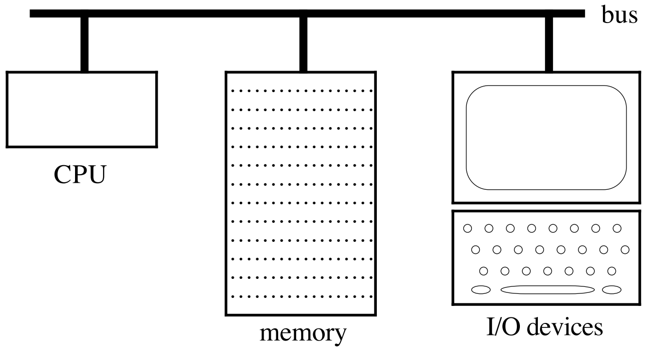
Introduction to MiniSim
A Simple von Neumann Machine

Programming languages—like C, C++, Java, or even Karel—are called **high-level languages** because they allow you to express algorithmic processes at a high level of abstraction. The advantage of such languages is that they enable you to focus on the concepts of algorithmic design and problem solving 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 features of high-level languages—such as the concepts of arrays and pointers—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. The structure of any modern computer is much too complex to cover in an introductory course. It is therefore traditional to introduce the concepts of machine architecture using a hypothetical machine. This handout describes MiniSim, a very simple machine that is nonetheless powerful enough to illustrate the basic concepts of digital computer architecture.

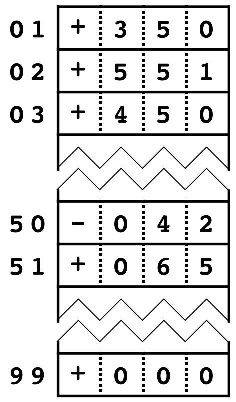
Overview of the MiniSim architecture

Although modern computers differ significantly in their internal architecture, most computer systems in use today share the same basic organizational structure. The typical computer consists of a central processing unit (CPU), some amount of memory, and some connection to input/output (I/O) devices so that the machine can communicate with the outside world. These components are connected by a bus so that data can flow between the separate units, as shown in the diagram below:



The memory of a computer system is typically broken down into individual memory cells of a fixed size. Each individual cell is called a memory **word** and is identified by a numeric **address.** MiniSim has 100 words of memory, numbered from address 00 to 99. To make it easier to catch various common programming errors, MiniSim makes it illegal to use address 00, which means that the usable memory of the machine actually begins at address 01.

Each of MiniSim's memory words contains a integer consisting of three decimal digits plus a sign, so that the range of each memory word is –999 to +999. The following diagram therefore shows a possible configuration of MiniSim's memory, though displaying only a few words:



In the diagram, the word at address 01 contains the number +350, the word at address 50 contains the number –42, and so forth.

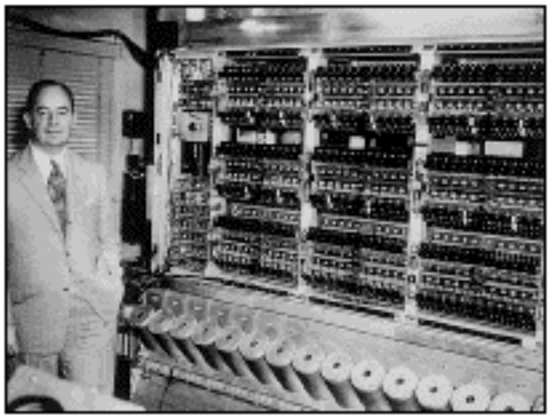
Even though MiniSim memory locations always contain three-digit signed integers, it is important to recognize that you can interpret those integers in different ways. For example, the value 65 in location 51 might be a decimal number or the ASCII character code for the character `'A'`. Each of these values has the internal representation 65. The correct interpretation depends on how the value is used.

In addition to the addressable memory, most computers include several memory cells that live in faster memory inside the processor. These special memory cells are generally called **registers.** MiniSim has the following five registers:

`AC`	the accumulator
`PC`	the program counter
`IR`	the instruction register
`XR`	the index register
`SP`	the stack pointer

Each of these registers has is own purpose, to be introduced in context.

The stored programming concept



In the earliest days of computing, programs were usually represented in a form that made them entirely separate from data. Typically, instructions were punched on paper tape and then fed into the machine, which would execute the instructions in sequence. If you wanted to change the program, you had to punch a new tape. One of the most important characteristics of modern computers is that the same memory words used to store data value can also be used to represent instructions. Machines that store their instructions internally as data are called **von Neumann machines,** after the mathematician John von Neumann (1903-1957), shown here with the machine he developed at the Institute for Advanced Study in Princeton.

In order to represent instructions inside a machine, there needs to be some encoding scheme that allows the hardware to interpret the contents of some memory location as an executable instruction. In the MiniSim machine, instructions are encoded in two separate parts. The first digit of the memory word—along with the sign—indicates what instruction is being performed, and the last two digits indicate an address in memory. Thus, an instruction in MiniSim is logically divided into the following fields:

operation code	*address field*

For example, if MiniSim were to execute the value in address 01 above as an instruction, it would take the value +350 and breaks it down into two components:

+	3	5	0

The first digit—along with the sign if necessary—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. In this case, the **`+3`** specifies an instruction called **`LOAD`**, and the **`LOAD`** instruction will operate on the word at address 50.

Figure 1. Program to add two integers

`(01)  +150`	`INPUT   50`
`(02)  +151`	`INPUT   51`
`(03)  +350`	`LOAD    50`
`(04)  +551`	`ADD     51`
`(05)  +452`	`STORE   52`
`(06)  +252`	`OUTPUT  52`
`(07)  +700`	`HALT`

Simple instructions by example

You can write a wide range of programs using just a few simple operations. To illustrate the operation of MiniSim, let's start with the following set of instructions:

code	name	operation
+1	**INPUT** *xx*	Reads a line containing an integer value from the terminal and stores it in memory address *xx*.
+2	**OUTPUT** *xx*	Displays the value in memory address *xx*.
+3	**LOAD** *xx*	Loads the **AC** with the value in the memory address *xx*. The contents of the memory word are not changed.
+4	**STORE** *xx*	Stores the value in the **AC** into memory address *xx*. The value in the **AC** remains unchanged
+5	**ADD** *xx*	Adds the value in memory address *xx* to the contents of the **AC**. The result is stored in the **AC**.
+6	**SUB** *xx*	Subtracts the value in memory address *xx* from the **AC**. As with **ADD**, the result is stored in the **AC**.
+7	**HALT**	Halts the simulated machine.

This set is sufficient to write a simple program that reads two numbers from the user, adds them together, and then displays the result. The code for the MiniSim version of the add-two-numbers program is shown in Figure 1. In this figure, the code on the left shows the addresses in memory where the instructions reside and the actual numeric values of the instructions. Instructions written in their numeric form represent the **machine language** version of the program. The equivalent code on the right is the **assembly language** version, which uses symbolic instruction names in place of the numeric operation codes. The assembly language form is much easier for humans to read, but it can easily be translated into its machine language counterpart, and vice versa.

The first line of the program is the instruction +150, which has the assembly language representation

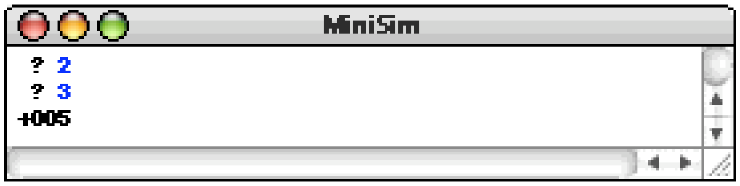
```
INPUT   50
```

When the MiniSim machine encounters this instruction, it types out a question mark and then waits for the user to enter a number on the keyboard. When the user hits the RETURN key to signal the end of the number, the MiniSim machine 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 are called **variables.**

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 number together. In MiniSim, 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 instructions

```
LOAD   50
ADD    51
STORE  52
```

From here, all the program needs to do is use an `OUTPUT` instruction to display the result in location 52. The program then moves on to the `HALT` instruction at location 7, which stops its execution. At the end of the program, the display screen might look like this:



```
 ? 2
 ? 3
+005
```

The instruction cycle

By convention, all MiniSim programs begin at address 01. To make sure that instructions are executed in their proper order, MiniSim—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. MiniSim uses another internal register, called the **instruction register** or `IR`, to hold the actual three-digit instruction word.

For each instruction, MiniSim executes the following instruction cycle:

1. *Fetch the current instruction.* In this phase of the instruction, MiniSim 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`, MiniSim 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, MiniSim must divide the instruction word into its operation-code and address-field components.
4. *Execute the instruction.* Once the operation code and address field have been identified, the MiniSim 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

Although the instructions you have seen so far make it possible to write many simple programs, they do not allow you to change the order in which instructions are executed. To do so, you need the following additional instructions:

code	name	operation
+7	**JUMP** *xx*	Sets the **PC** to *xx*, meaning that the next instruction will be taken from that memory location.
+8	**JZERO** *xx*	If the **AC** contains zero, this instruction transfers control to address *xx,* just like a **JUMP** instruction. If the value in the **AC** is nonzero, the program continues with the next instruction in sequence.
+9	**JNEG** *xx*	This instruction is like **JZERO**, except that it jumps to *xx* only if the value in the **AC** is negative.

Note that the operation code for **JUMP** is the same as that for **HALT**. MiniSim can distinguish the two instructions because the **HALT** instruction does not need an address field. Thus, MiniSim interprets the value **+700** as a **HALT** instruction and any other **+7** operation as a **JUMP**.

As an illustration of how these control instructions work, suppose that you want to write a program that adds a series of numbers, where the input is terminated by a zero. In English, you can express the logic of such a program as the following series of steps:

1. Designate a memory location to record the total so far. Call that location total.
2. Designate a second 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 the current value to the contents of total.
7. Go back to step 4 to get another number.

In MiniSim, you can easily write a simple program to execute this series of steps. That program appears in its machine language version in Figure 2, and in the equivalent assembly language program in Figure 3. The assembly language version of the program, which is stored in the file addlist.asm, illustrates several new features of the MiniSim assembler, which translates the programs you write into the internal machine-language code. The first is the use of comments, which are annotations written for human readers of the program. Comments start with the characters /* and end with the characters */, just as they do in Karel, C, C++, or Java.

Figure 2. Program to add a list of integers in MiniSim (machine language version)

```
(01)  +312
(02)  +413
(03)  +114
(04)  +314
(05)  +810
(06)  +313
(07)  +514
(08)  +413
(09)  +703
(10)  +213
(11)  +700
(12)  +000           This location is used to hold the constant 0.
(13)  +000           This location is used to keep track of the running total.
(14)  +000           This location is used to hold each input value.
```

Labels

The second feature introduced in Figure 3 is the use of symbolic names—such as **`start`**, **`loop`**, **`done`**, **`total`**, **`n`**, and **`zero`**—to refer to specific addresses in the program. In assembly language, such names are known as **labels**. Labels in MiniSim 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 defines the symbol **`start`** to be 1, since this instruction is being placed in location 1. Similarly, the symbol **`loop`** will have

Figure 3. Program to add a list of integers in MiniSim (assembly language version)

```
/*
 * File: addlist.asm
 * -----------------
 * This program adds a list of integers.  The user signals the
 * end of the input by entering the value 0.
 */

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

/* Constants */

zero:   0

/* Variables */

total:  0
n:      0
```

the value 3. Labels may be used before they are defined; when the actual definition appears, the appropriate value will be substituted back into any instructions that use it. Thus, when MiniSim gets around to the line labeled **`zero`** at memory location 12, it not only defines the label **`zero`** to have the value 12, but also goes back and fills in 12 as the address part of the instruction

```
LOAD    zero
```

in memory location 1. Because **`LOAD`** has the operation code +3, the value of memory location 1 after loading the assembly language version of the program will be +312, just as it is shown in the machine language version.

Constants

The final important concept to take from Figure 3 is how to specify constant values in an MiniSim program. In the English version of the program, the first actual 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 MiniSim assembler allows you to specify constants by writing a number sign (**`#`**) before an integer value, as in

```
LOAD    #0
```

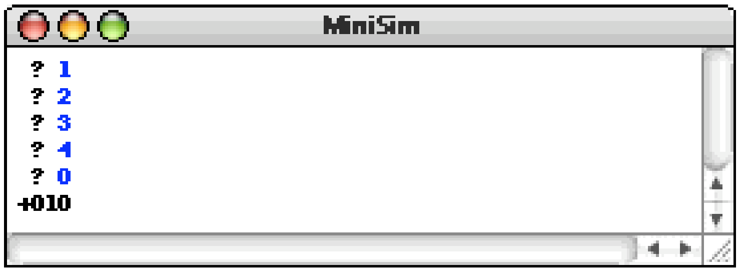
What the assembler does when it encounters the number sign is

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.

Playing computer

To get a better sense of how von Neumann machines work, it is important to go through the operation of a program on your own to make sure you can execute all the machine instructions. As an example, follow through the logic of the **`addlist.asm`** program from Figure 3 using the input values 1, 2, 3, 4, and 0. The output of that program would look like this:



```
 ? 1
 ? 2
 ? 3
 ? 4
 ? 0
+010
```

The important thing here is understanding how the instructions in the program accomplish the task.

Using the MiniSim applet

If you want to play with the ideas from MiniSim more actively, follow the link from my home page. This applet allows you to create and run programs for the MiniSim machine.

Examples

1. MiniSim program to count backward from 10 to 0

```
/*
 * File: countdown.asm
 * -------------------
 * This program counts backwards from 10 to 0
 */

start:  LOAD #10
loop:   STORE i
        OUTPUT i
        SUB #1
        JNEG done
        JUMP loop
done:   HALT

i:      0
```

2. The Fibonacci sequence

In the Fibonacci sequence, the first two terms are 0 and 1 and every subsequent term is the sum of the preceding two. If you number the terms beginning at 0, the first few terms of the Fibonacci sequence are $F_0 = 0$, $F_1 = 1$, $F_2 = 1$, $F_3 = 2$, $F_4 = 3$, $F_5 = 5$, $F_6 = 8$,… A MiniSim program that reads in an integer **`n`** and writes out the values of the Fibonacci sequence from F_0 through F_n can simulate the operation of the following Java method:

```
public void run() {
   int n = readInt(" ? ");
   int t1 = 0;
   int t2 = 1;
   for (int i = 0; i < n; i++) {
      println(t1);
      t3 = t1 + t2;
      t1 = t2;
      t2 = t3;
   }
}
```

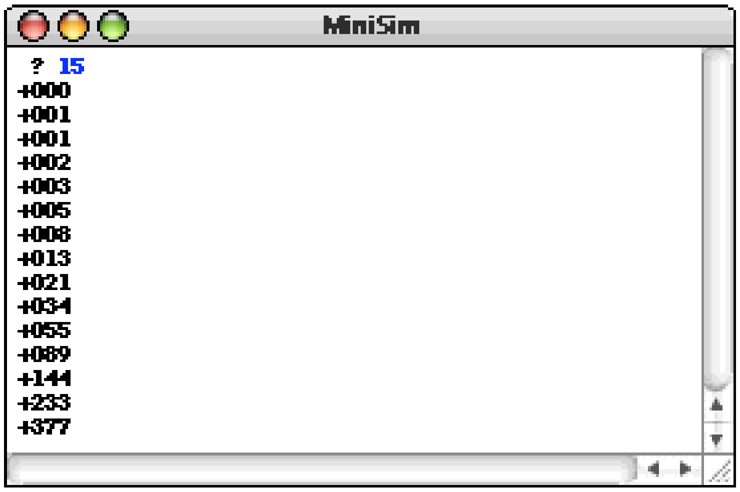
Here is the MiniSim program:

```
/*
 * File: Fibonacci.asm
 * -------------------
 * This program writes out the first n Fibonacci numbers.
 */

start:  INPUT    n         /* n = readInt(" / ");      */
        LOAD     #0        /* t1 = 0;                  */
        STORE    t1
        STORE    i         /* i = 0;                   */
        LOAD     #1
        STORE    t2        /* t2 = 1;                  */
loop:   LOAD     n         /* if (i == n)              */
        SUB      i
        JZERO    done      /* exit the loop            */
        OUTPUT   t1        /* println(t1);             */
        LOAD     t1        /* t3 = t1 + t2;            */
        ADD      t2
        STORE    t3
        LOAD     t2        /* t1 = t2;                 */
        STORE    t1
        LOAD     t3        /* t2 = t3;                 */
        STORE    t2
        LOAD     i         /* i++                      */
        ADD      #1
        STORE    i
        JUMP     loop      /* go back to start of loop */
done:   HALT

n:      0
i:      0
t1:     0
t2:     0
t3:     0
```

The program, for example, generates this sample run:



```
 ? 15
+000
+001
+001
+002
+003
+005
+008
+013
+021
+034
+055
+089
+144
+233
+377
```