Eric Roberts
CSCI 121

Handout #17
September 21, 2018

Computers on the Inside

This handout describes a few fundamental concepts about how the hardware works inside the computer. Nothing in this handout or in the lecture on Monday, September 24, is required for the class, but students have sometimes found it helpful to be able to visualize how the computer executes their code. This handout builds on the discussion of binary representation in section 6.1 of the reader and assumes that you understand how computers store integers as sequences of *bits* collected into fixed-size units called *words*.

Programming languages and the underlying machine

Programming languages like Python are called ***higher-level languages*** because they allow you to express algorithmic processes at a high level of abstraction. The advantage of higher-level languages is that they enable you to focus on the holistic aspects of algorithmic design rather than the nitty-gritty details of computer architecture. At the same time, it is often useful to understand how computers work on a more detailed level. For one thing, learning about the internal structure of a computer helps to demystify its operation, which in turn makes computing more accessible. For another, learning about the structure of a typical machine gives you additional insight and intuition into how certain Python features, such as references, actually work.

As useful as it is to know something about the architecture of the underlying machine, it isn't feasible to introduce these concepts in the context of a highly sophisticated machine like the PC or the Macintosh. Modern computers are much too complex to cover their entire structure in an introductory course. It is therefore traditional to introduce the concepts of machine architecture using a hypothetical machine and then simulating its operation. This handout describes a hypothetical machine called ***Toddler,*** a very simple machine that is nonetheless powerful enough to illustrate the basic concepts of digital computer architecture.

The stored-program concept

One of the most important breakthroughs in the history of computing was the realization that programs and data could be stored in the same memory, as long as the programs could be represented in numeric form. Fortunately, creating a numeric representation for programs is no harder than creating a numeric representation for characters, as described in Chapter 6. If a computer uses a particular set of instructions, its designers could apply the principle of enumeration by assigning a numeric value to each instruction. That numeric value, together with additional numeric information to indicate what memory location was involved, would then constitute a ***machine instruction.*** A program is then simply a sequence of machine instructions, each of which has a numerical value. The technique of storing programs and data in the same memory is called the ***stored-program model.***

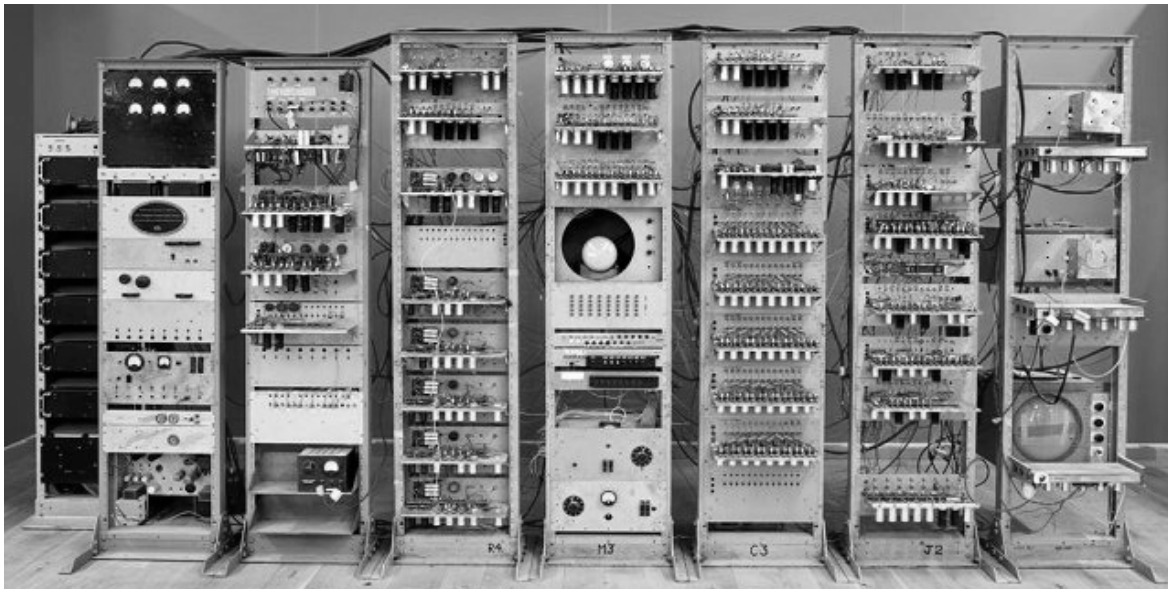
John von Neumann

It is impossible—and presumably unnecessary—to identify a single inventor of the stored-program idea. This design is traditionally called the ***von Neumann architecture*** after the Hungarian/American mathematician John von Neumann, who published a detailed description of the model in 1945. At that time, however, the stored-program idea was already circulating among American computing pioneers. The fundamental ideas are even older than that. Representing programs as data is central to Alan Turing's work on computability, which you will learn about if you go on to take CSCI 387. Some elements of that idea were also used by the German engineer Konrad Zuse in the 1930s.

Even though it is unclear who deserves credit for the idea of the stored-program model, there is no controversy as to when the stored-program model was first implemented in practice. That distinction belongs to the Small-Scale Experimental Machine—nicknamed the "Baby"—which ran its first program on June 21, 1948. The leaders of the design team at the University of Manchester were F. C. Williams, Tom Kilburn, and Geoff Tootill. The primary goal of the project was to demonstrate the feasibility of the stored-program model rather than to build a computer that could be used for practical applications. As a prototype, however, the Baby was extremely successful and led directly to the development of the Ferranti Mark I, which was the first commercial computer to use the stored-program design. In 1998, computer history enthusiasts from the British Computer Society created a replica of the Baby to celebrate its fiftieth anniversary. That replica, which appears in Figure 1, is on display at the Museum of Science and Industry in Manchester, England.

Despite its nickname, the Baby was gargantuan by modern standards, at least in terms of its physical dimensions. It stood more than seven feet tall and occupied a series of cabinets that extended for 17 feet. Modern processors have billions of times the capacity of the Baby, even though they fit on a chip much smaller than a penny.

Figure 1. The rebuilt Manchester Baby



The computational capacity of the Baby was also severely limited. Its repertoire of seven instructions included subtraction but not addition, because it is possible to simulate addition by subtracting a number from zero and then subtracting that result from some other value. The Baby contained only 32 words of memory, which had to store both the data and the program instructions. These restrictions rendered the Baby unsuitable for anything other than the simplest of programs, such as carrying out a single long division, finding the largest factor of a number, or calculating the greatest common divisor.

Introducing Toddler

The restrictions on the Baby are so severe that it doesn't work well as a teaching tool. To give you a sense of how von Neumann architectures work, this chapter introduces a more powerful computer that is a few steps up from the Manchester Baby (which accounts for the name *Toddler*). Although Toddler's increased size and power makes it possible to write more interesting programs, the machine is still small enough so that you can see its entire structure in a single diagram, as shown in Figure 2.

The Toddler machine is divided into two parts. The three labeled boxes on the left side of Figure 2 represent the ***central processing unit,*** or ***CPU,*** which acts as the computational engine for the machine. The right side of the diagram shows the ***memory,*** which consists of 100 words, where each word contains three decimal digits preceded by a plus or a minus sign.

After spending a section of the reader extolling the virtues of the binary system, the choice of decimal representation for the Toddler machine will certainly seem like a step backward. As it happens, Claude Shannon's critics were partly right in their defense of the decimal system: decimal representation is easier for humans to understand than its binary counterpart. Given that the goal of the Toddler machine is to help you understand how computers work on the inside, it makes sense to use decimal arithmetic.

Each individual word in Toddler's memory is identified by a numeric ***address,*** which ranges from 00 to 99. Because doing so makes it easier to catch various common programming errors, the Toddler hardware makes it illegal to use address 00, which therefore appears as a gray box in Figure 2. This design decision, which is replicated in

Figure 2. Structure of the Toddler machine

AC +000

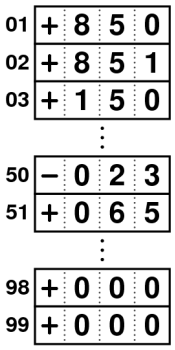
PC 01

IR +000

	0*x*	1*x*	2*x*	3*x*	4*x*	5*x*	6*x*	7*x*	8*x*	9*x*
0		+450	+000	+000	+000	+000	+000	+000	+000	+000
1	+850	+251	+000	+000	+000	+000	+000	+000	+000	+000
2	+851	+503	+000	+000	+000	+000	+000	+000	+000	+000
3	+150	+950	+000	+000	+000	+000	+000	+000	+000	+000
4	+451	+500	+000	+000	+000	+000	+000	+000	+000	+000
5	+613	+000	+000	+000	+000	+000	+000	+000	+000	+000
6	+709	+000	+000	+000	+000	+000	+000	+000	+000	+000
7	+250	+000	+000	+000	+000	+000	+000	+000	+000	+000
8	+503	+000	+000	+000	+000	+000	+000	+000	+000	+000
9	+151	+000	+000	+000	+000	+000	+000	+000	+000	+000

some form in most modern architectures, means that the usable memory actually begins at address 01.

Given that each of Toddler's memory words holds a three-digit signed integer, the range of each memory word is –999 to +999. The following diagram therefore shows a possible configuration of Toddler's memory, although only a few words of the 99 usable memory addresses are shown:



In the diagram, the word at address 01 contains the number +850, the word at address 50 contains the number –23, and so forth.

Even though memory locations in the Toddler machine contain three-digit signed integers, it is important to keep in mind that you can interpret those integers in different ways. For example, the value 65 in location 51 could represent either the number 65 or the Unicode character `'A'`, as described in Chapter 6. Both the number and the character have the internal representation 65. The correct interpretation depends on how the value is used.

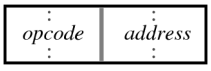
In addition to the circuitry required to implement the operations of the machine, the CPU includes three specialized memory cells called ***registers,*** as follows:

- `AC` The ***accumulator,*** which is used for arithmetic computation
- `PC` The ***program counter,*** which determines which instruction to execute
- `IR` The ***instruction register,*** which decodes the current instruction

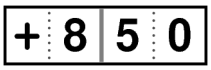
The purpose and format of each of these registers are described later in this handout.

Programming the Toddler machine

In order to represent instructions inside a machine, there needs to be an encoding scheme that allows the hardware to interpret the contents of some memory location as an executable instruction. In the Toddler machine, each instruction is divided into two parts. The first digit of the memory word indicates what instruction is being performed, and the last two digits indicate an address in memory. Thus, an instruction in Toddler is logically divided into the following fields, which are called the ***opcode*** (short for operation code) and the ***address:***



For example, if Toddler were to execute the value in address 01 above as an instruction, it would take the value +850 and divide it into its opcode and address components, as follows:



The first digit (together with the sign, which is used for extended instructions beyond the scope of this handout) specifies the code for the particular instruction to be performed. The last two digits specify the address of the word in memory on which the operation will be performed.

The Toddler instruction set

The standard instructions of the Toddler machine consist of the nine instructions shown in Figure 3. In the current example, the +8 opcode specifies an **INPUT** instruction, which reads in a value from the user. Given that the complete instruction word is +850, the value entered by the user is stored at address 50.

You can use the Toddler instructions to write simple programs that execute their instructions in order. The following program, for example, reads in two numbers from the user and then prints their sum:

```
(01) +850
(02) +851
(03) +150
(04) +351
(05) +252
(06) +952
(07) +500
```

The numbers in parentheses indicate the addresses in which each instruction is stored. By convention, the instructions in a Toddler program beginning at address 01.

The first line of the program is the instruction +850, which represents an **INPUT** instruction. When the Toddler machine encounters this instruction, it types out a

Figure 3. The Toddler instruction set

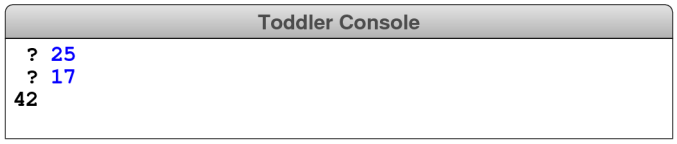
Numeric	Assembler form	Description
+1*xx*	**LOAD** *xx*	Loads the AC with the contents of memory address *xx*.
+2*xx*	**STORE** *xx*	Stores the contents of the AC into memory address *xx*.
+3*xx*	**ADD** *xx*	Adds the contents of memory address *xx* to the AC.
+4*xx*	**SUB** *xx*	Subtracts the contents of memory address *xx* from the AC.
+5*xx*	**JUMP** *xx*	Jumps to address *xx*. If *xx* is 00, the machine halts.
+6*xx*	**JUMPZ** *xx*	Jumps to address *xx* if the AC is zero.
+7*xx*	**JUMPN** *xx*	Jumps to address *xx* if the AC is negative.
+8*xx*	**INPUT** *xx*	Reads an integer from the user into address *xx*.
+9*xx*	**OUTPUT** *xx*	Prints the integer in address *xx* on the console.

question mark and then waits for the user to enter a number. When the user hits the RETURN key to signal the end of the number, Toddler stores that value in location 50. The next instruction does the same thing for the second input value, storing the result in location 51. In this program, memory locations 50 and 51 are used to hold data values, which are being interpreted here as integers. Locations used to hold data that changes over the course of the program represent ***variables,*** which have the same purpose in Toddler than they do in Python.

At this point in the execution of the program, the variables in location 50 and 51 contain the two input values. The next step in the process is to add the numbers together. In the Toddler machine—and in the Manchester Baby that serves as its inspiration—all arithmetic must be done in the **AC** register. Thus, to add the two numbers, the program must first load one of the values into the **AC**, add the second, and then store the result back into memory. These operations are accomplished using the following code, which consists of a **LOAD** instruction, an **ADD** instruction, and a **STORE** instruction:

```
+150
+351
+252
```

At this point, address 52 contains the desired result. To display that result back to the user, the program executes the +952 instruction in location 06, which is an **OUTPUT** instruction. The program then moves on to the +500 at location 07, which is a **HALT** instruction. At the end of the program, the Toddler console looks like this:



The instruction cycle

By convention, all Toddler programs begin at address 01. To make sure that instructions are executed in their proper order, Toddler—like any machine that uses the von Neumann architecture—devotes an internal register that keeps track of the next instruction in sequence. That register is called the ***program counter*** or **PC**. When the program is started, the **PC** is set to 01 to indicate that the first instruction to be executed comes from address 01. Toddler uses another internal register, called the ***instruction register*** or **IR**, to hold the three-digit instruction word.

For each instruction, Toddler executes the following instruction cycle:

1. *Fetch the current instruction.* In this phase of the instruction, Toddler finds the word from the memory address specified by the **PC** and copies its value into the **IR**.
2. *Increment the program counter.* Once the current instruction has been copied into the **IR**, Toddler adds one to the **PC** so that it indicates the next instruction in sequence.
3. *Decode the instruction in the instruction register.* The value copied into the **IR** is a three-digit integer. To use it as an instruction, Toddler must divide the instruction word into its ***opcode*** and address components.

4. *Execute the instruction.* Once the operation code and address field have been identified, the Toddler processor must carry out the steps necessary to perform the indicated action.

This cycle is repeated until a **HALT** instruction is executed or an error occurs.

Controlling the order of execution

The add-two-numbers program on page 5 runs through its instructions in a completely linear fashion, starting at the beginning and moving forward until it reaches the end. In order to perform more complex calculations, programs for the Toddler machine must be able to control the order in which instructions are executed. To do so, Toddler uses the **JUMP** instructions, which come in three forms. The **JUMP** instruction itself jumps unconditionally to the address stored in the instruction. The **JUMPZ** and **JUMPN** instructions are similar, but jump to the address only if a particular condition is met. The **JUMPZ** instruction jumps only if the value of the **AC** is zero; the **JUMPN** instruction jumps only if the value of the **AC** is negative.

As an example, suppose that you want to write a program that adds numbers entered by the user until the user enters 0 to mark the end of the input. In English, you can express the logic of such a program as the following series of steps:

1. Designate a memory location called **total** to record the total so far.
2. Designate a memory location called **value** to hold each value as it appears.
3. Initialize **total** to zero.
4. Use the **INPUT** instruction to read a number into **value**.
5. If **value** is zero, output the value in **total** and halt.
6. Add **value** to the contents of **total**.
7. Go back to step 4 to get another number.

In Toddler, you can easily write a simple program to execute this series of steps. That program looks like this in its purely numeric form:

```
(01) +111
(02) +250
(03) +851
(04) +151
(05) +609
(06) +350
(07) +250
(08) +503
(09) +950
(10) +500
(11) +000
```

This program uses a +609 instruction at address 05 to check whether the user entered a zero value. If so, the program jumps ahead to print out the result. It also uses a +503

instruction at address 08 to jump back to the instruction at address 03 that reads in the next value.

Assembly language

Although it is possible to trace the operation of the program to add a list of numbers in its numeric form, it is tedious to do so. While numeric representation makes it easier to store instructions in memory, it also makes it harder for programmers to understand what is going on. From the earliest days of computing, stretching back to Ada Lovelace's notes on the Analytical Engine, programmers have preferred to write instructions in a more human-readable form. Programs written in their numeric form are called ***machine language*** programs. Programs that use symbolic instruction names in place of the numeric opcodes are referred to as ***assembly language*** programs. Assembly language programs are much easier to read, but can nonetheless be translated directly into their machine language counterparts. An assembly language version of the program to add a list of integers looks like this:

```
start:  LOAD    zero
        STORE   total
loop:   INPUT   n
        LOAD    n
        JUMPZ   done
        ADD     total
        STORE   total
        JUMP    loop
done:   OUTPUT  total
        HALT

zero:   0
total:  0
n:      0
```

The most important assembly-language feature introduced in this program listing is the use of instruction names to represent particular opcodes. This makes it possible, for example, to write

```
HALT
```

at the end of the program in preference to the machine-language instruction

```
+500
```

The second new feature is the use of symbolic names—such as **`start`**, **`loop`**, **`done`**, **`zero`**, **`total`**, and **`n`**—to refer to specific addresses in the program. In assembly language, such names are known as ***labels***. Labels in Toddler are defined by writing a name followed by a colon, which defines that name as being equal to the current location in memory. For example, the first line sets the label **`start`** to 1, since this instruction is stored at address 01. Similarly, the symbol **`loop`** will be set to the value 3.

Labels may be used before they are defined; when the actual definition appears, the appropriate value is substituted back into any instructions that use it. Thus, when Toddler

gets to the line labeled **zero** at memory location 11, it not only defines the label **zero** to have the value 11, but also goes back and fills in 11 as the address part of the instruction

```
LOAD    zero
```

in memory location 01. Because **LOAD** has the operation code +1, the value of memory location 01 after loading the assembly language version of the program will be +111.

The last new feature in the assembly language program shows how to specify constant values in a Toddler program. In the English version of the program, the first operation after giving names to the data values is to set the variable **total** to zero. To do so, you could not simply write

```
LOAD    0
STORE   total
```

The **LOAD** instruction here will try to load the value in address 0, rather than the integer value 0, which is what the program needs. To work with a constant integer value, you need to put that constant in a memory word and then specify its address in the appropriate instruction. Here, for example, the instruction

```
LOAD    zero
```

loads from the address corresponding to the label **zero**, which is defined by the program to contain the value 0, as follows:

```
zero:   0
```

Because it is cumbersome to define all constants by putting them in a memory word and then using that address in other instructions, the Toddler assembler allows you to specify constants by writing a number sign (**#**) before an integer value, as in

```
LOAD    #0
```

What the assembler does when it encounters the number sign is

1. Find some unused address at the end of the program.
2. Put the constant value into that address.
3. Use the address of the constant in the instruction that contained the constant.

The **#** syntax therefore has exactly the same effect as storing the constant in a memory location and using that location's address. The advantage of using the **#** form is that the resulting program is easier to read.

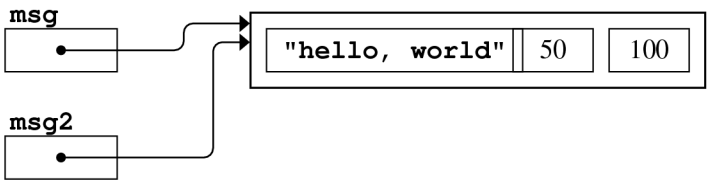
Takeaways for Python programmers

The computers that you are using to run Python are vastly larger and more sophisticated than Toddler. Even so, modern computers retain the basic von Neumann architecture and store values in memory cells that have an address. In Python, the address of an object in memory is called a ***reference.*** References are described briefly in Chapter 3, which emphasizes the fact that **GObject** values—and indeed all Python objects—are stored

using references. The example that appears in Chapter 3 diagrams the effect of the assignment statements

```
msg = GLabel("hello, world", 50, 100)
msg2 = msg
```

These statements do not create two **GLabel** objects but rather a single **GLabel** object whose address is stored in both the variables **msg** and **msg2**. Conceptually, the effect of references can be diagrammed using arrows that point from a variable to the object to which that variable refers, as follows:



As a Python programmer, you don't need to know the internal addresses of references, but it is important to understand that those references are there. And if you're curious, the standard implementation of Python allows you to determine the address of an object by calling the built-in function **id**. In the above diagram, for example, calling **id(msg)** would return the internal address at which the **GLabel** is stored. Calling **id(msg2)** would return the same value.

Understanding the concepts of memory and addresses may also help you to visualize how strings are stored in Python. Although support for the full Unicode range introduces some complications, the ASCII characters in a string are stored in consecutive memory locations. In the Toddler machine, the characters in the string **"hello, world"** would be stored in memory like this:

50	+1044	"h"
51	+1041	"e"
52	+1048	"l"
53	+1048	"l"
54	+1111	"o"
55	+0444	","
56	+0432	" "
57	+1119	"w"
58	+1111	"o"
59	+1114	"r"
60	+1048	"l"
61	+1000	"d"
62	+0000	

Toddler could then keep track of the entire string using the address 50 as a reference. The value 000 at address 62 marks the end of the sequence of characters. The underlying implementation of Python adopts much the same approach.