the JUMP opcode, except that in this case, the destination address is stored in a register rather than being specified explicitly in the instruction. The second digit specifies the register and the third and fourth digits are ignored. So, for example, `7200` would execute the command $PC \leftarrow R_2$.

8 – JUMP AND LINK The JUMP AND LINK opcode, together with the JUMP INDIRECT opcode, provide a way to write *subroutines* in Toy code. You would use this opcode if you wanted to branch to some section of code that represented a function, and at the end of the function, you wanted to continue execution right after where you called the function from. You can think of this as saving your page in a book while you go do something else, so you can resume reading where you left off. This opcode needs a register to store the current PC and also a destination address. The register number is the second digit, and the destination address is the third and fourth digits. Therefore, the instruction `824D` would execute the command $R_2 \leftarrow PC; PC \leftarrow 4D$. It should now be clear why the JUMP INDIRECT command is useful – the function called should use the JUMP INDIRECT opcode to return to the caller.

9 – LOAD The LOAD opcode provides a way to get a value from memory into a register. The second digit designates the register to receive the value (destination register). The third and fourth digits are used to compute the memory address from which to retrieve the value. The third digit designates a register holding a memory address; the fourth digit designates a constant. The memory address is the sum of the contents of the register and the constant. Consider the command `9326`. This executes the command $R_3 \leftarrow M[R_2 + 6]$.

A – STORE The STORE opcode is just the reverse of the LOAD opcode. It allows you to take a value stored in a register, and put it into memory. It's syntax is similar to the LOAD instruction; the value to be stored is in the register designated by the second digit and the memory address is calculated using the third and fourth digits to give a register and constant. For example, instruction `A235` would execute the command $M[R_3 + 5] \leftarrow R_2$.

B – LOAD IMMEDIATE The LOAD IMMEDIATE allows you to put a value into a register without going to memory. It simply takes the destination register as the second digit, and the value to load as the third and fourth digit. For example, `B25F` would execute the command $R_2 \leftarrow 5F$.

C – EXCLUSIVE OR The EXCLUSIVE OR opcode allows you to perform a bitwise exclusive or (xor) on the values in two registers, and save the result in a third. (Bitwise exclusive or can be done by the C operator `^`. The xor of two bits is a 1 if the bits are different, and is a 0 if they are the same. Xor is also often denoted using the symbol $\oplus$.) The EXCLUSIVE OR command takes the destination register as the second digits, and the two operand registers as the third and fourth digits, just as the arithmetic opcodes ADD, SUBTRACT, and MULTIPLY do. For example, the instruction `C451` would execute the command $R_4 \leftarrow R_5 \mathbin{\hat{}} R_1$.

D – AND The AND opcode performs a bitwise logical and of two registers, and stores the result in a third register. The syntax is the same as the EXCLUSIVE OR opcode. So, the instruction `D152` would execute the command $R_1 \leftarrow R_5 \& R_2$.

E – SHIFT RIGHT The SHIFT RIGHT opcode allows you to shift the bits in a register to the right. (Any bits that "fall off" the right end of the register are simply lost.) The result of the shift operation is stored in another register. This opcode takes the destination register as the second digit, the source register as the third digit, and the fourth digit is a register specifying by how many bits to shift. Therefore, the instruction `E510` would execute the command $R_5 \leftarrow R_1 >> R_0$ ($>>$ is C notation).

F – SHIFT LEFT The SHIFT LEFT opcode is exactly the same as the SHIFT RIGHT opcode, except it shifts the other way. So, the instruction `F104` would execute the command $R_1 \leftarrow R_0 << R_4$ ($<<$ is C notation).

4 The Simulator

Unfortunately, we don't have the resources to build a Toy machine for real. Instead, we use a *simulator*.

The Toy simulator is a C program that reads as input a program written in the Toy machine language (henceforth called a *Toy program*), and simulates the execution of the Toy machine on that program.

Using the toy simulator

The toy program will be stored in a file; the format of that file is specified below. Suppose the file `mytoyprog.toy` contains a program written in the Toy machine language. To simulate the program you type

```
/u/cs126/toy/simulate  <mytoyprog.toy
```

(The `‘‘<’’` directs UNIX to send the contents of the file `mytoyprog.toy` as standard input to the simulation program. IT IS NECESSARY. Forget it, and the program will sit waiting for input.)

The simulator will execute the Toy program, printing what the Toy machine would print on the standard output. When the Toy machine would halt, the simulator produces a "dump" giving the final values in the machine PC, registers and memory on the standard output (terminal). If you want the simulator output to go to a file use the output redirection `‘‘>’’`:

```
/u/cs126/toy/simulate  <mytoyprog.toy >mytoyprog.out
```

The simulator also has a *trace* option that will give an instruction-by-instruction history of the toy instructions that have been executed, giving the memory location, machine-code instruction, and comment showing what the instruction did (like the comments written for the Toy programs in the lecture notes). To use this trace option type:

```
/u/cs126/toy/simulate  -t <mytoyprog.toy >mytoyprog.out
```

The trace is put in the file `mytoyprog.out` with the final dump. When the trace option is used, the output that is printed by the Toy program is part of the trace, i.e. there is no separate output from print statements (system calls to print) in the Toy program. The trace option is very useful for debugging a Toy program.

Format for Toy Program

The file containing a Toy program for simulation should have the following:

The **first line** of the file must contain the starting address of the program, i.e. the initial value of the PC, in hexadecimal (hex). Nothing else should be on this line.

The **remaining lines** of the file should each contain 3 fields; the fields are separated by blanks or tabs:

Field 1 the memory address in hex to be used in storing the contents to follow. This field **must be terminated with a ":"**.

Field 2 the contents to be put in memory – either an instruction or data, written as 4 hex digits.

Field 3 an optional comment, which is ignored.

The simulator will read this file. It will first initialize the PC to the value found on the first line. Then, the simulator will store the contents field of each remaining line in the indicated memory location of the Toy machine. After the file has been read, the simulator will start executing Toy instructions using the initial PC value for the address of the first instruction.

Example

The following lines are a legal Toy program for the simulator;
find them in file `/u/cs126/toy/horner.toy`:

```
10
  10: B330      R3 <- 30          (base address)
  11: 9430      Load R4 <- M[R3+0] = M[30] = x
  12: 9531      Load R5 <- M[R3+1] = M[31] = a
  13: 3554      R5 <- R5 * R4 = a*x
  14: 9632      Load R6 <- M[R3+2] = M[32] = b
  15: 1556      R5 <- R5 + R6 = a*x+b
  16: 3554      R5 <- R5 * R4 = (a*x+b)*x
  17: 9633      Load R6 <- M[R3+3] = M[33] = c
  18: 1556      R5 <- R5 + R6 = (a*x+b)*x + c
  19: 4502      write R5 to tty
  1A: 0000      halt
  30: 000A
  31: 0002
  32: 0003
  33: 0009
```

This program evaluates the polynomial $ax^2 + bx + c$ using Horner's rule (see the comment on line 18 above). The values of x, a, b, and c, are in memory locations 30, 31, 32, and 33, respectively. The following output is produced by `/u/cs126/toy/simulate <horner.toy`. Note that the dump does not print lines for which all memory contents are 0000:

```
0xEF

PC=001B


0000 0000 0000 0030 000A 00EF 0009 0000


0010: B330 9430 9531 3554 9632 1556 3554 9633
0018: 1556 4502 0000 0000 0000 0000 0000 0000
0030: 000A 0002 0003 0009 0000 0000 0000 0000
```