Evo Illustrates a Simplified Computer

By Richard Born
Associate Professor Emeritus
Northern Illinois University
rborn2@niu.edu

A Historical Note

ENIAC (***E***lectronic ***N***umerical ***I***ntegrator ***A***nd ***C***omputer), dedicated in the year 1946, was one of the first electronic, general-purpose computers ever constructed. Although its name *ENIAC* rhymes with the word *maniac*, it was not a lunatic. It could, never-the-less, do calculations on the order of one thousand times faster than existing electro-mechanical machines.

By the mid-1960's, there was a great deal of interest in teaching the fundamentals of the inner-workings of a computer to students at the high-school level. But how could this be accomplished at a time when these students could not get their hands on a real computer? Bell Telephone Labs came up with the idea of making a "computer" out of cardboard. They called it **cardiac**, short for ***card***board ***i***llustrative ***a***id to ***c***omputation. This *cardiac* had nothing to do with *heart*, though it may have reached the heart of many from a generation of high school students. It consisted of an instruction manual and a base-ten cardboard "computer", complete with bugs. See the photo in Figure 1.

Figure 1

By the late 1970's and early 1980's, ordinary people could actually purchase working electronic programmable computers—the TRS-80 (Tandy Radio Shack), Apple II, Atari, and Commodore. The author of this paper was a consultant for Tandy during that time and wrote educational courseware packages for its TRS-80. One of these was called ***The Illustrated Computer***. This package allowed students to learn interactions among the main components of a computer, including the input unit, output unit, control unit, memory, instruction

address register, and accumulator. In the process, students wrote structured machine language programs for a simple base-ten simulated computer and learned the basics of the fetch/execute cycle of computers. The use of base-ten allowed students to learn the inner-workings of a computer without needing to worry about the binary and hexadecimal number systems. Students could enter and run their programs, debug with memory dumps, and view program termination information including a trace of the last ten instructions executed. See Figure 2 for a photo related to this courseware package.

Figure 2

Fast Forward to Today

A third of a century after the TRS-80, we find ourselves in the midst of a robot revolution. Central to the K-12 educational applications of robots is Evollve's tiny Ozobot ***Evo***. Evo's small size coupled with its ability to follow lines, display LEDs of many colors, speak and be programmed make it perfect for teaching and learning the inner-workings of a simple computer. Evo can be programmed to animate the fetch/execute cycle of a computer, speaking the instruction codes, addresses and data being transferred between components of a computer. Meanwhile, LED color can be used to show which component of a computer is receiving a data item in a transfer. By its very nature, Evo goes way beyond animations that are widely available on the web—Evo is a *physical* device whose *motion* on a maze containing the fetch/execute cycle brings life and excitement to this cycle! See Figure 3.

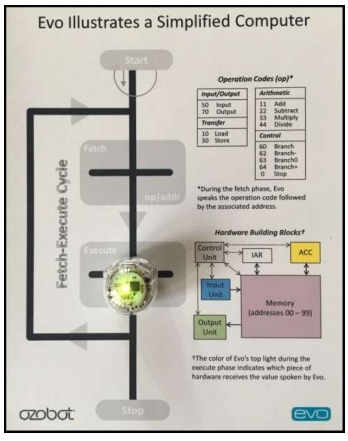


Figure 3

Introduction

This OzoBlockly program has been designed to introduce high school students to concepts related to machine language programming. This is accomplished by having Evo simulate a simplified computer which students program using a simplified machine language. Students will also learn fundamentals of structured flowcharting and programming in addition to writing program documentation to unify a machine language program and its structured flowchart.

Evo introduces students to machine language by providing *hands-on experience* with a simplified machine language. Evo simulates an easy to understand computer having memory, an accumulator, an instruction address register, an input unit, an output unit and a control unit.

One of the most important features of Evo's simplified computer is that it uses the base-ten number system. This means that students can learn machine language instructions including load/store, input/output, arithmetic, and branching without having to work with binary or hexadecimal number systems. Once the student has learned these basics using Evo, transferring these concepts to other real-world computer architectures should be relatively easy.

The Architecture of Evo's Simulated Computer

Figure 4 shows the basic components of Evo's simulated computer. These include the control unit, memory, input unit, output unit, accumulator (ACC), and instruction address register (IAR). The solid arrows show the direction in which numbers move between components. The dotted arrows connecting the control unit to the other components indicate its ability to both sense and transmit data and commands to them. Evo's memory consists of 100 locations each capable of storing integers between -128 and 127, inclusive. These memory locations have addresses ranging from 00 to 99, inclusive.

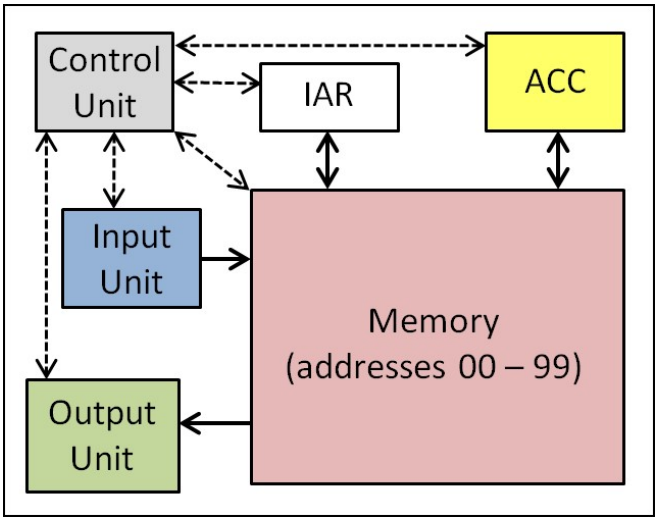


Figure 4

The color shown for each unit in Figure 1 has special meaning for Evo's simulation. Specifically, the color of Evo's top light indicates which component ***receives*** the value that is spoken by Evo when a particular machine instruction is executed. The functions of each of the six components will be discussed in the next section.

Functions of the Components of Evo's Simulated Computer

- *Control unit.* The control unit reads the program's instructions that are stored as numbers in memory, directing the other units to perform their operations in the proper sequence.
- *Memory.* The memory has been designed to hold signed integers ranging from -128 to 127. It is common terminology to refer to each memory location as a ***word***. Just as houses on a street are given an address, each word in memory also as an address. These addresses range from 00 through 99. The concept of *zero-index addressing*—starting with 00 instead of 01—is quite common in the computing field.
- *Input unit.* The input unit allows the movement of data from the outside world to memory. Evo's input will always be signed integers accomplished by pressing Evo's button. The details for entering signed integers is quite straight forward using Evo's button and will be discussed later.
- *Output unit.* The output unit provides for the movement of data from Evo's memory to the outside world. Output is always a number spoken by Evo in the range -128 to 127.
- *Accumulator (ACC).* With Evo's simulated computer, each instruction has just one operand associated with it. Arithmetic operations all require two operands. Therefore, one of these operands is always in memory and the other is in the accumulator. The results of any arithmetic operation will always be placed in the accumulator.
- *Instruction Address Register (IAR).* The instruction address register holds the address of the instruction that is currently being executed. The IAR is incremented at the completion of instruction execution so that it points to the next instruction in the sequence. The branch instructions, however, can modify the contents of the IAR so that normal sequencing does not occur.

The Instruction Format of Evo's Simulated Computer

Each of Evo's simulated computer instructions uses ***two*** of its memory words and is illustrated in Figure 5.

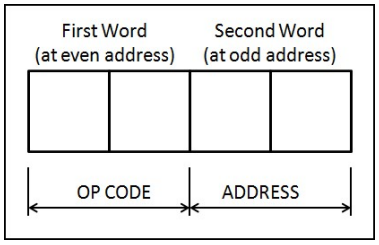


Figure 5

The operation code, or *op code* for short, will always be a two-digit positive integer and is in the first word of an instruction for Evo's simulated computer. It will always be at an ***even*** memory address, and the ***first instruction*** to be executed by an Evo simulated computer program must always be at address 00. The op code indicates what operation is to be executed. (For example, it might tell the simulated computer to ADD.) The operation codes will be explained in detail in the next section. The second word of the instruction is always at the odd address immediately following the first word of the instruction. The second word specifies the *address* or operand of the instruction. (For example, it might tell the computer where to find the value to add to the accumulator.)

The Instruction Set for Evo's Simulated Computer

Evo's simulated computer instruction set consists of a set of thirteen instructions. These are categorized into four groups: Input/Output, Transfer, Arithmetic, and Control. The tables of Figure 6, 7, 8, and 9 provide the details for each of these instructions.

Input/Output Instructions			
Instruction	*Op Code*	*Description*	*Example*
Input	50	The signed integer entered by the user with Evo's button is placed in the addressed location, destroying the previous content of the address location.	If the user entered the number -7 with Evo's button, then the instruction 5085 would write the number -7 to memory address 85.
Output	70	The numeric content of the address is spoken by Evo. The content of the addressed location remains unchanged by this instruction.	If the content of address 87 is 52, then the instruction 7087 would cause Evo to speak the number 52.

Figure 6

Transfer Instructions			
Instruction	*Op Code*	*Description*	*Example*
Load	10	The content of the addressed location is written to the accumulator, destroying the previous content of the accumulator. The addressed location is left unchanged.	If address 77 contains the number 125, then the instruction 1077 would write the number 125 into the accumulator.
Store	30	The content of the accumulator is written to the memory location specified by the address. The content of the accumulator is left unchanged, but the previous content of the addressed location is destroyed.	If the content of the accumulator is 27, then the instruction 3084 would cause Evo to write the number 27 to memory location 84.

Figure 7

Arithmetic Instructions*			
Instruction	*Op Code*	*Description*	*Example*
Add	11	The content of the addressed location is added to the content of the accumulator. The content of the addressed location is not changed.	If the accumulator contains the number 24 and address 91 contains the number 16, then the instruction 1191 would result in the accumulator containing the number 40 (24 + 16).
Subtract	22	The content of the addressed location is subtracted from the content of the accumulator. The content of the addressed location is not changed.	If the accumulator contains the number 24 and address 91 contains the number 16, then the instruction 2291 would result in the accumulator containing the number 8 (24 – 16).
Multiply	33	The content of the accumulator is multiplied by the content of the addressed location, and the resulting product replaces the value of the accumulator. The content of the addressed location is not changed.	If the accumulator contains the number 17 and address 91 contains the number 5, then the instruction 3391 would result in the accumulator containing the number 85 (17 * 5).
Divide	44	The content of the accumulator is divided by the content of the addressed location, with the result replacing the value of the accumulator. The result will be the greatest integer not greater than the true quotient. The content of the addressed location is not changed. Division by zero will cause Evo to flash a police light animation and then stop and turn off.	If the accumulator contains the number 27 and address 74 contains the number 4, then the instruction 4474 would result in the accumulator containing the number 6 (**⌊**27/4**⌋**).

**If the result of any arithmetic operation exceeds 127 or is less than -128, Evo will set the result to 127 or -128, respectively. So you need to be cautious whenever you get a result of 127 or -128. Is this really the result you want or has Evo experienced an overflow?*

Figure 8

Control Instructions†			
Instruction	*Op Code*	*Description*	*Example*
Branch	60	Control is unconditionally transferred to the instruction in the addressed location. The content of the addressed location is not altered.	The instruction 6026 would transfer control to the instruction at addresses 26 and 27.
Branch Negative	62	If the content of the accumulator is less than zero, then control is transferred to the addressed location. If the content of the accumulator is greater than or equal to zero, then control is transferred to the instruction that is next in sequence after the Branch Negative instruction.	The instruction 6218 at addresses 28 and 29 would transfer control to the instruction at addresses 18 and 19 if the content of the accumulator is negative. Otherwise it would transfer control to the instruction at addresses 30 and 31.
Branch Zero	63	If the content of the accumulator is equal to zero, then control is transferred to the addressed location. If the content of the accumulator is not equal to zero, then control is transferred to the instruction that is next in sequence after the Branch Zero instruction.	The instruction 6318 at addresses 28 and 29 would transfer control to the instruction at addresses 18 and 19 if the content of the accumulator is zero. Otherwise it would transfer control to the instruction at addresses 30 and 31.
Branch Positive	64	If the content of the accumulator is positive, then control is transferred to the addressed location. If the content of the accumulator is less than or equal to zero, then control is transferred to the instruction that is next in sequence after the Branch Positive instruction.	The instruction 6418 at addresses 28 and 29 would transfer control to the instruction at addresses 18 and 19 if the content of the accumulator is positive. Otherwise it would transfer control to the instruction at addresses 30 and 31.
Stop	00	This will stop program execution and turn Evo off.	In the instruction 00XX, XX is usually 00 by convention.
*†All of the branch instructions modify the contents of the instruction address register (IAR). If the branch condition is true, then the IAR is modified to contain the value of the address word of the modifying instruction. If the branch condition is false, then the IAR is incremented by 2. (Remember that each instruction is **two words** in length for Evo's simulated computer and must start at an even address.)*			

Figure 9

Evo Simulated Computer: Program to Add Two Numbers

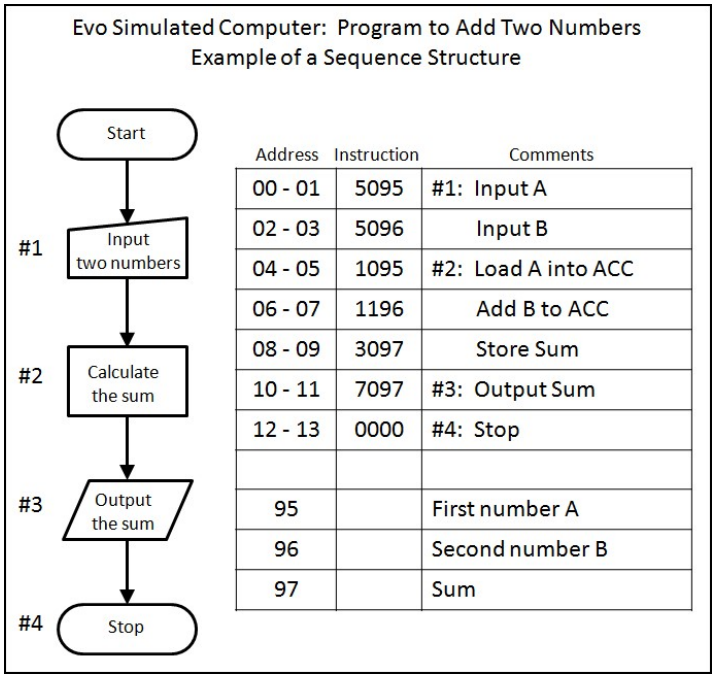
There are three major constructs involved in creating structured programs:

- *sequence*—sequential processing of instructions, one after the other in memory
- *selection*—involving decisions based upon a condition
- *iteration*—looping or repeating a process.

In this section, we will investigate the simplest structure—sequence. In a sequence structure, processes are executed one after another, in sequence. We will study a simple program to input two numbers, compute their sum, and then output the sum. The flowchart consists of four numbered steps, shown in Figure 10. The corresponding Evo simulated computer program is also shown in the figure. The program shows the memory address locations where each instruction is found, with the requirement that *the first instruction in an Evo*

simulated computer program be located in addresses 00 and 01. Also included are English-like comments that document each of the machine language instructions and relate the program to the flowchart. Such comments are essential to preserve the relationship between the abstract low-level machine instructions and what they accomplish in the real world.

The first instruction, 5095, inputs the first number into address 95. The second instruction, 5096, executed in sequence after the first instruction, inputs the second number into address 96. The instruction 1095 then loads the first number into the accumulator. Next, instruction 1196 adds the second number to the accumulator, resulting in the sum being in the accumulator. The instruction 3097 stores the sum at address 97, and the instruction 7097 outputs the sum to the output unit. Finally, the instruction 0000 stops the program and turns Evo off. The variables A, B, and Sum could be anywhere in memory not holding instructions. Note that all 100 memory locations are initialized at the value zero before program execution begins.



Address	Instruction	Comments
00 - 01	5095	#1: Input A
02 - 03	5096	Input B
04 - 05	1095	#2: Load A into ACC
06 - 07	1196	Add B to ACC
08 - 09	3097	Store Sum
10 - 11	7097	#3: Output Sum
12 - 13	0000	#4: Stop
95		First number A
96		Second number B
97		Sum

Figure 10

Running the Evo Simulated Computer Program to Add Two Numbers

1. Open the program *SimulatedComputerAdd2Numbers.ozocode* while at the OzoBlockly web site. After scrolling the workspace if necessary, you will see a subroutine called “Define Program”. You notice that the simulated program’s instructions have already been keyed in by the user into an array representing the simulated computer’s memory, starting at memory location 00 (shown as element 0 in the array). Addresses 00 and 01 contain 50 and 95, respectively, and so on through the rest of the program. See Figure 11. (When you are ready to start defining a program from scratch, then you would open the program *SimulatedComputerTemplate.ozocode* and simply key in your instructions into the appropriate addresses in the array.)

2. Now load the program *SimulatedComputerAdd2Numbers.ozocode* into Evo, place Evo on the start location on the "Evo Illustrates a Simplified Computer" Ozomap found on the final page of this document, and start the program.
3. Evo will stop at the fetch location on the map and speak the instruction "fifty…ninety five" on his speaker to let you know that the simulated computer has just fetched the first instruction in your program.
4. Evo will then move to the execute location on the map and give you the opportunity to enter the value for the first number for your addition problem. Let's say that you want this first number to be -7. Whenever the input unit wants you to key in a value, Evo begins by showing a red LED for three seconds. If the number you want to enter is negative, as in our case for the first number, then you have three seconds to push Evo's button once. Upon pushing the button Evo will flash white very quickly to let you know that your negative sign has been recognized. *If you don't press the button during this three second interval, Evo will take the number to be a positive number.* Evo's top LED will then turn deep blue for 12 seconds. During this 12-second interval you would push Evo's button 7 times, each time receiving an acknowledgement white LED flash.

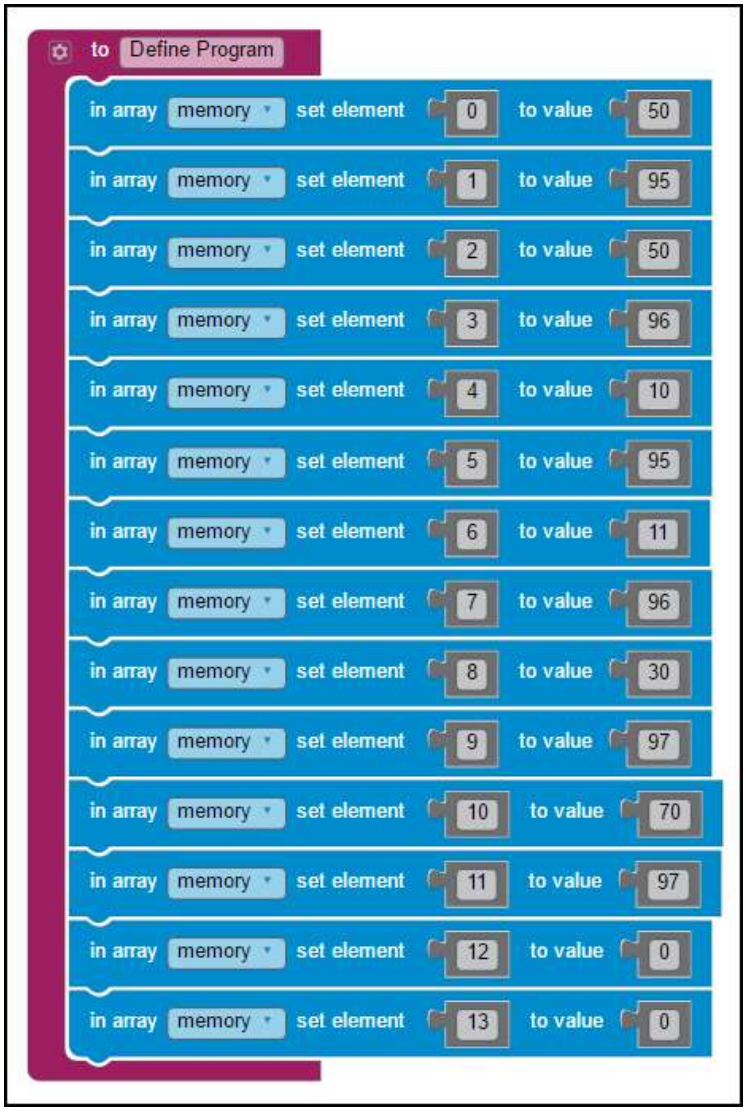


Figure 11

5. At the end of the 12-second interval, Evo will speak the number you entered "minus seven" while the LED is red. When Evo speaks the number, the red LED means that the number is being written to the requested *memory* location (in this case address 95). Note that ***red*** is the color of ***memory*** in the "Hardware Building Blocks" diagram on the Ozomap.
6. Evo will then cycle back up to the fetch portion of the cycle, fetch the next instruction 5096, and speak "fifty ninety-six" so that you know which instruction was fetched. He will then move to the execute portion of the cycle, where you would enter the second number in your addition problem. Let's say that this number is 9. Since the number is positive, you would not press the button when the LED is red, but when it turns deep blue you would have twelve seconds to press the button 9 times. After the 12-second interval is over, Evo will speak "nine" to acknowledge the value you entered. While speaking, the LED will again be red, letting you know that the number you entered is being written to the requested *memory* location (in this case 96).

   With only twelve seconds to press Evo's button, you will probably be limited to entering numbers from about -10 to 10. This may seem quite limiting, but the purpose of the Evo simulated computer is to get you to understand the inner-workings. This can be accomplished even with limited input capabilities.

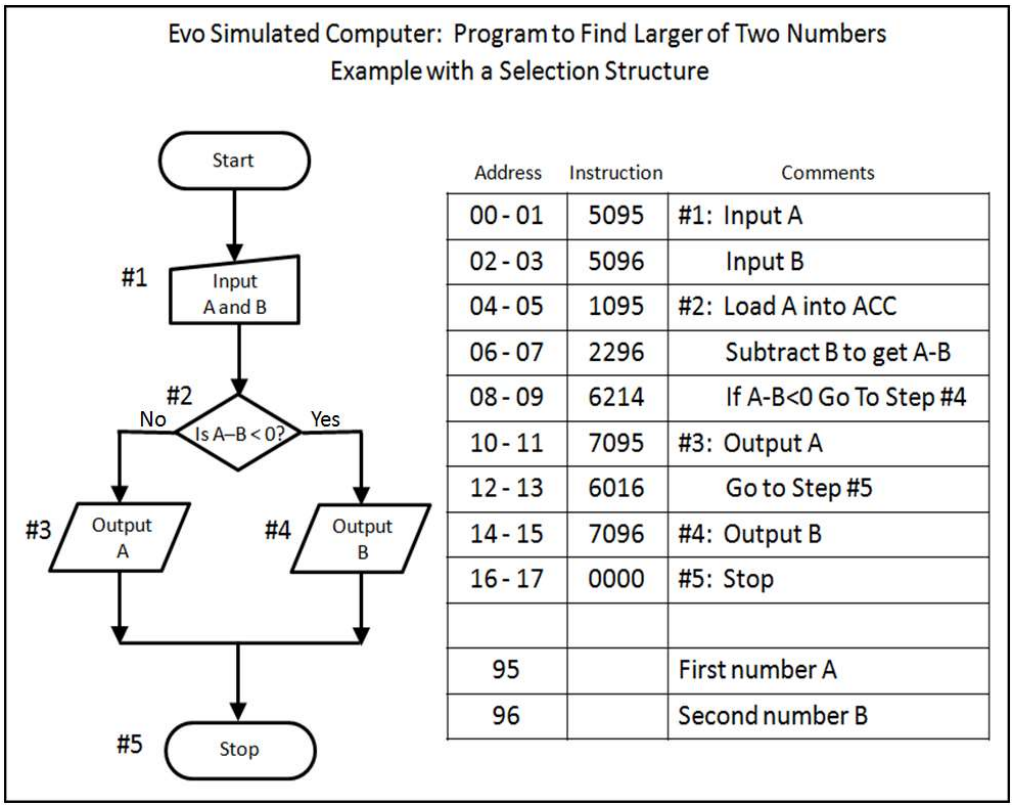
7. Evo will cycle back to fetch and speak the instruction 1095 by saying "ten…ninety-five". He will then move to execute and speak "minus seven" while the top LED is yellow. Yellow indicates that the value -7 is being written to the accumulator, shown as yellow in the "Hardware Building Blocks" diagram on the map.
8. Evo will cycle back to fetch and speak the instruction 1196 by saying "eleven…ninety-six". He will then move to execute and speak "two" (-7 + 9) after adding 9 to the accumulator. Again the top LED will be ***yellow***, since the content of the *accumulator* has changed.
9. Evo will cycle back to fetch and speak the instruction 3097 by saying "thirty…ninety-seven" . He will then move to execute and speak "two" while writing the sum 2 to address 97. While speaking, the top LED will be red, as red is the color of memory on the "Hardware Building Blocks" diagram.
10. Evo will cycle back to fetch and speak the instruction 7097 by saying "seventy…ninety-seven". He will then move to execute and speak the sum "two". While speaking, the top LED will be green, as green is the color of the output unit on the "Hardware Building Blocks" diagram.
11. Finally, Evo will fetch the instruction 0000, indicating that the program is completed. Evo will move to the "Stop" location on the Ozomap and power down.

There is a complete video (about 3 minutes in length) of Evo's movements, display colors, and speach accompanying this document.

Evo Simulated Computer: Program to Find the Larger of Two Numbers

This program provides a simple example with a selection structure and is shown in Figure 12. In a nutshell, the program inputs two numbers and outputs the larger of the two numbers, as shown in the flowchart. The instruction 6214, whose op code is 62—Branch-, looks at the difference A-B. If it is negative, then B is the larger and control is transferred to the instruction at address 14. If it is positive, then A is the larger and control continues with the instruction at addresses 10. When Evo reaches the execute portion of the map,

Evo will display a white light (the color of the IAR in the "Hardware Building Blocks" diagram) and speak the even address (either 10 or 14) of the instruction to be executed next.



Address	Instruction	Comments
00 - 01	5095	#1: Input A
02 - 03	5096	Input B
04 - 05	1095	#2: Load A into ACC
06 - 07	2296	Subtract B to get A-B
08 - 09	6214	If A-B<0 Go To Step #4
10 - 11	7095	#3: Output A
12 - 13	6016	Go to Step #5
14 - 15	7096	#4: Output B
16 - 17	0000	#5: Stop
95		First number A
96		Second number B

Figure 12

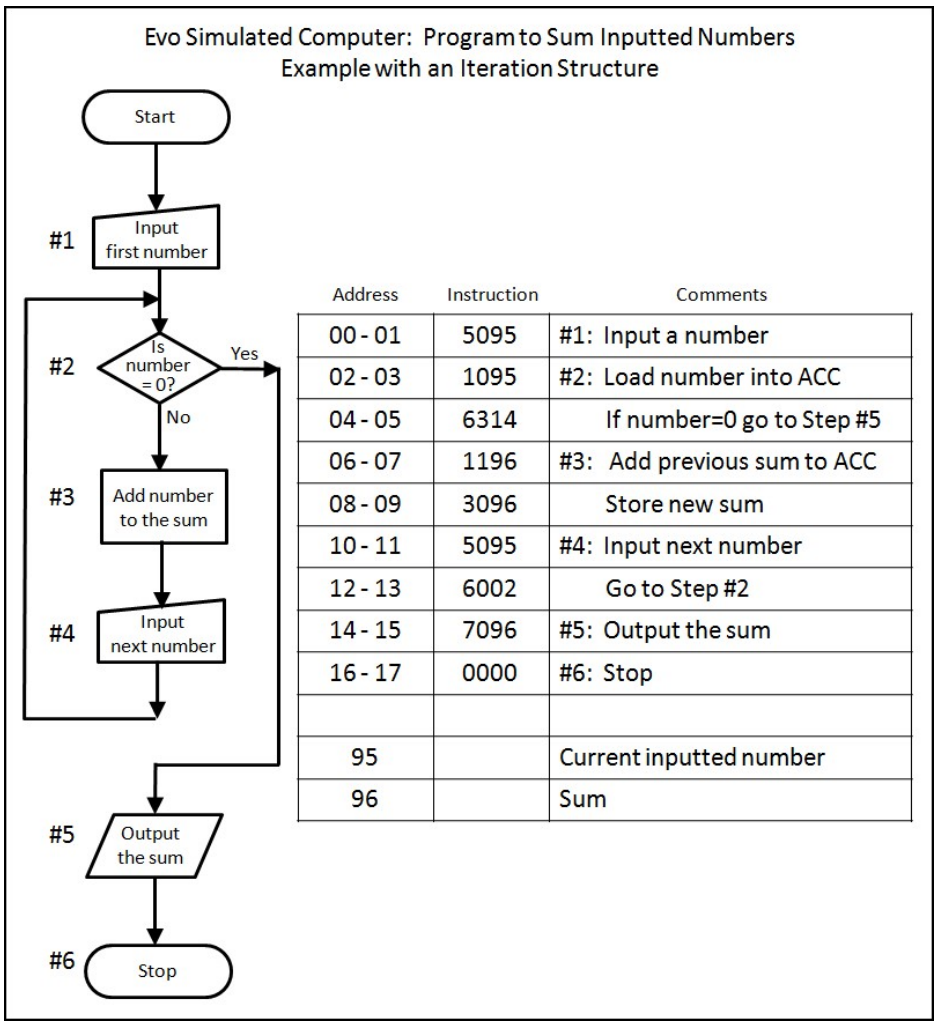
The instruction 6016, whose op code is 62—Branch, provides an unconditional transfer to the stop instruction at address 16. When Evo reaches the execute portion of the map, a white LED is displayed while Evo speaks the address 16 of the next instruction to be executed. The file *SimulatedComputerFindLargerOfTwoNumbers.ozocode*, accompanying this document, can be loaded into Evo and run to view the fetch/execute cycle animation for this program.

Evo Simulated Computer: Program to Sum Inputted Numbers

This program provides a simple example with an iteration structure and is shown in Figure 13. The program inputs any number of numbers and then outputs the sum of those numbers, as shown in the flowchart.

In this example, we agree to flag the end of the process of inputting numbers by inputting a zero. This explains why the decision symbol uses Branch0. Structured looping insists that the loop must go back to the decision symbol in the flowchart where the loop begins. This makes it necessary to input the second and any succeeding numbers at the end of the loop (Step #4) in the flowchart) before looping back to the decision symbol (Step #2). The initial input (Step #1) is sometimes referred to as the priming input. Generally speaking, the value of the variable in the decision symbol at the start of the loop must be altered somewhere within the loop. Otherwise, the loop, once entered, will never terminate. In our example program, Step #4 alters the value of "number" by inputting a new number.

The file *SimulatedComputerSumInputtedNumbers.ozocode,* accompanying this document, can be loaded into Evo and run to view the fetch/execute cycle animation for this program.



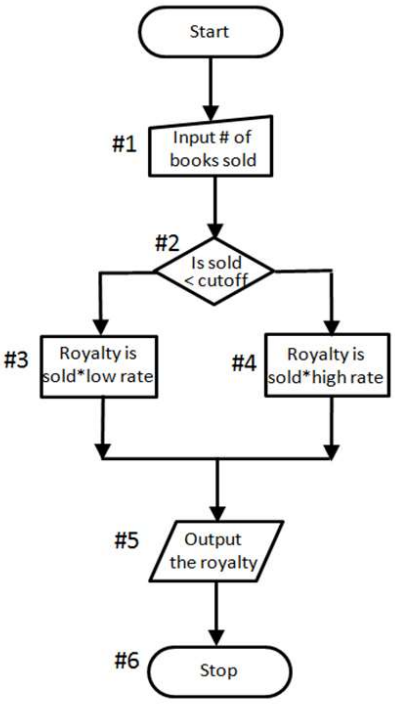
Address	Instruction	Comments
00 - 01	5095	#1: Input a number
02 - 03	1095	#2: Load number into ACC
04 - 05	6314	If number=0 go to Step #5
06 - 07	1196	#3: Add previous sum to ACC
08 - 09	3096	Store new sum
10 - 11	5095	#4: Input next number
12 - 13	6002	Go to Step #2
14 - 15	7096	#5: Output the sum
16 - 17	0000	#6: Stop
95		Current inputted number
96		Sum

Figure 13

Evo Simplified Computer: Program to Calculate Royalty

Now let's look at a practical business problem. Imagine that a publisher pays royalties to its authors in such a way that if a book sells fewer than 10 copies, then a $4 royalty is paid per copy sold. On the other hand, if 10 or more books are sold, then the royalty is $7 per copy. Our program should input the number of copies sold, calculate the royalty, and output the royalty. The structured flowchart and Evo simulated computer program are shown in Figure 14. This program illustrates how to introduce constants into our programs. Note that address 81, 82, and 83 contain the constants 10, 4, and 7, respectively, that are needed to process the input number of books sold. The number of books sold must be compared to the cutoff value of 10, and then multiplied by either 4 or 7, as appropriate, to determine the royalty amount. The instructions 1095 and 3383 calculate the royalty amount if 10 or more books are sold. The instruction 6018 causes an unconditional branch over the case where fewer than 10 books are sold.

Evo Simulated Computer: Program to Calculate Royalty
Example with Constants



Address	Instruction	Comments
00 - 01	5095	#1: Input # of books sold
02 - 03	1081	#2: Load cutoff point into ACC
04 - 05	2295	Subtract # of books sold
06 - 07	6414	If #sold <cutoff go to step #4
08 - 09	1095	#3: Load #books into ACC
10 - 11	3383	Multiply by high rate
12 - 13	6018	Go to Step #5
14 - 15	1095	#4: Load #books sold
16 - 17	3382	Multiply by high rate
18 - 19	3096	#5: Store royalty amount
20 - 21	7096	Output royalty amount
22 - 23	0000	#6: Stop
95		Variable: # of books sold
96		Variable: Royalty amount in $
81	10	Constant: Cutoff point in books
82	4	Constant: Low rate in $
83	7	Constant: High rage in $

Figure 14

The file *SimulatedComputerCalculateRoyalty.ozocode,* accompanying this document, can be loaded into Evo and run to view the fetch/execute cycle animation for this program. Figure 15 contains the last part of the OzoBlockly "Define Function" subroutine. This figure shows the last two instructions in the program (7096 and 0000) followed by assigning the constants 10, 4, and 7 to memory locations 81, 82, and 83, respectively. It should be noted that any memory locations not containing the actual program could have been used to store the constants.

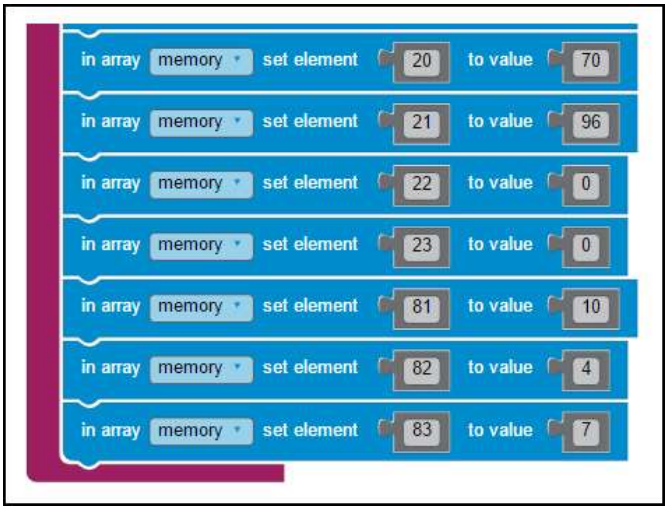


Figure 15