5. Computer Architecture [1]

"Make everything as simple as possible, but not simpler."

(Albert Einstein, 1879-1955)

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 previous chapters, and integrate them into a general-purpose computer capable of running stored programs written in a machine language. The specific computer that 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 supplied hardware simulator. 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 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 2 gives its exact hardware specification. Section 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.

In the spirit of the opening quote of this chapter, 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 reasonable performance. The comparison of this machine to typical computers is taken up in Section 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, following the technical instructions given in the chapter's last section. The result will be a minimal yet surprisingly powerful computer.

1. Background

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.

[1] From *The Elements of Computing Systems*, Nisan & Schocken, MIT Press, forthcoming in 2003, www.idc.ac.il/csd

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. Importantly, the logic of these programs is not embedded in the hardware platform, 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.

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 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 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. We now turn to describe this architecture in some detail.

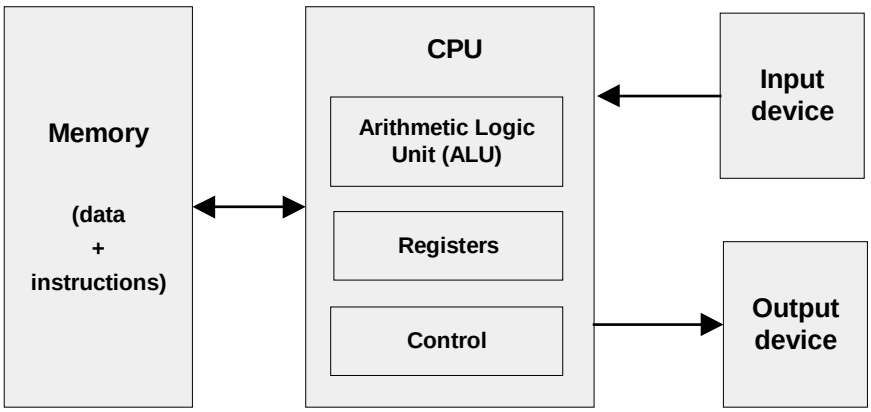


FIGURE 1: The von-Neumann Architecture (conceptual), which, at this level of detail, describes the architecture of almost all digital computers. The program that operates the computer resides in its memory, in accordance with the stored program concept.

Memory

The memory in 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 are stored in separate memory units. In spite of their different functions, both types of information are represented as binary numbers which 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: High level programs use structured commands like `while j<100 {sum=sum+j}`. When translated into machine language, such a command becomes a series of words, each representing a single machine language instruction. 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 underlying 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*. Using these instructions, the programmer can command the CPU to perform arithmetic and logic operations, fetch and store values from and to the memory, move values from one register to another, test Boolean conditions, and so on. In some computers, the specification of each elementary operation (the operation code and the registers and/or memory locations on which it operates) is represented in one word. Computers with a "thin" word-width (e.g. 16-bit) may split this specification over several words.

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. In order to execute these tasks, every CPU employs at least three hardware elements: an *Arithmetic-Logic Unit*, 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, the results of such calculations can often be stored locally, rather than shipped 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- or 32-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 the *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, as we describe below.

Registers

Memory access is a slow process. 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 direct-access logic locates the memory register whose address is *j*; (c) the contents of RAM[*j*] travels 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 shorter 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)*c` where `a`, `b` and `c` are memory locations, 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, i.e. 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 (PC) 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, the CPU loads *n* into the PC.

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, etc., 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 will 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 segment of memory. In particular, each I/O device is allocated an exclusive area in memory, called its "memory map". In the case of an *input* device, the memory map is made to continuously *reflect* the physical state of the device; In the case of an *output* device, 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 some pixels on the screen or move a robotic arm), 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 computer architectures.

We see that in a memory-mapped architecture, 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 configuration tasks 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.

2. The Hack Hardware Platform Specification

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.

The computer can only execute programs that reside in the 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 which is pre-burned 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 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 the data memory and to the 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 specification (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 is 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 sore the ALU output, and the j-bits specify a jump condition, all according to the Hack machine language specification. The computer is built in such a way that the program counter (PC) is connected to the address input of the ROM. This way, the ROM always emits the contents of ROM[PC]. This value is called the *current instruction*. The overall computer operation during each clock cycle is as follows:

Execute: Parts of the current instruction are simultaneously fed to both the A-register and to the ALU. If it's an *address* instruction (most significant bit = 0), the A-register is set to the 15-bit constant embedded in the instruction, and the instruction execution is over. If it's a *compute* instruction (MSB=1), then the a- and c-bits tell the ALU which function to compute. The ALU output is then simultaneously routed to the A and D registers and to the RAM register currently addresses by A. Each one of these registers is equipped with a "load bit" that enables/disables it to incoming inputs. These load bits, in turn, are connected to the three d-bits of the current instruction. For example, "011" causes the machine to disable A, enable D, and enable RAM[A] to load the ALU output.

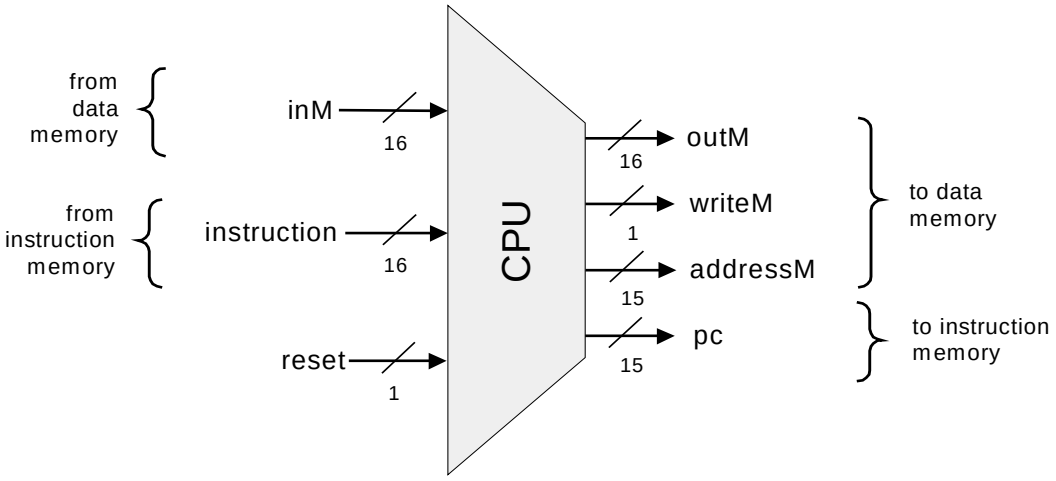
Fetch: Which instruction to fetch next is determined by the three jump bits of the current instruction and by the ALU status output bits. Taken together, these inputs determine if a jump should materialize. If so, the program counter (PC) is set to the value of the A-register; otherwise, the PC is incremented 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. We see that the PC←A setting causes the program flow to branch to the location specified by A, whereas the PC←PC+1 setting causes the program flow to continue with the next instruction in the program.

We now turn to formally specify the Hack hardware platform. Before starting, we wish to point out that most of this platform can be assembled from previously built components. The CPU is based on the *Arithmetic-Logic Unit* 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.

2.2 Central Processing Unit

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. Diagram 2 gives the specification details.

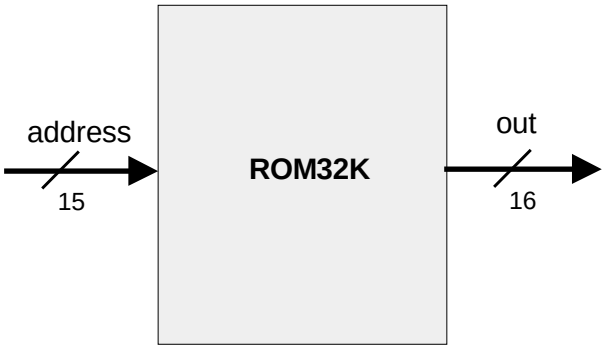



```
Chip Name: CPU                // Central Processing Unit
Inputs:    inM[16],           // input from data memory (M)
           instruction[16],   // instruction from instruction memory
           reset              // signals whether to re-start the current
                              // program (reset=1) or continue executing
                              // the current program (reset=0)
Outputs:   outM[16],          // output to data memory (M)
           writeM,            // write-enable the data memory
           addressM[15],      // address in data memory (of M)
           pc[15]             // address of next instruction
Function:  Executes the inputted 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 external memory location addressed by A, i.e. to
           Memory[A]. The inM input holds the value of this location.

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

           The outM and writeM outputs are combinational: they are
           affected instantaneously by the execution of the current
           instruction.  The addressM and pc outputs are clocked: although
           they are affected by the execution of the current instruction,
           they commit to their new values only in the next time unit.

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

DIAGRAM 2: The Central Processing Unit. This CPU can be built from the ALU and the registers built in Chapters 2 and 3, respectively.

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:



Chip Name:	ROM	// 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 pre-loaded with a machine language program. Simulators must supply a mechanism for loading a program into the ROM.	

DIAGRAM 3: Instruction Memory.

2.4 Data Memory

Hack's *data memory* chip has the interface of a typical RAM device, like those built in Chapter 3 (see for example figure 3-3). To read the contents of register *j*, we put *j* 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 *j*, we put *v* in the `in` input, *j* 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 is stored in the keyboard's memory map.

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

Screen: The Hack computer can be connected to 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 the chip API.

```
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 refreshing a 256 by 512 black-
           and-white screen (simulators must simulate this
           service).  Each row in the physical screen is
           represented by 32 consecutive 16-bit words, starting
           with 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 in Screen[r*32+c/16].
```

DIAGRAM 4: Screen interface

Keyboard: The Hack computer can be connected to a standard keyboard, like that of a personal computer. The computer interfaces with the physical keyboard via a chip called `Keyboard`. 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 chip recognizes, and responds to, the keys listed in table 5.

Key pressed	Keyboard Output	Key pressed	Keyboard Output
new line	128	end	135
backspace	129	page up	136
left arrow	130	page down	137
right arrow	131	insert	138
up Arrow	132	delete	139
down arrow	133	esc	140
home	134	f1-f12	141-152

TABLE 5: Special keyboard keys

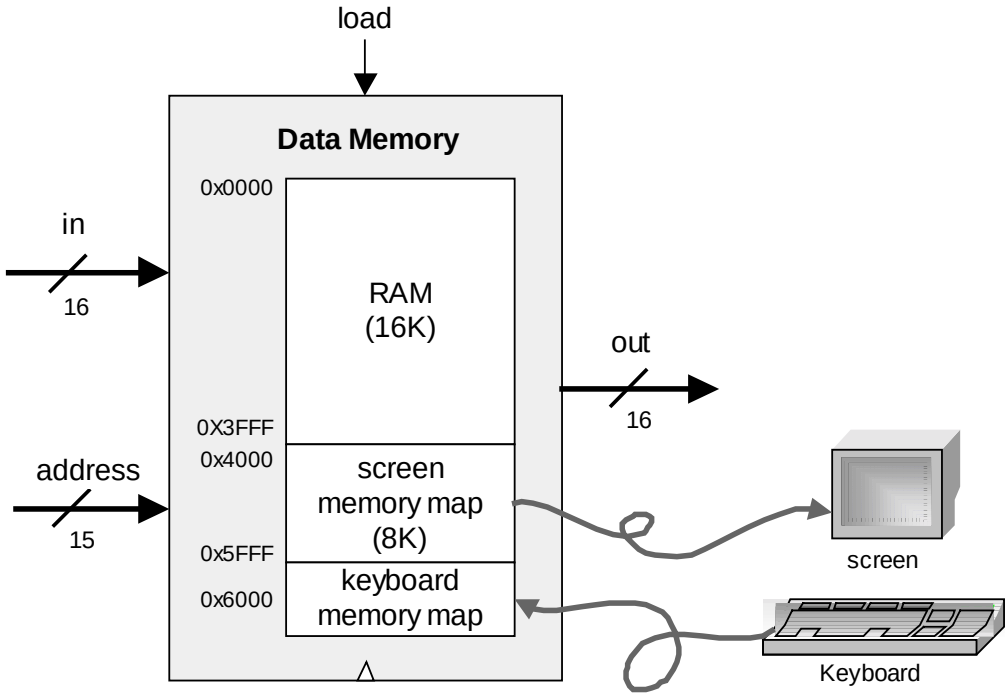
```
Chip Name:  Keyboard     // Memory map of the physical keyboard.
                         // Outputs the code of the currently
                         // pressed key.
Output:     out[16]      // The ASCII code of the pressed key, or
                         // one of the special codes listed in
                         // Table 4-11, 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).
```

DIAGRAM 6: Keyboard interface

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 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:

- Addresses 0-16383 (0x0000-0x3FFF): Regular RAM (16K);
- Addresses 16384-24575 (0x4000-0x5FFF): Screen memory map (8K);
- Address 24576 (0x6000): Keyboard memory map (1 word);
- Addresses 24577-32767 (0x6001-0x7FFF): Unused segment.

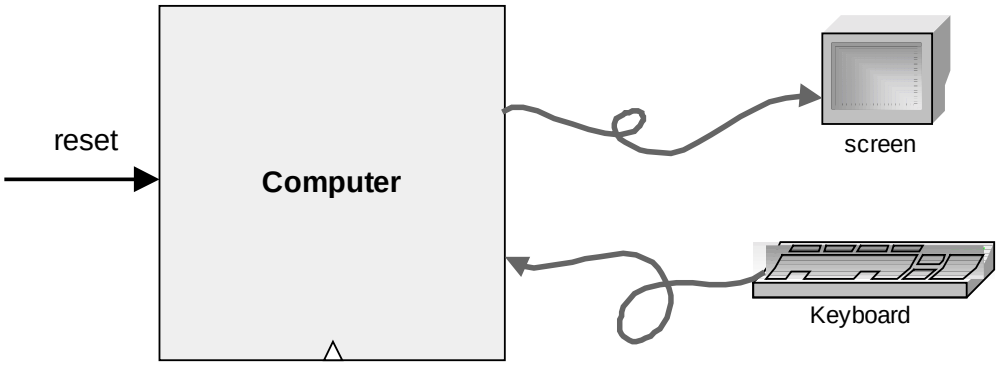



```
Chip Name: Memory       // complete memory address space
Inputs:    in[16],      // what to write
           load,        // write-enable bit
           address[15]  // where to write
Output:    out[16]      // Memory value at the given address
Function:  1. out(t)=Memory[address(t)](t)
           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 address>0x6000 is invalid.  Access to any
           address in the range 0x4000–0x5FFF results in
           accessing the screen memory map.  Access to address
           0x6000 results in accessing the keyboard memory map.
           The behavior in these addresses is described in the
           Screen and Keyboard specifications.
```

DIAGRAM 7: Data Memory.

2.5 Computer

The top-most chip in the Hack hardware hierarchy is a complete computer system designed to execute programs written in the Hack machine language. This `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 pre-loaded into the ROM. Control of the screen and the keyboard is achieved via their memory maps, as described in their specifications.




```
Chip Name: Computer  // top-most 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 "down" (0).

           Depending on the program's code, the screen will
           show some output and the user will be able to
           interact with the computer via the keyboard.

           From this point onward the user is at the mercy
           of the person or company who wrote the software.
```

DIAGRAM 8: Computer. Top-most chip of the Hack hardware platform.

3. Implementation

This section gives general guidelines on how the Hack platform can be built to deliver the various services described in its specification (Section 2). As usual, we don't give exact building instructions, since we expect readers to come up with 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 "Build It" section (section 5).

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 is straightforward.

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. 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 diagram 9.

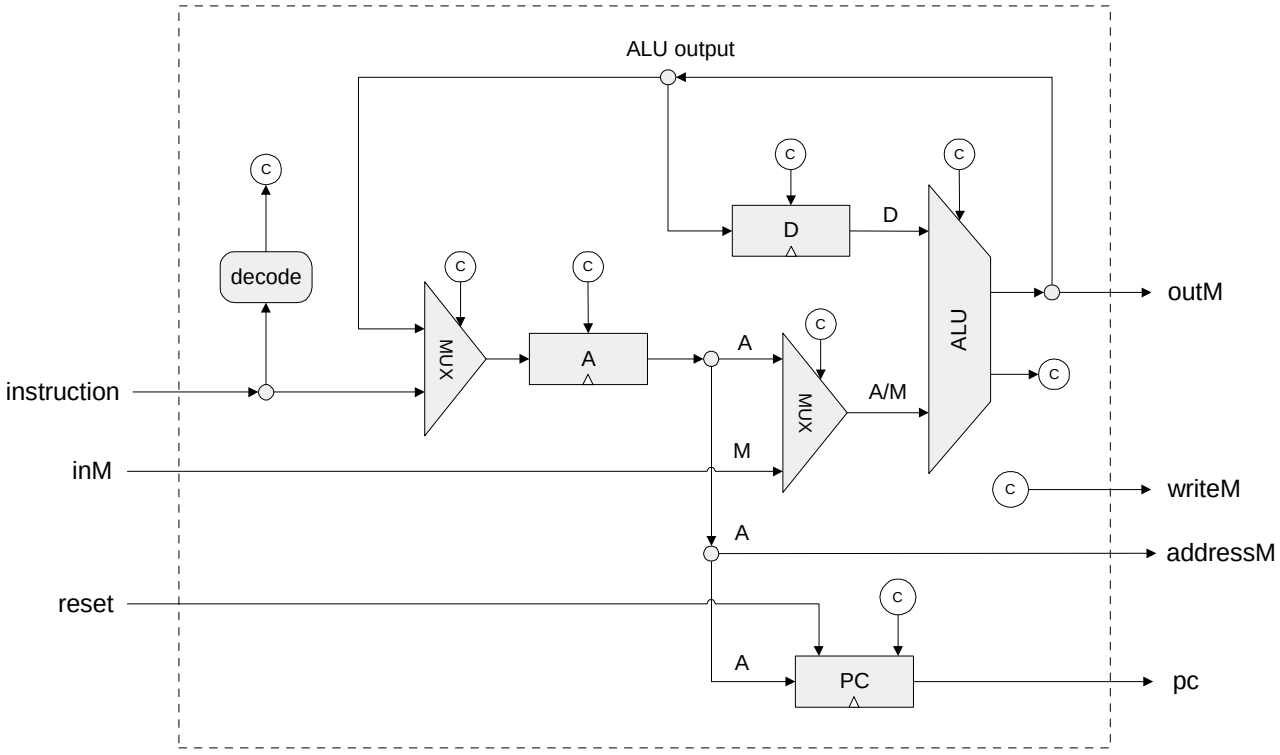


DIAGRAM 9: Proposed CPU Implementation. The diagram shows only *data* and *address paths*, i.e. wires that carry data and addresses from one place to another. The diagram does not show the CPU's *control logic*, except for inputs and outputs of control bits, labeled ©. Thus it should be viewed as an incomplete chip diagram.

The key element missing in diagram 9 is the CPU's *control logic*. The control logic is a rather simple set of gates and wires designed to perform three 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 diagram 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 represent the `comp` part, the `d`-bits represent the `dest` part, and the `j`-bits represent 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 or on the `Memory`, 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 "seat of control" 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 `PC(t)=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 jump(t) then PC(t)=A(t-1)
else PC(t)=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), i.e. 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. Taken together, the `j`-bits and the ALU output status determine whether a jump needs to occur. If we have a jump, the `PC` must be loaded with `A`'s output. If we don't have a jump, the `PC` should increment by 1.

Additionally, if we want the computer to re-start 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 input of the `PC` chip.

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 diagram 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 RAM units, as we have done in Chapter 3 (see figure 3-6 and the discussion of *n-registers memory*).

3.3 Computer

Once the `CPU` and the `Memory` chips have been implemented and tested, the construction of the overall computer is straightforward. Diagram 10 depicts a possible implementation.

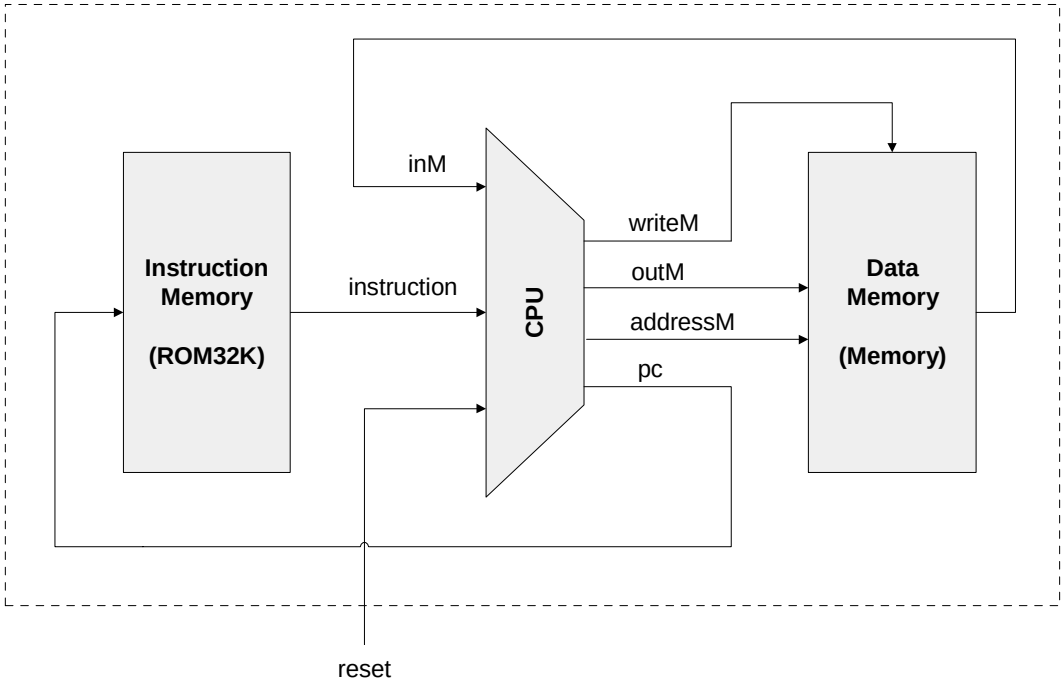


DIAGRAM 10: Proposed implementation of the top-most Computer chip.

4. Perspective

Following the general spirit of the book, the architecture of the Hack computer is rather simple and minimal. Typical computer platforms have more registers, more data types (rather than 16-bit integers only), more powerful ALU's, and more elaborate instruction sets. However, these differences are mainly quantitative. From a qualitative standpoint, Hack is quite similar to most digital computers, as they all follow the same design 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*, which are typically embedded in other systems like cell-phones, game-consoles, appliances, various automobile and airline control systems, factory equipment, and so on. General-purpose computers typically store data and instructions in the RAM, and are able to load programs dynamically into memory. In contrast, dedicated computers typically store software in a ROM unit: for any particular application, a single program is burned into the ROM, and is the only one executed by the embedded computer (for example, in game consoles the game software resides in an external cartridge which is simply a ROM chip encased in some fancy package). In that regard, Hack is similar to a dedicated computer. It should be noted however that general purpose and dedicated computers share the same architectural ideas: stored programs, fetch-decode-execute logic, CPU, registers, program counter, and so on.

In a similar fashion, Hack's I/O devices are as simple as possible (but not simpler). In principle, a computer can be connected to many I/O devices: printers, hard disks, digital cameras, network connections, etc. 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/white color, a number of bits is typically devoted to control the level of brightness of each of the three primary colors that, together, effect the pixel's ultimate color.

Likewise, the memory mapping of the Hack screen is very simplistic. Instead of mapping pixels directly into bits of memory, most modern computers employ more indirect memory maps. In particular, they allow the CPU to send higher-level graphic instructions to a graphics card that controls the screen.

Finally, it should be stressed that most of the effort and creativity in designing computer hardware is aimed at achieving better performance. Thus, hardware architecture courses evolve around such issues as implementing memory hierarchies (cache), better access to I/O devices, pipelines, parallelism, instruction pre-fetching, and other optimization techniques. Historically, the attempts to enhance the processor's performance have led to two main schools of hardware design. Advocates of the CISC (*Complex Instruction Set Computer*) approach argue for achieving better performance by providing as rich and powerful instruction sets as possible. At the same time, the RISC (*Reduced Instruction Set Computer*) 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. Build It

Objective: Build the Hack computer platform, culminating in the top-most `Computer` chip. The only building blocks that you can use are the chips described in this chapter and in previous chapters, and chips that you may build on top of them.

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

Contract: The computer platform that you build 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 below.

Testing: As was just said, a natural way to test your overall `Computer` chip implementation is to have it execute some sample programs written in the Hack language. In order to run such a test, one can write a test script that loads the `Computer` chip into the simulator, loads a program from an external text file into its `ROM32K` 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:

- **`Add.hack`:** this program adds the two constants 2 and 3 and writes the result in `RAM[0]`. Test scripts: `ComputerAdd.tst`, `ComputerAdd.cmp`.
- **`Max.hack`:** this program computes the maximum of `RAM[0]` and `RAM[1]` and writes the result in `RAM[2]`. Test scripts: `ComputerMax.tst`, `ComputerMax.cmp`.
- **`Rect.hack`:** this program draws a rectangle of width 16 pixels and length `RAM[0]` at the top left of the screen. Test scripts: `ComputerRect.tst`, `ComputerRect.cmp`.

Before testing your `Computer` chip on the above programs, read the relevant `.tst` file and be sure that you understand the instructions given to the simulator. Section 8 of Appendix B may be a useful reference here.

Tips: In addition to all the files necessary to test the three programs mentioned above, we supply test scripts and compare files for testing the `Memory` and `CPU` chips. It's important to complete the testing of these chips before you set out to build and test the overall `Computer` chip.

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. The `RAM16K` chip was built 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 diagram 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 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 top-most `Computer` chip can be composed from the chips mentioned above, using diagram 10 as a blueprint.

Steps:

1. Create a directory called `project5` on your computer;
2. Download the `project5.zip` file and extract it to your `project5` directory;
3. Build and test the chips in the order mentioned above.

`