5 Computer Architecture

Form ever follows function.
—Louis Sullivan (1856–1924), architect

Form IS function.
—Ludwig Mies van der Rohe (1886–1969), architect

This chapter is the pinnacle of the "hardware" part of our journey. We are now ready to take all the chips that we built in chapters 1–3 and integrate them into a general-purpose computer capable of running stored programs written in the machine language presented in chapter 4. The specific computer we will build, called *Hack*, has two important virtues. On the one hand, Hack is a simple machine that can be constructed in just a few hours, using previously built chips and the hardware simulator supplied with the book. On the other hand, Hack is sufficiently powerful to illustrate the key operating principles and hardware elements of any digital computer. Therefore, building it will give you an excellent understanding of how modern computers work at the low hardware and software levels.

Following an introduction of the *stored program* concept, section 5.1 gives a detailed description of the *von Neumann architecture*—a central dogma in computer science underlying the design of almost all modern computers. The Hack platform is one example of a von Neumann machine, and section 5.2 gives its exact hardware specification. Section 5.3 describes how the Hack platform can be implemented from available chips, in particular the ALU built in chapter 2 and the registers and memory systems built in chapter 3.

The computer that will emerge from this construction will be as simple as possible, but not simpler. This means that it will have the minimal configuration necessary to run interesting programs and deliver a reasonable performance. The comparison of this machine to typical computers is taken up in section 5.4, which emphasizes the critical role that *optimization* plays in the design of industrial-strength computers, but not in this chapter. As usual, the simplicity of our approach has a purpose: All the chips mentioned in the chapter, culminating in the Hack computer itself, can be built and tested on a personal computer running our hardware simulator, following the technical instructions given in section 5.5. The result will be a minimal yet surprisingly powerful computer.

5.1 Background

5.1.1 The Stored Program Concept

Compared to all the other machines around us, the most unique feature of the digital computer is its amazing versatility. Here is a machine with finite hardware that can perform a practically infinite array of tasks, from interactive games to word processing to scientific calculations. This remarkable flexibility—a boon that we have come to take for granted—is the fruit of a brilliant idea called the *stored program* concept. Formulated independently by several mathematicians in the 1930s, the stored program concept is still considered the most profound invention in, if not the very foundation of, modern computer science.

Like many scientific breakthroughs, the basic idea is rather simple. The computer is based on a fixed hardware platform, capable of executing a fixed repertoire of instructions. At the same time, these instructions can be used and combined like building blocks, yielding arbitrarily sophisticated programs. Moreover, the logic of these programs is not embedded in the hardware, as it was in mechanical computers predating 1930. Instead, the program's code is stored and manipulated in the computer memory, *just like data*, becoming what is known as "software." Since the computer's operation manifests itself to the user through the currently executing software, the same hardware platform can be made to behave completely differently each time it is loaded with a different program.

5.1.2 The von Neumann Architecture

The stored program concept is a key element of many abstract and practical computer models, most notably the *universal Turing machine* (1936) and the *von Neumann machine* (1945). The Turing machine—an abstract artifact describing a deceptively simple computer—is used mainly to analyze the logical foundations of

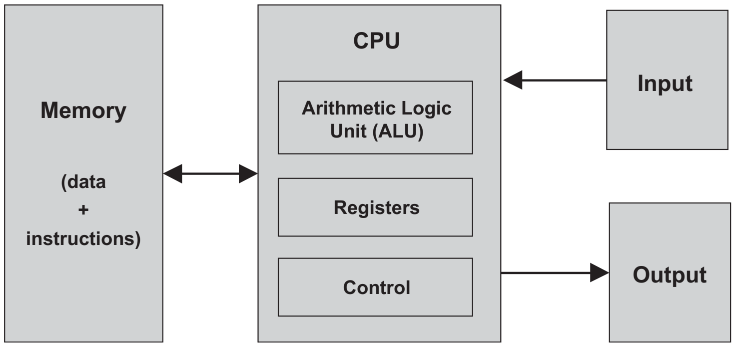


Figure 5.1 The von Neumann architecture (conceptual). At this level of detail, this model describes the architecture of almost all digital computers. The program that operates the computer resides in its memory, in accordance with the "stored program" concept.

computer systems. In contrast, the von Neumann machine is a practical architecture and the conceptual blueprint of almost all computer platforms today.

The von Neumann architecture is based on a *central processing unit* (CPU), interacting with a *memory* device, receiving data from some *input* device, and sending data to some *output* device (figure 5.1). At the heart of this architecture lies the stored program concept: The computer's memory stores not only the data that the computer manipulates, but also the very instructions that tell the computer what to do. Let us explore this architecture in some detail.

5.1.3 Memory

The memory of a von Neumann machine holds two types of information: data items and programming instructions. The two types of information are usually treated differently, and in some computers they are stored in separate memory units. In spite of their different functions though, both types of information are represented as binary numbers that are stored in the same generic random-access structure: a continuous array of cells of some fixed width, also called *words* or *locations*, each having a unique *address*. Hence, an individual word (representing either a data item or an instruction) is specified by supplying its address.

Data Memory High-level programs manipulate abstract artifacts like variables, arrays, and objects. When translated into machine language, these data abstractions become series of binary numbers, stored in the computer's data memory. Once an individual word has been selected from the data memory by specifying its address, it can be either *read* or *written* to. In the former case, we retrieve the word's value. In the latter case, we store a new value into the selected location, erasing the old value.

Instruction Memory When translated into machine language, each high-level command becomes a series of binary words, representing machine language instructions. These instructions are stored in the computer's *instruction memory*. In each step of the computer's operation, the CPU *fetches* (i.e., *reads*) a word from the instruction memory, decodes it, executes the specified instruction, and figures out which instruction to execute next. Thus, changing the contents of the instruction memory has the effect of completely changing the computer's operation.

The instructions that reside in the instruction memory are written in an agreed-upon formalism called *machine language*. In some computers, the specification of each operation and the codes representing its operands are represented in a single-word instruction. Other computers split this specification over several words.

5.1.4 Central Processing Unit

The CPU—the centerpiece of the computer's architecture—is in charge of executing the instructions of the currently loaded program. These instructions tell the CPU to carry out various calculations, to read and write values from and into the memory, and to conditionally jump to execute other instructions in the program. The CPU executes these tasks using three main hardware elements: an *Arithmetic-Logic Unit* (ALU), a set of *registers*, and a *control unit*.

Arithmetic Logic Unit The ALU is built to perform all the low-level arithmetic and logical operations featured by the computer. For instance, a typical ALU can add two numbers, test whether a number is positive, manipulate the bits in a word of data, and so on.

Registers The CPU is designed to carry out simple calculations *quickly*. In order to boost performance, it is desirable to store the results of these calculations locally, rather than ship them in and out of memory. Thus, every CPU is equipped with a small set of high-speed *registers*, each capable of holding a single word.

Control Unit A computer instruction is represented as a binary code, typically 16, 32, or 64 bits wide. Before such an instruction can be executed, it must be decoded, and the information embedded in it must be used to signal various hardware devices (ALU, registers, memory) how to execute the instruction. The instruction decoding is done by some *control unit*, which is also responsible for figuring out which instruction to fetch and execute next.

The CPU operation can now be described as a repeated loop: fetch an instruction (word) from memory; decode it; execute it, fetch the next instruction, and so on. The instruction execution may involve one or more of the following micro tasks: have the ALU compute some value, manipulate internal registers, read a word from the memory, and write a word to the memory. In the process of executing these tasks, the CPU also figures out which instruction to fetch and execute next.

5.1.5 Registers

Memory access is a slow affair. When the CPU is instructed to retrieve the contents of address j of the memory, the following process ensues: (a) j travels from the CPU to the RAM; (b) the RAM's direct-access logic selects the memory register whose address is j; (c) the contents of RAM[j] travel back to the CPU. Registers provide the same service—data retrieval and storage—without the round-trip travel and search expenses. First, the registers reside physically inside the CPU chip, so accessing them is almost instantaneous. Second, there are typically only a handful of registers, compared to millions of memory cells. Therefore, machine language instructions can specify which registers they want to manipulate using just a few bits, resulting in thinner instruction formats.

Different CPUs employ different numbers of registers, of different types, for different purposes. In some computer architectures each register can serve more than one purpose:

Data registers: These registers give the CPU short-term memory services. For example, when calculating the value of $(a - b) \cdot c$, we must first compute and remember the value of $(a - b)$. Although this result can be temporarily stored in some memory location, a better solution is to store it locally inside the CPU—in a *data register*.

Addressing registers: The CPU has to continuously access the memory in order to read data and write data. In every one of these operations, we must specify which individual memory word has to be accessed, namely, supply an address. In some cases this address appears as part of the current instruction, while in others it depends on the execution of a previous instruction. In the latter case, the address should be stored in a register whose contents can be later treated as a memory address—an *addressing register*.

Program counter register: When executing a program, the CPU must always keep track of the address of the next instruction that must be fetched from the instruction memory. This address is kept in a special register called *program counter*, or PC. The contents of the PC are then used as the address for fetching instructions from the instruction memory. Thus, in the process of executing the current instruction, the CPU updates the PC in one of two ways. If the current instruction contains no *goto* directive, the PC is incremented to point to the next instruction in the program. If the current instruction includes a *goto n* directive that should be executed, the CPU loads *n* into the PC.

5.1.6 Input and Output

Computers interact with their external environments using a diverse array of input and output (I/O) devices. These include screens, keyboards, printers, scanners, network interface cards, CD-ROMs, and so forth, not to mention the bewildering array of proprietary components that embedded computers are called to control in automobiles, weapon systems, medical equipment, and so on. There are two reasons why we do not concern ourselves here with the anatomy of these various devices. First, every one of them represents a unique piece of machinery requiring a unique knowledge of engineering. Second, and for this very same reason, computer scientists have devised various schemes to make all these devices look exactly the same to the computer. The simplest trick in this art is called *memory-mapped I/O*.

The basic idea is to create a binary emulation of the I/O device, making it "look" to the CPU like a normal memory segment. In particular, each I/O device is allocated an exclusive area in memory, becoming its "memory map." In the case of an *input* device (keyboard, mouse, etc.), the memory map is made to continuously *reflect* the physical state of the device; in the case of an *output* device (screen, speakers, etc.), the memory map is made to continuously *drive* the physical state of the device. When external events affect some input devices (e.g., pressing a key on the keyboard or moving the mouse), certain values are written in their respective memory maps. Likewise, if we want to manipulate some output devices (e.g., draw something on the screen or play a tune), we write some values in their respective memory maps. From the hardware point of view, this scheme requires each I/O device to provide an interface similar to that of a memory unit. From a software point of view, each I/O device is required to define an interaction contract, so that programs can access it correctly. As a side comment, given the multitude of available computer platforms and I/O devices, one can appreciate the crucial role that *standards* play in designing computer architectures.

The practical implications of a memory-mapped I/O architecture are significant: The design of the CPU and the overall platform can be totally independent of the number, nature, or make of the I/O devices that interact, or *will* interact, with the computer. Whenever we want to connect a new I/O device to the computer, all we have to do is allocate to it a new memory map and "take note" of its base address (these one-time configurations are typically done by the operating system). From this point onward, any program that wants to manipulate this I/O device can do so—all it needs to do is manipulate bits in memory.

5.2 The Hack Hardware Platform Specification

5.2.1 Overview

The Hack platform is a 16-bit von Neumann machine, consisting of a CPU, two separate memory modules serving as instruction memory and data memory, and two memory-mapped I/O devices: a screen and a keyboard. Certain parts of this architecture—especially its machine language—were presented in chapter 4. A summary of this discussion is given here, for ease of reference.

The Hack computer executes programs that reside in its instruction memory. The instruction memory is a read-only device, and thus programs are loaded into it using some exogenous means. For example, the instruction memory can be implemented in a ROM chip that is preburned with the required program. Loading a new program can be done by replacing the entire ROM chip. In order to simulate this operation, hardware simulators of the Hack platform must provide a means for loading the instruction memory from a text file containing a program written in the Hack machine language. (From now on, we will refer to Hack's data memory and instruction memory as RAM and ROM, respectively.)

The Hack CPU consists of the ALU specified in chapter 2 and three registers called *data register* (D), *address register* (A), and *program counter* (PC). D and A are general-purpose 16-bit registers that can be manipulated by arithmetic and logical instructions like `A=D-1`, `D=D|A`, and so on, following the Hack machine language specified in chapter 4. While the D-register is used solely to store data values, the contents of the A-register can be interpreted in three different ways, depending on the instruction's context: as a data value, as a RAM address, or as a ROM address.

The Hack machine language is based on two 16-bit command types. The *address instruction* has the format `0vvvvvvvvvvvvvvv`, each `v` being 0 or 1. This instruction causes the computer to load the 15-bit constant `vvv...v` into the A-register. The *compute instruction* has the format `111accccccdddjjj`. The `a`- and `c`-bits instruct the ALU which function to compute, the `d`-bits instruct where to store the ALU output, and the `j`-bits specify an optional jump condition, all according to the Hack machine language specification.

The computer architecture is wired in such a way that the output of the program counter (PC) chip is connected to the address input of the ROM chip. This way, the ROM chip always emits the word ROM[PC], namely, the contents of the instruction memory location whose address is "pointed at" by the PC. This value is called the *current instruction*. With that in mind, the overall computer operation during each clock cycle is as follows:

Execute: Various bit parts of the current instruction are simultaneously fed to various chips in the computer. If it's an *address instruction* (most significant bit = 0), the A-register is set to the 15-bit constant embedded in the instruction. If it's a *compute instruction* (MSB = 1), its underlying `a`-, `c`-, `d`- and `j`-bits are treated as control bits that cause the ALU and the registers to execute the instruction.

Fetch: Which instruction to fetch next is determined by the jump bits of the current instruction and by the ALU output. Taken together, these values determine whether a jump should materialize. If so, the PC is set to the value of the A-register; otherwise, the PC is incremented by 1. In the next clock cycle, the instruction that the program counter points at emerges from the ROM's output, and the cycle continues.

This particular fetch-execute cycle implies that in the Hack platform, elementary operations involving memory access usually require two instructions: an *address instruction* to set the `A` register to a particular address, and a subsequent *compute instruction* that operates on this address (a read/write operation on the RAM or a jump operation into the ROM).

We now turn to formally specify the Hack hardware platform. Before starting, we wish to point out that this platform can be assembled from previously built components. The CPU is based on the ALU built in chapter 2. The *registers* and the *program counter* are identical copies of the 16-bit register and 16-bit counter, respectively, built in chapter 3. Likewise, the ROM and the RAM chips are versions of the memory units built in chapter 3. Finally, the *screen* and the *keyboard* devices will interface with the hardware platform through memory maps, implemented as built-in chips that have the same interface as RAM chips.

5.2.2 Central Processing Unit (CPU)

The CPU of the Hack platform is designed to execute 16-bit instructions according to the Hack machine language specified in chapter 4. It expects to be connected to two separate memory modules: an instruction memory, from which it fetches instructions for execution, and a data memory, from which it can read, and into which it can write, data values. Figure 5.2 gives the specification details.

5.2.3 Instruction Memory

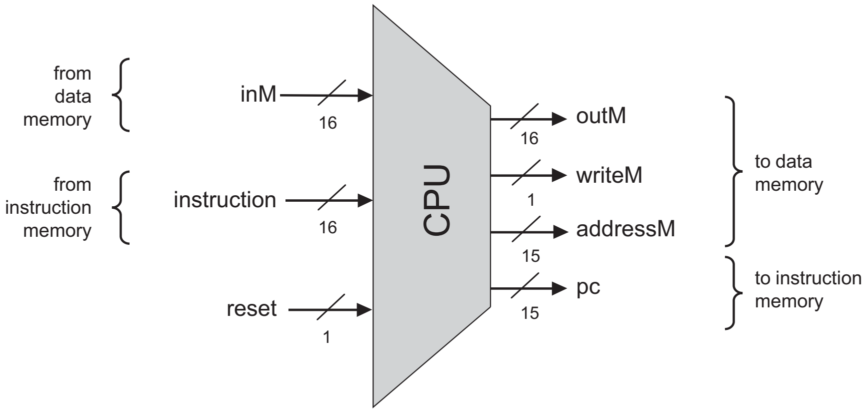
The Hack instruction memory is implemented in a direct-access Read-Only Memory device, also called ROM. The Hack ROM consists of 32K addressable 16-bit registers, as shown in figure 5.3.

5.2.4 Data Memory

Hack's *data memory* chip has the interface of a typical RAM device, like that built in chapter 3 (see, e.g., figure 3.3). To read the contents of register n, we put n in the memory's `address` input and probe the memory's `out` output. This is a combinational operation, independent of the clock. To write a value v into register n, we put v in the `in` input, n in the `address` input, and assert the memory's `load` bit. This is a sequential operation, and so register n will commit to the new value v in the next clock cycle.

In addition to serving as the computer's general-purpose data store, the data memory also interfaces between the CPU and the computer's input/output devices, using *memory maps*.

Memory Maps In order to facilitate interaction with a user, the Hack platform can be connected to two peripheral devices: *screen* and *keyboard*. Both devices interact with the computer platform through *memory-mapped* buffers. Specifically, screen images can be drawn and probed by writing and reading, respectively, words in a designated memory segment called *screen memory map*. Similarly, one can check which key is presently pressed on the keyboard by probing a designated memory word called *keyboard memory map*. The memory maps interact with their respective I/O devices via peripheral logic that resides outside the computer. The contract is as follows: Whenever a bit is changed in the screen's memory map, a respective pixel is drawn on the physical screen. Whenever a key is pressed on the physical keyboard, the respective code of this key appears in the keyboard's memory map.

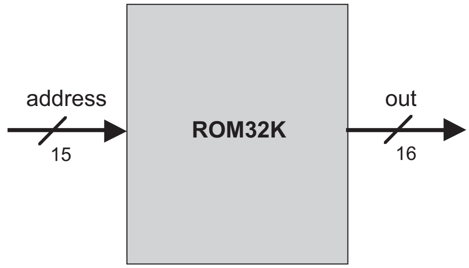



```
Chip Name: CPU                // Central Processing Unit
Inputs:    inM[16],           // M value input  (M = contents of RAM[A])
           instruction[16],   // Instruction for execution
           reset              // Signals whether to restart the current
                              // program (reset=1) or continue executing
                              // the current program (reset=0)
Outputs:   outM[16],          // M value output
           writeM,            // Write to M?
           addressM[15],      // Address of M in data memory
           pc[15]             // Address of next instruction
Function:  Executes the instruction according to the Hack machine language
           specification. The D and A in the language specification refer to
           CPU-resident registers, while M refers to the memory location
           addressed by A (inM holds the value of this location).

           If the instruction needs to write a value to M, the value is
           placed in outM, the address is placed in addressM, and the writeM
           bit is asserted. (When writeM=0, any value may appear in outM.)

           If reset=1, then the CPU jumps to address 0 (i.e., sets pc=0 in
           the next time unit) rather than to the address resulting from
           executing the current instruction.
```

Figure 5.2 The Central Processing Unit. Assembled from the ALU and the registers built in chapters 2 and 3, respectively.




```
Chip Name: ROM32K             // 16-bit read-only 32K memory
Input:     address[15]        // Address in the ROM
Output:    out[16]            // Value of ROM[address]
Function:  out=ROM[address]   // 16-bit assignment
Comment:   The ROM is preloaded with a machine language program.
           Hardware implementations can treat the ROM as a
           built-in chip. Software simulators must supply a
           mechanism for loading a program into the ROM.
```

Figure 5.3 Instruction memory.

We specify first the built-in chips that interface between the hardware interface and the I/O devices, then the complete memory module that embeds these chips.

Screen The Hack computer can interact with a black-and-white screen organized as 256 rows of 512 pixels per row. The computer interfaces with the physical screen via a memory map, implemented by a chip called Screen. This chip behaves like regular memory, meaning that it can be read and written to. In addition, it features the side effect that any bit written to it is reflected as a pixel on the physical screen (1 = black, 0 = white). The exact mapping between the memory map and the physical screen coordinates is given in figure 5.4.

Keyboard The Hack computer can interact with a standard keyboard, like that of a personal computer. The computer interfaces with the physical keyboard via a chip called Keyboard (figure 5.5). Whenever a key is pressed on the physical keyboard, its 16-bit ASCII code appears as the output of the Keyboard chip. When no key is pressed, the chip outputs 0. In addition to the usual ASCII codes, the Keyboard chip recognizes, and responds to, the keys listed in figure 5.6.

```
Chip Name: Screen        // Memory map of the physical screen
Inputs:    in[16],       // What to write
           load,         // Write-enable bit
           address[13]   // Where to write
Output:    out[16]       // Screen value at the given address
Function:  Functions exactly like a 16-bit 8K RAM:
           1. out(t)=Screen[address(t)](t)
           2. If load(t-1) then Screen[address(t-1)](t)=in(t-1)
           (t is the current time unit, or cycle)
Comment:   Has the side effect of continuously refreshing a 256
           by 512 black-and-white screen (simulators must
           simulate this device). Each row in the physical
           screen is represented by 32 consecutive 16-bit words,
           starting at the top left corner of the screen. Thus
           the pixel at row r from the top and column c from the
           left (0<=r<=255, 0<=c<=511) reflects the c%16 bit
           (counting from LSB to MSB) of the word found at
           Screen[r*32+c/16].
```

Figure 5.4 Screen interface.

```
Chip Name: Keyboard     // Memory map of the physical keyboard.
                        // Outputs the code of the currently
                        // pressed key.
Output:    out[16]      // The ASCII code of the pressed key, or
                        // one of the special codes listed in
                        // figure 5.6, or 0 if no key is pressed.
Function:  Outputs the code of the key presently pressed on the
           physical keyboard.
Comment:   This chip is continuously being refreshed from a
           physical keyboard unit (simulators must simulate this
           service).
```

Figure 5.5 Keyboard interface.

Key pressed	**Keyboard output**	**Key pressed**	**Keyboard output**
newline	128	end	135
backspace	129	page up	136
left arrow	130	page down	137
up arrow	131	insert	138
right arrow	132	delete	139
down arrow	133	esc	140
home	134	f1–f12	141–152

Figure 5.6 Special keyboard keys in the Hack platform.

Now that we've described the internal parts of the data memory, we are ready to specify the entire data memory address space.

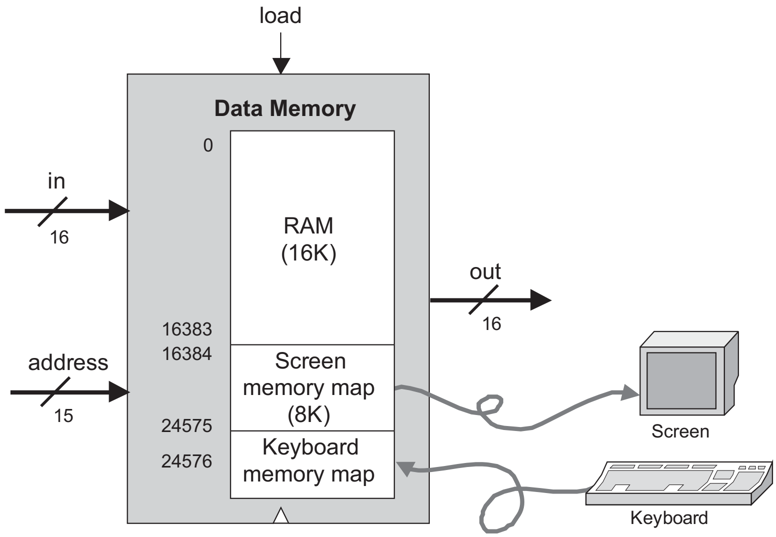
Overall Memory The overall address space of the Hack platform (i.e., its entire data memory) is provided by a chip called Memory. The memory chip includes the RAM (for regular data storage) and the screen and keyboard memory maps. These modules reside in a single address space that is partitioned into four sections, as shown in figure 5.7.

5.2.5 Computer

The topmost chip in the Hack hardware hierarchy is a complete computer system designed to execute programs written in the Hack machine language. This abstraction is described in figure 5.8. The Computer chip contains all the hardware devices necessary to operate the computer including a CPU, a data memory, an instruction memory (ROM), a screen, and a keyboard, all implemented as internal parts. In order to execute a program, the program's code must be preloaded into the ROM. Control of the screen and the keyboard is achieved via their memory maps, as described in the Screen and Keyboard chip specifications.

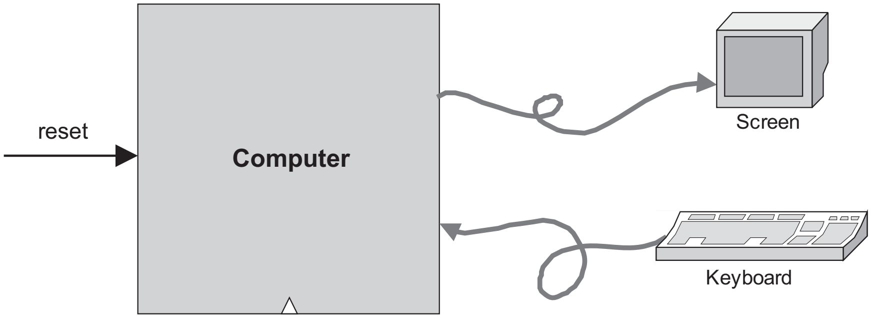
5.3 Implementation

This section gives general guidelines on how the Hack computer platform can be built to deliver the various services described in its specification (section 5.2). As usual, we don't give exact building instructions, expecting readers to come up with




```
Chip Name: Memory        // Complete memory address space
Inputs:    in[16],       // What to write
           load,         // Write-enable bit
           address[15]   // Where to write
Output:    out[16]       // Memory value at the given address
Function:  1. out(t)=Memory[address(t)](t)
           2. If load(t-1) then Memory[address(t-1)](t)=in(t-1)
           (t is the current time unit, or cycle)
Comment:   Access to any address>24576 (0x6000) is invalid.
           Access to any address in the range 16384-24575
           (0x4000-0x5FFF) results in accessing the screen
           memory map. Access to address 24576 (0x6000) results
           in accessing the keyboard memory map. The behavior
           in these addresses is described in the Screen and
           Keyboard chip specifications.
```

Figure 5.7 Data memory.




```
Chip Name: Computer  // Topmost chip in the Hack platform
Input:     reset
Function:  When reset is 0, the program stored in the computer's
           ROM executes. When reset is 1, the execution of the
           program restarts. Thus, to start a program's
           execution, reset must be pushed "up" (1) and then
           "down" (0).

           From this point onward the user is at the mercy of
           the software. In particular, depending on the
           program's code, the screen may show some output and
           the user may be able to interact with the computer
           via the keyboard.
```

Figure 5.8 Computer. Topmost chip of the Hack hardware platform.

their own designs. All the chips can be built in HDL and simulated on a personal computer using the hardware simulator that comes with the book. As usual, technical details are given in the final Project section of this chapter.

Since most of the action in the Hack platform occurs in its Central Processing Unit, the main implementation challenge is building the CPU. The construction of the rest of the computer platform is straightforward.

5.3.1 The Central Processing Unit

The CPU implementation objective is to create a logic gate architecture capable of executing a given Hack instruction and fetching the next instruction to be executed.

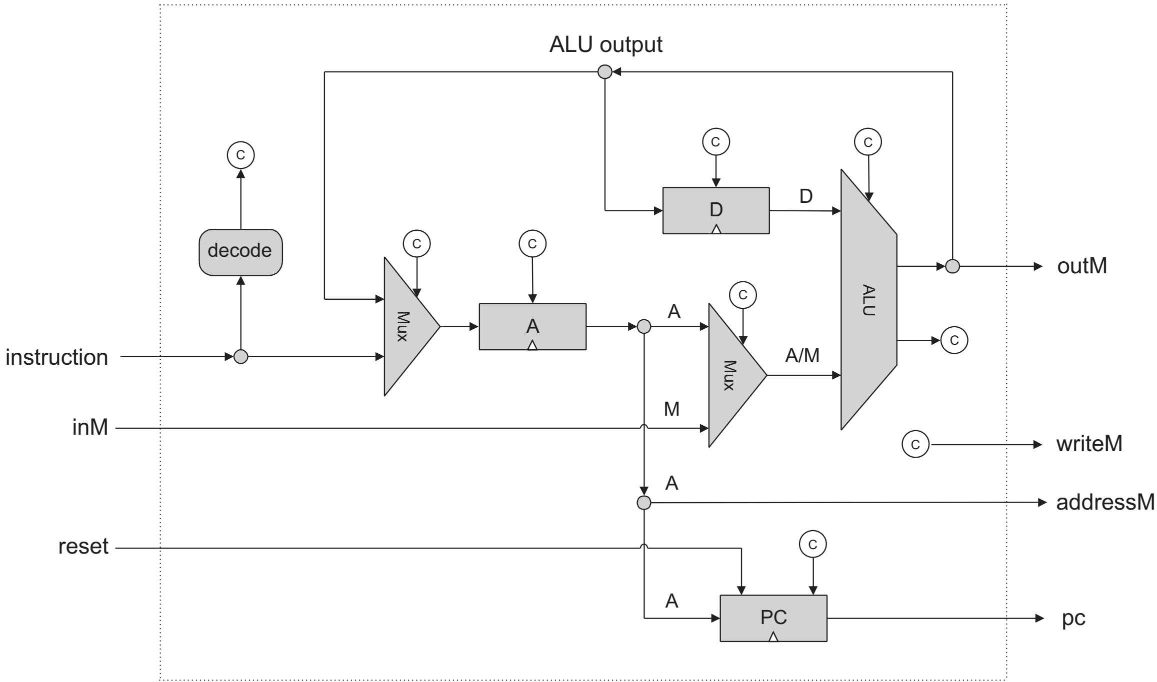


Figure 5.9 Proposed CPU implementation. The diagram shows only *data* and *address paths*, namely, wires that carry data and addresses from one place to another. The diagram does not show the CPU's *control logic*, except for inputs and outputs of control bits, labeled with a circled "c". Thus it should be viewed as an incomplete chip diagram.

Naturally, the CPU will include an ALU capable of executing Hack instructions, a set of registers, and some control logic designed to fetch and decode instructions. Since almost all these hardware elements were already built in previous chapters, the key question here is how to connect them in order to effect the desired CPU operation. One possible solution is illustrated in figure 5.9.

The key element missing in figure 5.9 is the CPU's *control logic*, designed to perform the following tasks:

- *Instruction decoding:* Figure out what the instruction means (a function of the instruction).
- *Instruction execution:* Signal the various parts of the computer what they should do in order to execute the instruction (a function of the instruction).

- *Next instruction fetching:* Figure out which instruction to execute next (a function of the instruction and the ALU output).

(In what follows, the term *proposed CPU implementation* refers to figure 5.9.)

Instruction Decoding The 16-bit word located in the CPU's instruction input can represent either an *A*-instruction or a *C*-instruction. In order to figure out what this 16-bit word means, it can be broken into the fields "`i xx a cccccc ddd jjj`". The `i`-bit codes the instruction type, which is `0` for an *A*-instruction and `1` for a *C*-instruction. In case of a *C*-instruction, the `a`-bit and the `c`-bits code the *comp* part, the `d`-bits code the *dest* part, and the `j`-bits code the *jump* part of the instruction. In case of an *A*-instruction, the 15 bits other than the `i`-bit should be interpreted as a 15-bit constant.

Instruction Execution The various fields of the instruction (`i`-, `a`-, `c`-, `d`-, and `j`-bits) are routed simultaneously to various parts of the architecture, where they cause different chips to do what they are supposed to do in order to execute either the *A*-instruction or the *C*-instruction, as mandated by the machine language specification. In particular, the `a`-bit determines whether the ALU will operate on the A register input or on the Memory input, the `c`-bits determine which function the ALU will compute, and the `d`-bits enable various locations to accept the ALU result.

Next Instruction Fetching As a side effect of executing the current instruction, the CPU also determines the address of the next instruction and emits it via its `pc` output. The "driver" of this task is the *program counter*—an internal part of the CPU whose output is fed directly to the CPU's `pc` output. This is precisely the PC chip built in chapter 3 (see figure 3.5).

Most of the time, the programmer wants the computer to fetch and execute the next instruction in the program. Thus if t is the current time-unit, the default program counter operation should be $\mathrm{PC}(t) = \mathrm{PC}(t-1) + 1$. When we want to effect a *goto* n operation, the machine language specification requires to first set the `A` register to n (via an *A*-instruction) and then issue a jump directive (coded by the `j`-bits of a subsequent *C*-instruction). Hence, our challenge is to come up with a hardware implementation of the following logic:

If $\mathrm{jump}(t)$ then $\mathrm{PC}(t) = \mathrm{A}(t-1)$
else $\mathrm{PC}(t) = \mathrm{PC}(t-1) + 1$

Conveniently, and actually by careful design, this jump control logic can be easily effected by the proposed CPU implementation. Recall that the PC chip interface (figure 3.5) has a `load` control bit that enables it to accept a new input value. Thus, to effect the desired jump control logic, we start by connecting the output of the `A` register to the input of the PC. The only remaining question is when to enable the PC to accept this value (rather than continuing its steadfast counting), namely, when does a jump need to occur. This is a function of two signals: (a) the `j`-bits of the current instruction, specifying on which condition we are supposed to jump, and (b) the ALU output status bits, indicating whether the condition is satisfied. If we have a jump, the PC should be loaded with A's output. Otherwise, the PC should increment by 1.

Additionally, if we want the computer to restart the program's execution, all we have to do is reset the program counter to 0. That's why the proposed CPU implementation feeds the CPU's reset input directly into the reset pin of the PC chip.

5.3.2 Memory

According to its specification, the Memory chip of the Hack platform is essentially a package of three lower-level chips: RAM16K, Screen, and Keyboard. At the same-time, users of the Memory chip must see a single logical address space, spanning from location 0 to 24576 (`0x0000` to `0x6000`—see figure 5.7). The implementation of the Memory chip should create this continuum effect. This can be done by the same technique used to combine small RAM units into larger ones, as we have done in chapter 3 (see figure 3.6 and the discussion of *n-register memory* that accompanies it).

5.3.3 Computer

Once the CPU and the Memory chips have been implemented and tested, the construction of the overall computer is straightforward. Figure 5.10 depicts a possible implementation.

5.4 Perspective

Following the general spirit of the book, the architecture of the Hack computer is rather minimal. Typical computer platforms have more registers, more data types, more powerful ALUs, and richer instruction sets. However, these differences are

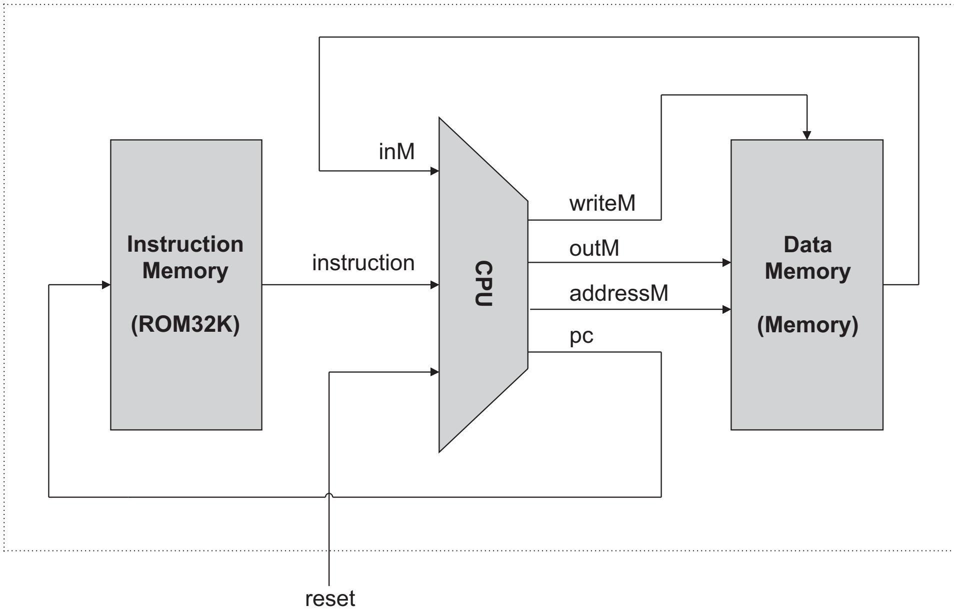


Figure 5.10 Proposed implementation of the topmost Computer chip.

mainly quantitative. From a qualitative standpoint, Hack is quite similar to most digital computers, as they all follow the same conceptual paradigm: the von Neumann architecture.

In terms of function, computer systems can be classified into two categories: *general-purpose computers*, designed to easily switch from executing one program to another, and *dedicated computers*, usually embedded in other systems like cell phones, game consoles, digital cameras, weapon systems, factory equipment, and so on. For any particular application, a single program is burned into the dedicated computer's ROM, and is the only one that can be executed (in game consoles, for example, the game software resides in an external cartridge that is simply a replaceable ROM module encased in some fancy package). Aside from this difference, general-purpose and dedicated computers share the same architectural ideas: stored programs, fetch-decode-execute logic, CPU, registers, program counter, and so on.

Unlike Hack, most general-purpose computers use a single address space for storing both data and instructions. In such architectures, the instruction address as well as the optional data address specified by the instruction must be fed into the same destination: the single address input of the shared address space. Clearly, this cannot be done at the same time. The standard solution is to base the computer implementation on a two-cycle logic. During the *fetch cycle*, the instruction address is fed to the address input of the memory, causing it to immediately emit the current instruction, which is then stored in an *instruction register*. In the subsequent *execute cycle*, the instruction is decoded, and the optional data address inferred from it is fed to the memory's address input, allowing the instruction to manipulate the selected memory location. In contrast, the Hack architecture is unique in that it partitions the address space into two separate parts, allowing a single-cycle fetch-execute logic. The price of this simpler hardware design is that programs cannot be changed dynamically.

In terms of I/O, the Hack keyboard and screen are rather spartan. General-purpose computers are typically connected to multiple I/O devices like printers, disks, network connections, and so on. Also, typical screens are obviously much more powerful than the Hack screen, featuring more pixels, many brightness levels in each pixel, and colors. Still, the basic principle that each pixel is controlled by a memory-resident binary value is maintained: instead of a single bit controlling the pixel's black or white color, several bits are devoted to control the level of brightness of each of the three primary colors that, together, produce the pixel's ultimate color. Likewise, the memory mapping of the Hack screen is simplistic. Instead of mapping pixels directly into bits of memory, most modern computers allow the CPU to send high-level graphic instructions to a *graphics card* that controls the screen. This way, the CPU is relieved from the tedium of drawing figures like circles and polygons directly—the graphics card takes care of this task using its own embedded chip-set.

Finally, it should be stressed that most of the effort and creativity in designing computer hardware is invested in achieving better performance. Thus, hardware architecture courses and textbooks typically evolve around such issues as implementing memory hierarchies (cache), better access to I/O devices, pipelining, parallelism, instruction prefetching, and other optimization techniques that were sidestepped in this chapter.

Historically, attempts to enhance the processor's performance have led to two main schools of hardware design. Advocates of the *Complex Instruction Set Computing* (CISC) approach argue for achieving better performance by providing rich and elaborate instruction sets. Conversely, the *Reduced Instruction Set Computing* (RISC) camp uses simpler instruction sets in order to promote as fast a hardware implementation as possible. The Hack computer does not enter this debate, featuring neither a strong instruction set nor special hardware acceleration techniques.

5.5 Project

Objective Build the Hack computer platform, culminating in the topmost Computer chip.

Resources The only tools that you need for completing this project are the hardware simulator supplied with the book and the test scripts described here. The computer platform should be implemented in the HDL language specified in appendix A.

Contract The computer platform built in this project should be capable of executing programs written in the Hack machine language, specified in chapter 4. Demonstrate this capability by having your Computer chip run the three programs given here.

Component Testing We supply test scripts and compare files for unit-testing the Memory and CPU chips in isolation. It's important to complete the testing of these chips before building and testing the overall Computer chip.

Test Programs A natural way to test the overall Computer chip implementation is to have it execute some sample programs written in the Hack machine language. In order to run such a test, one can write a test script that loads the Computer chip into the hardware simulator, loads a program from an external text file into its ROM chip, and then runs the clock enough cycles to execute the program. We supply all the files necessary to run three such tests, as follows:

1. `Add.hack`: Adds the two constants 2 and 3 and writes the result in RAM[0].
2. `Max.hack`: Computes the maximum of RAM[0] and RAM[1] and writes the result in RAM[2].
3. `Rect.hack`: Draws a rectangle of width 16 pixels and length RAM[0] at the top left of the screen.

Before testing your Computer chip on any one of the above programs, read the test script associated with the program and be sure to understand the instructions given to the simulator. Appendix B may be a useful reference here.

Steps Build the computer in the following order:

- *Memory:* Composed from three chips: RAM16K, Screen, and Keyboard. The Screen and the Keyboard are available as built-in chips and there is no need to build them. Although the RAM16K chip was built in the project in chapter 3, we recommend using its built-in version, as it provides a debugging-friendly GUI.

- *CPU:* Can be composed according to the proposed implementation given in figure 5.9, using the ALU and register chips built in chapters 2 and 3, respectively. We recommend using the built-in versions of these chips, in particular ARegister and DRegister. These chips have exactly the same functionality of the Register chip specified in chapter 3, plus GUI side effects.

In the course of implementing the CPU, it is allowed (but not necessarily recommended) to specify and build some internal chips of your own. This is up to you. If you choose to create new chips not mentioned in the book, be sure to document and test them carefully before you plug them into the architecture.

- *Instruction Memory:* Use the built-in ROM32K chip.
- *Computer:* The topmost Computer chip can be composed from the chips mentioned earlier, using figure 5.10 as a blueprint.

The Hardware Simulator As in the projects in chapters 1–3, all the chips in this project (including the topmost Computer chip) can be implemented and tested using the hardware simulator supplied with the book. Figure 5.11 is a screen shot of testing the `Rect.hack` program on a Computer chip implementation.

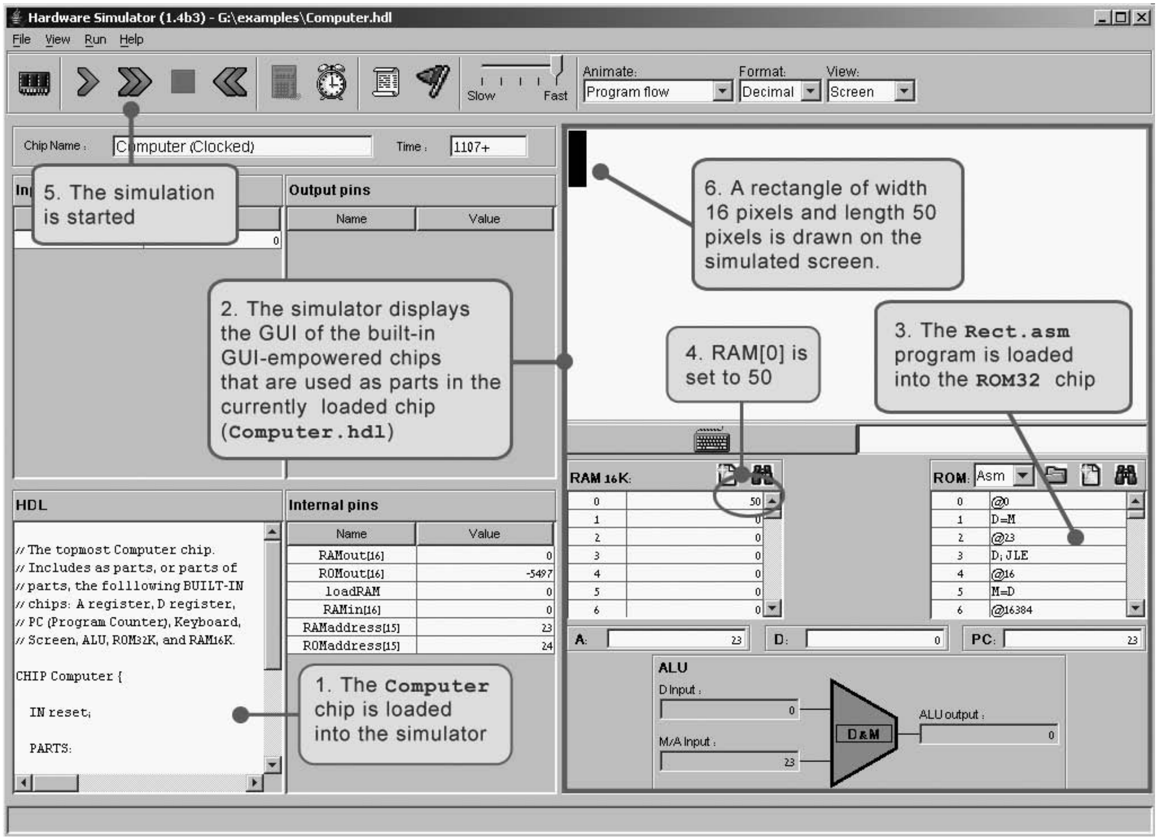


Figure 5.11 Testing the Computer chip on the hardware simulator. The Rect program draws a rectangle of width 16 pixels and length RAM[0] at the top left of the screen. Note that the program is correct. Thus, if it does not work properly, it means that the computer platform on which it runs (Computer.hdl and/or some of its lower-level parts) is buggy.