4. 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 one or two days of work, using previously-built chips and the supplied hardware simulator. On the other hand, Hack is sufficiently powerful to illustrate all 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-level hardware and software levels.

Following an introduction of the *stored program* concept, Section 4.1 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 4.2 gives its exact specification, focusing on its architecture, language, and hardware interfaces. Section 4.3 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 4.4, 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 your home computer, following the instructions given in the chapter's last section. The result will be a minimal yet surprisingly powerful computer.

4.1 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 an 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, if not the very foundation of, modern computer science.

[1] From *The Digital Core*, by Nisan & Schocken, 2003, www.idc.ac.il/csd, forthcoming by MIT Press.

`

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 design, 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 *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 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 (Fig. 4-1). 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 at all times. We now turn to describe this architecture in detail.

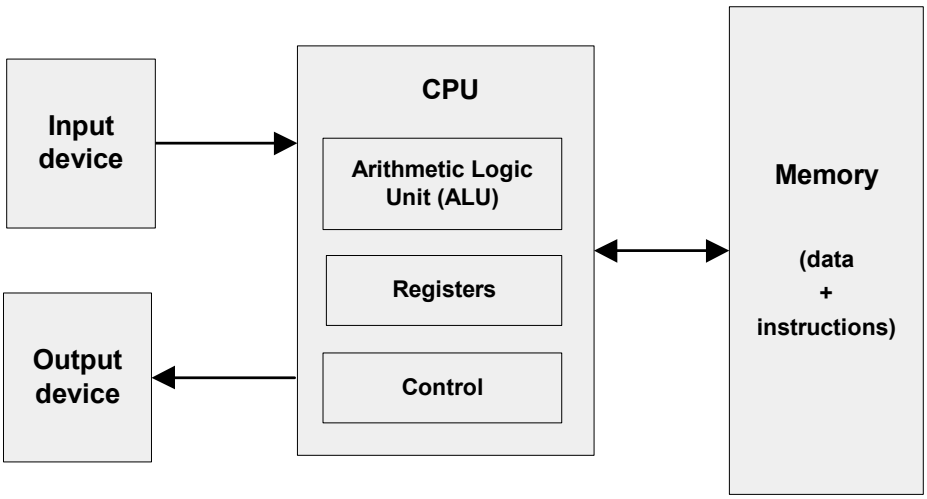


FIGURE 4-1: The von-Neumann Architecture, which, at this level of detail, describes the architecture of almost all digital computers. The program that operates the computer resides in its memory, in accordance with the *stored program* concept.

Memory

In terms of memory organization, the von Neumann architecture comes in two main variants. In some computer platforms, the data and the instructions share the same address space. Other platforms employ separate data and instruction memories, each featuring a different address

space. In spite of their different functions, both memories have 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* according to which it is accessed. Hence, an individual word (representing either a data item or an instruction) is specified by supplying its address. We note in passing that all these memory devices can be constructed from the RAM chips built in Chapter 3.

In some computer platforms (e.g. game machines) the instruction memory is implemented in a read-only direct-access device called ROM (*Read Only memory*). In order to run a new program (play a new game) on such a computer, the entire instruction memory (game cartridge) is replaced, and the computer is reset. Each replaceable ROM device can store a different program, permanently burnt into its memory circuits.

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, the CPU retrieves the word's value. In the latter case, the CPU stores 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.

Computer platforms with separate data and instruction memories have simpler architectures. Therefore, in what follows, we assume a computer with separate memory modules.

Central Processing Unit

The CPU -- the centerpiece of the computer's architecture -- is in charge for 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. The fixed repertoire of the possible ALU operations is called *instruction set*, and is considered a key element of the underlying hardware platform.

Registers: The CPU is designed to carry out *simple* calculations, *quickly*. In order to boost performance, the results of many 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 instructions is represented as a binary code, typically 16- or 32-bits long. This code is divided into several binary fields, each coding a specific operation code (*add*, *sub*, etc.) or arguments of the operation, e.g. register numbers and memory addresses. Before such an instruction can be executed, it must be decoded, and the information embedded in its fields must be used to signal various hardware devices (ALU, registers, memory) how to execute the instruction. The instruction decoding task is done by the CPU's *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 *n* of the memory, the following process ensues: (a) *n* travels from the CPU to the RAM; (b) the RAM locates address *n*; (c) the contents of the selected word travels back to the CPU. Registers provide the same service -- data retrieval and storage -- without the 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), and thus machine language instructions can specify which registers they want to manipulate using just a few bits, yielding faster decoding and processing throughput. Different CPUs employ different numbers of registers, and these registers may be segmented in different classes according to their purpose.

General-purpose 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. Hence, to optimize performance, every CPU uses several general-purpose registers that can serve as immediate outputs and inputs of the ALU.

Addressing registers: The CPU has to continuously access the memory in order to read data, write data, and fetch instructions. In every one of these operations, we must specify *which* individual memory word has to be accessed, i.e. supply an address. This task is also done by the CPU, using built-in addressing hardware. Specifically, the CPU employs an *address register*, whose output feeds the *address input* of the memory chip, via a set of wires called *address bus*. This way, each time the value of the address register changes, the memory responds "automatically" (after a time delay) by selecting the word whose address equals the register's new contents, which is typically the result of some previous ALU calculation. In short, to facilitate direct-access to memory, the CPU needs at least one address register.

Program Counter (PC) register: When executing a program, the CPU must always keep track of the address of the next instruction (word) that must be fetched from the instruction memory. This address is kept in a special address register called *program counter*, or PC, whose output is connected to the address input of the instruction memory. Thanks to this setting, the instruction memory always emits the instruction that the PC points at. In the process of executing the current

instruction, the CPU updates the PC, in one of two different ways. If the current instruction contains no "goto" directive, the PC is incremented to point to the next instruction in the program's code. If the current instruction includes a "*goto n*" directive, the CPU loads *n* into the PC. Since the PC output is hard-wired to the address input of the instruction memory, the next instruction is automatically fetched and served to the CPU for execution.

Machine Language

The computer is controlled by a *program*, which is a series of words (binary numbers). Each word codes an instruction, 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. Thus, as opposed to high level languages, whose basic design goals are generality and power of expression, the goal of machine language's design is direct execution in, and total control of, a given hardware platform. Of course, generality, power and elegance are still desired, but only to the extent that they support, rather than complicate, the basic requirement of direct execution in hardware.

Machine language is the most profound interface in the overall computer enterprise -- the fine line where hardware and software meet. This is the point where the abstract thoughts of the programmer, as manifested in symbolic instructions, are magically turned into physical operations performed in silicon. Thus, machine language is considered both a programming tool and an integral part of the hardware architecture. In fact, just like we say that the machine language is designed to exploit a given hardware architecture, we can say that the hardware architecture is designed to fetch, interpret and execute instructions written in a given machine language. This duality will characterizes much of this chapter.

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*, as we now turn to explain.

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 "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 effect 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. Clearly, this scheme requires that both the computer design and the design of each

I/O device will agree on a pre-defined interaction contract. This contract is simply a *mapping* that specifies how each state of the I/O device is represented in binary codes. As a side comment, given the huge number of available computer platforms and I/O devices, one can appreciate the crucial role that *standards* play in computer architectures.

FIGURE X: A memory-mapped I/O device. Showing how a physical device like a screen is managed using both "resident logic" and "peripheral logic".

We see that memory maps enable the CPU (under the programmer's control) to affect, or be affected by, the physical operation of any I/O device. For example, a memory-mapped screen enables the programmer to use the CPU to probe the screen contents and draw various images on it. Of course, different screens with different resolutions and color pallets will entail different mappings. Using the documented mapping, and knowing the memory base address of the device's map, the programmer can control any connected I/O device by simply reading and writing values from and to the memory.

Memory-mapped I/O provides a good example of how a simple idea can go a long way to achieve hardware simplification and scalability. 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 words in memory.

4.2. The Hack Platform Specification

4.2.1 Basic Constructs

The Hack platform is a special case of the von Newman architecture. It is a 16-bit 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.

Memory Address Spaces: The Hack programmer is aware of two distinct memory address spaces: an *instruction memory* and a *data memory*. Both memories are 16-bit wide and have a 15-bit address space, meaning that the maximum size of each memory is 32K 16-bit words.

The CPU 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 to load the instruction memory from a text file.

Registers: The Hack programmer is aware of three registers called `D`, `A`, and `PC`. `D` and `A` are general-purpose 16-bit registers that can be manipulated explicitly by arithmetic and logical instructions, e.g. `A=D-1` or `D=!A`. (where "`!`" means "`not`"). While `D` is used solely to store data values, `A` doubles as both a data register and an address register. That is to say, depending on the instruction context, the contents of `A` can be interpreted either as a data value, or as an address in the data memory, or as an address in the instruction memory, as explained below.

First, the `A` register can be used to facilitate direct access to the data memory (which, from now on, will be often referred to as "memory"). As the next section will describe, the syntax of the Hack language is such that memory access instructions do not specify an explicit address. Instead, they operate on an implicit memory location labeled "`M`", e.g. `D=M+1`. In order to resolve this address, the contract is such that any Hack instruction involving `M` should effect the memory word whose address equals the current value of `A`, i.e. `Memory[A]`. For example, if we set the `A` register to 516, the subsequent instruction `D=M-1` would imply `D=Memory[516]-1`.

Second, in addition to doubling as a general-purpose register and as an address register for the data memory, the hard working `A` register is also used to facilitate direct access to the instruction memory. As we will see shortly, the syntax of the Hack language is such that "jump" instructions do not specify a particular address. Instead, the contract is such that any "jump" operation effects a jump to the instruction memory word addressed by `A`. For example, if we set `A` to 35, a subsequent "goto" instruction would cause the computer to fetch the instruction located in InstructionMemory[35].

Jump instructions also effect the program counter, or `PC`. As a rule, the next instruction that the computer will fetch and execute is always InstructionMemory[`PC`]. Thus, if the current instruction specifies no jump, the `PC` is incremented by 1. If the current instruction implies a jump, the `PC` is set to the current value of `A`. Hence, the `PC` is also effected by the programmer, but only implicitly.

4.2.2 Machine Language Specification

The *Hack machine language* features two generic instructions represented by 16-bit codes, of which the MSB specifies which instruction it is. A 16-bit code beginning with a "0" represents an *address* instruction, also called *A*-instruction. A 16-bit code beginning with a "1" represents a *compute-store-jump* instruction, also called *C*-instruction. Each instruction has a binary representation, a symbolic representation, and an effect on the computer, as we now turn to specify.

The *A*-Instruction

The *A*-instruction is used to set the `A` register to a 15-bit value:

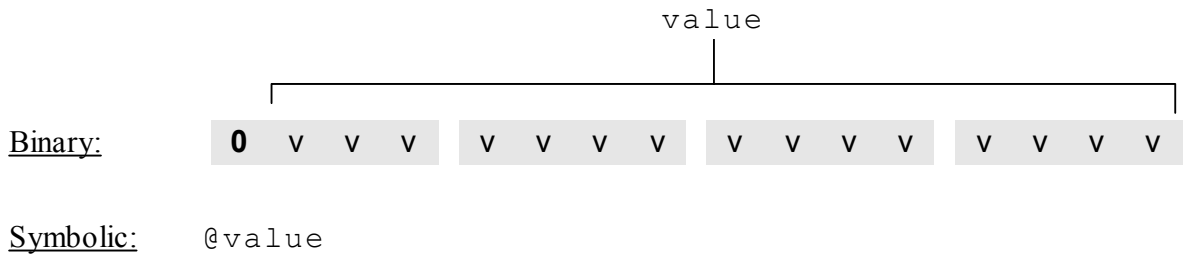


FIGURE 4-2: The *A*-Instruction syntax.

The symbolic syntax of the *A*-instruction is such that `value` is the decimal representation of the 15-bit constant `vvvvvvvvvvvvvvv` (each `v` being 0 or 1). This instruction causes the computer to store the constant in the `A` register. For example, the instruction `@5`, which is equivalent to `0000000000000101`, causes the computer to store the 2's complement binary representation of 5 in the `A` register.

The *A*-instruction is used for three different purposes. First, it provides the only way to enter a constant into the computer under program control. Second, it sets the stage for a *C*-instruction designed to manipulate a certain data memory location, by first setting `A` to the address of that location. Third, it sets the stage for a *C*-instruction that involves a jump, by first loading the address of the jump destination to the `A` register. These different use patterns are illustrated below.

The *C*-Instruction

The *C*-instruction is the programming workhorse of the Hack platform -- the instruction that gets almost everything done. The instruction code is a specification that answers three questions: (a) what to compute? (b) where to store the computed value? and (c) what to do next? Along with the *A*-instruction, these specifications determine all the possible operations of the computer.

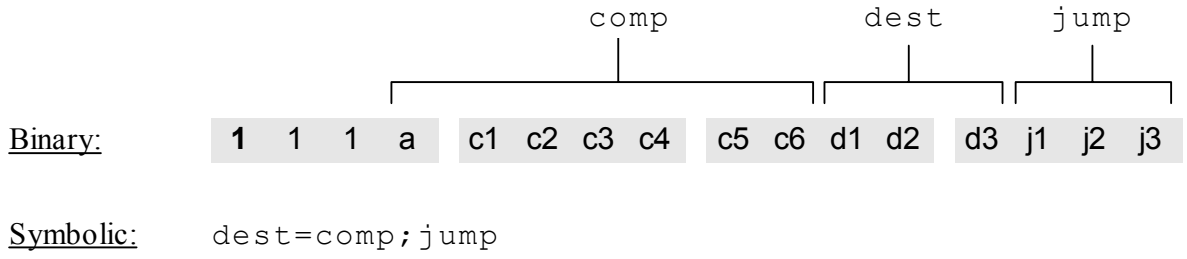


FIGURE 4-3: The *C*-Instruction syntax.

The MSB is the *C*-instruction code, which is 1. The next two bits are not used. The remaining bits form three fields that correspond to the three parts of the instruction's symbolic representation. Taken together, the semantics of the symbolic instruction `dest=comp;jump` is as follow. The `comp` field instructs the CPU what to compute. The `dest` field instructs where to store the computed value. The `jump` field specifies a jump condition.

The computation specification: The ALU is designed to compute a fixed set of functions on the `D`, `A`, and `M` registers (where `M`=Memory[`A`]). The computed function is specified by the `a`-bit and the six `c`-bits comprising the instruction's `comp` field. This 7-bit pattern can potentially code 128 different functions, of which only the 28 listed in Table 4-4 are documented in the computer specification.

a=0							
mnemonic	c1	c2	c3	c4	c5	c6	
0	1	0	1	0	1	0	
1	1	1	1	1	1	1	
-1	1	1	1	0	1	0	
D	0	0	1	1	0	0	
A	1	1	0	0	0	0	M
!D	0	0	1	1	0	1	
!A	1	1	0	0	0	1	!M
-D	0	0	1	1	1	1	
-A	1	1	0	0	1	1	-M
D+1	0	1	1	1	1	1	
A+1	1	1	0	1	1	1	M+1
D-1	0	0	1	1	1	0	
A-1	1	1	0	0	1	0	M-1
D+A	0	0	0	0	1	0	D+M
D-A	0	1	0	0	1	1	D-M
A-D	0	0	0	1	1	1	M-D
D&A	0	0	0	0	0	0	D&M
D|A	0	1	0	1	0	1	D|M
	c1	c2	c3	c4	c5	c6	***mnemonic***
a=1							

TABLE 4-4: The "compute" specification of the *C*-instruction. `D` and `A` are names of registers. `M` refers to the memory location addressed by `A`, i.e. to Memory[`A`]. The symbols "+" and "−" denote 16-bit 2's complement addition and subtraction, respectively, while "`!`", "`|`", and "`&`" denote the 16-bit bit-wise Boolean operators `Not`, `Or`, `And`, respectively. Note the similarity between this instruction set and the ALU specification given in Table 2-7.

Recall that the format of the *C*-instruction is "`111a cccc ccdd djjj`". Suppose we want to compute `D-1`, i.e. "the current value of the `D` register minus 1". According to Table 4-4, this can be done by issuing the instruction "`1110 0011 10xx xxxx`" (we use "`x`" to label bits that are irrelevant to the given example). To compute the value of `D|M`, we issue the instruction "`1111 0101 01xx xxxx`". To compute the constant -1, we issue the instruction "`1110 1110 10xx`

xxxx", and so on. In short, the instruction set supports a selection of 28 different functions of interest involving unary, binary, and null operations on A, D, and M.

The destination specification: The value computed by the comp part of the *C*-instruction can be simultaneously stored in several destinations, as specified by the instruction's dest part. The first and second d-bits code whether to store the computed value in the A register and in the D register, respectively. The third d-bit codes whether to store the computed value in M (i.e. in Memory[A]). One, more than one, or none of these bits may be asserted, as follows:

d1	d2	d3	*mnemonic*	*destination (where to store the computed value)*
0	0	0	null	the value is not stored anywhere
0	0	1	M	Memory[A] (memory register addressed by A)
0	1	0	D	D register
0	1	1	MD	Memory[A] and D register
1	0	0	A	A register
1	0	1	AM	A register and Memory[A]
1	1	0	AD	A register and D register
1	1	1	AMD	A register, Memory[A], and D register

TABLE 4-5: The "destination" specification of the *C*-instruction.

Recall that the format of the *C*-instruction is "111a cccc ccdd djjj". Suppose we want the computer to increment the value of Memory[7] by 1, and also store the result in the D register. According to tables 4-4 and 4-5, this can be accomplished by the instructions:

```
0000 0000 0000 0111    // @7
1111 1101 1101 1xxx    // DM=M+1  (x=irrelevant bits)
```

The first *A*-instruction causes the computer to select the memory register whose address is 7 (the so called "M register"). The subsequent *C*-instruction computes the value of M+1 and stores the result in both D and M. The role of the 3 LSB bits of the second instruction are explained next.

The jump specification: The jump field of the *C*-instruction tells the computer what to do next. There are two possibilities: the next instruction to execute may either be the next instruction in the instruction memory (implying a no-jump: PC=PC+1, where PC is the *program counter*) or it may be the instruction whose address is held in the A register (implying a jump: PC=A). In the latter case, we assume that the A register has been previously set to the address to which we want to jump.

A jump is performed conditionally according to the sign of the value computed by the ALU. The first j-bit specifies whether to jump in case the ALU output is negative, the second j-bit in case the ALU output is zero, and the third j-bit in case it is positive. This gives 8 possible jump conditions, listed in Table 4-6.

j1 ($out \prec 0$)	**j2** ($out = 0$)	**j3** ($out \succ 0$)	**Mnemonic**	**Effect**
0	0	0	`null`	no jump
0	0	1	`JGT`	if $out \succ 0$ jump
0	1	0	`JEQ`	if $out = 0$ jump
0	1	1	`JGE`	if $out \geq 0$ jump
1	0	0	`JLT`	if $out \prec 0$ jump
1	0	1	`JNE`	if $out \neq 0$ jump
1	1	0	`JLE`	if $out \leq 0$ jump
1	1	1	`JMP`	jump

TABLE 4-6: The "jump" specification of the *C*-instruction. In the right column, the "jump" directive implies "fetch the instruction addressed by the `A` register" (i.e. set `PC=A`), while *out* refers to the ALU output value. This value results from the instruction's `comp` part.

For example, suppose we want to effect the following logic: `if Memory[3]=5 goto 100 else goto 200`. According to the language specification, this can be accomplished by the following code:

Desired logic

```
if Memory[j]=5 then
   goto 100
else goto 200
```

Hack implementation

```
@3    // D=Memory[3]
D=M
@5    // D=D-5
D=D-A
@100  // if D=0 goto 100
D;JEQ
@200  // goto 200
0;JMP
```

The last instruction ("`0;JMP`") effects an unconditional jump. Since the *C*-instruction syntax requires that we always effect *some* computation, we instruct to compute 0 (an arbitrary choice).

Restrictions on the *C*-Instruction: As was illustrated in the code above, the programmer can use the `A` register in order to select a *data memory* location for a subsequent *C*-instruction involving `M`, or else `A` can be used to select an *instruction memory* location in order to set the stage for a subsequent *C*-instruction involving a jump. To prevent conflicting use of the `A` register, we require that *a C-instruction that may cause a jump (i.e. with some non-zero j bits) cannot contain a reference to* `M`. Violations of this rule will cause the computer to do something unexpected. Problematic instructions like this one can be intercepted by an *assembler program*, as we will see in the next chapter.

Example

Consider the following C code, designed to sum the numbers between 1 and 100:

```
int i=1;
int sum=0;
while (i<=100){
   sum+=i;
   i++;
}
```

The Hack language implementation of this algorithm is given in Prog. 4-7.

	Symbolic code		addr	Instruction
0	@16	// i=1 (i is at M[16])	0	0000 0000 0001 0000
1	M=1		1	1110 1111 1100 1000
2	@17	// sum=0 (sum is at M[17])	2	0000 0000 0001 0001
3	M=0		3	1110 1010 1000 1000
4	@16	// loop: if i-100>0 goto end	4	0000 0000 0001 0000
5	D=M		5	1111 1100 0001 0000
6	@100		6	0000 0000 0110 0100
7	D=D-A		7	1110 0100 1101 0000
8	@18		8	0000 0000 0001 0010
9	D;JGT		9	1110 0011 0000 0001
10	@16	// sum+=i	10	0000 0000 0001 0000
11	D=M		11	1111 1100 0001 0000
12	@17		12	0000 0000 0001 0001
13	M=D+M		13	1111 0000 1000 1000
14	@16	// i++	14	0000 0000 0001 0000
15	M=M+1		15	1111 1101 1100 1000
16	@4	// goto loop	16	0000 0000 0000 0100
17	0;JMP		17	1110 1010 1000 0111
18	@18	// end: goto end	18	0000 0000 0001 0010
19	0;JMP		19	1110 1010 1000 0111

PROGRAM 4-7: Symbolic and binary representations of the same program. The contract is such that when a program is loaded into the instruction memory, the binary code of the program's *n*-th instruction is stored in address *n*, starting at 0.

Explanation: It's important to remember that in the Hack platform, the *instruction memory* and the *data memory* represent two separate address spaces. Thus address 16 in the former and address 16 in the latter refer to two totally different registers. Prog. 4-7 assumes that variables `i` and `sum` are represented by data memory registers #16 and #17, respectively. This is a reasonable assumption: any other two addresses in the data memory address space will be just as good, as long as *all* references to `i` and `sum` in the program are consistent with this assumption, as indeed is the case. The remaining code is self-explanatory, except perhaps for instructions 18-19. These instructions terminate the program by putting the computer in an infinite loop.

Like all machine languages, Hack's native code is cumbersome and hard to use. Chapter 5 of the book introduces an extension of this language, called *assembly*. The chapter then explains how to

write the *Hack Assembler* – a program that translates Hack assembly programs to binary programs written in the Hack instruction set.

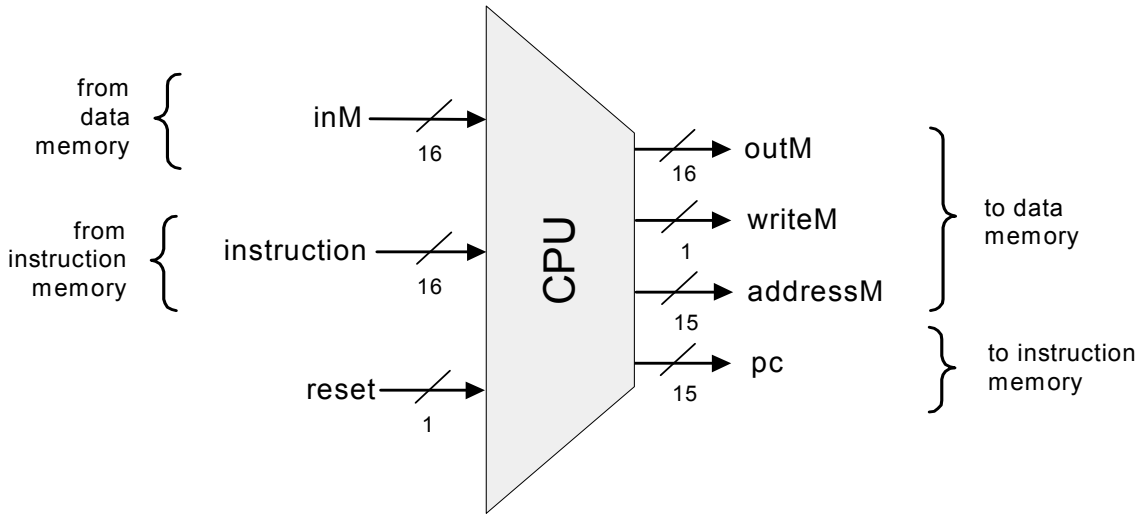
4.2.3 Hardware Specification

Machine languages are *abstractions*. In order to run programs written in a given machine language, we need a machine. This section specifies a hardware platform designed to execute programs written in the Hack machine language. As usual, we describe the machine at the interface level, dealing with implementations issues in a later section.

Before we start specifying the hardware, we wish to point out that most of it can be assembled from previously built components. The CPU is based on the *Arithmetic-Logic Unit* built in Chapter 2. The CPU's *registers* and *program counter* are identical copies of the 16-bit register and 16-bit counter, respectively, built in chapter 3. Likewise, the *instruction memory* and the *data memory* are versions of RAM units, also built in Chapter 3. Finally, the *screen* and the *keyboard* devices will be represented in the architecture through memory maps, and their physical realizations will be implemented as built-in devices, externally available or simulated by the hardware simulator.

Central Processing Unit

The CPU of the Hack platform is designed to execute 16-bit instructions according to the Hack machine language. It expects to be connected to two separate memory modules: an instruction memory, from which it accepts instructions for execution, and a data memory, from which it can read, and into which it can write, data values. Fig. 4-8 gives the specification details.

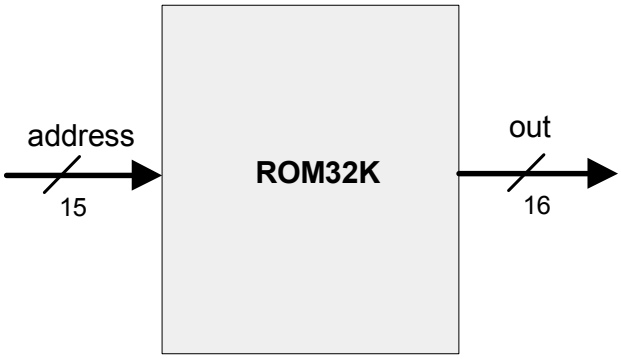



```
Chip Name: CPU                // Central Processing Unit
Inputs:    inM[16],           // input from data memory (M)
           instruction[16],   // instruction from instruction memory
           reset              // signals whether to re-start the current
                              // program (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.
```

INTERFACE 4-8: The Central Processing Unit. This CPU can be built from the ALU and the registers built in Chapters 2 and 3, respectively.

Instruction Memory

The Hack instruction memory is implemented in a read-only direct-access memory device, and thus we will sometimes refer to it as the "ROM". The Hack ROM is a read-only vector 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.
```

INTERFACE 4-9: Read-Only Memory. This ROM chip serves as the instruction memory of the Hack platform.

Data Memory

Hack's *data memory* chip has the interface of a typical RAM device, like those built in Chapter 3 (see for example Fig. 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, as we now turn to explain.

Memory Maps: In order to facilitate interaction with a user, the Hack platform is 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 ascertain 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 specify first the built-in chips that provide the hardware interface to the I/O devices and then specify the complete memory module that embeds them.

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=Screen[address]
           2. If load(t-1) then Screen[address]=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 the screen).
           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].
```

INTERFACE 4-10: Screen chip (memory map)

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 memory map, implemented by a chip called `Keyboard`. Whenever a letter or a digit key is pressed on the physical keyboard, its 16-bit Unicode appears as the output of the `Keyboard` chip. Whenever a control key is pressed, the `Keyboard` chip emits one of the following codes:

Key pressed	Keyboard output
new line	128
backspace	129
left arrow	130
right arrow	131
up Arrow	132
down arrow	133

home	134
end	135
page up	136
page down	137
insert	138
delete	139
esc	140
f1-f12	141-152

TABLE 4-11: Special keyboard keys

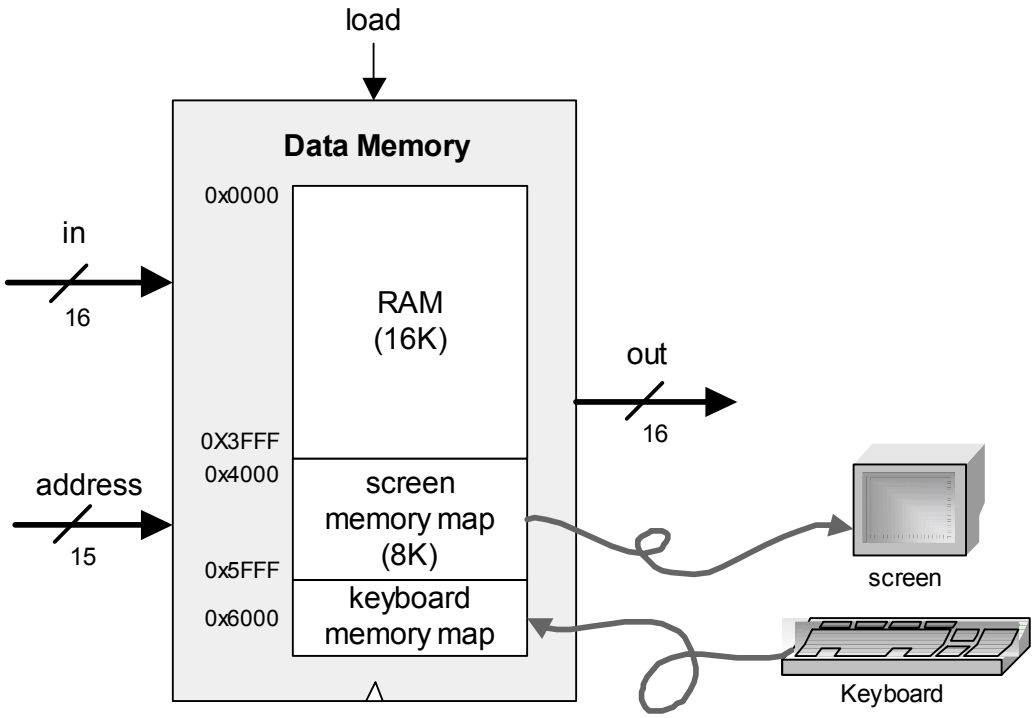
```
Chip Name: Keyboard     // Memory map of the physical keyboard.
                        // Outputs the code of the currently
                        // pressed key.
Output:    out[16]      // If the currently pressed key is a
                        // letter or a digit, the Unicode of that
                        // key.  If it's a control key, one of
                        // the codes listed in table 4-11.
Function:  Outputs the code of the currently pressed key. If no
           key is currently pressed then out=0.
Comment:   Has the side effect of being refreshed from a physical
           keyboard unit (simulators must simulate this service).
```

INTERFACE 4-12: Keyboard chip (memory map)

Now that we've described the internal parts of the data memory, we are in a position 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 interfaces to the RAM (for regular data storage), to the screen, and to the keyboard. These interfaces reside in a single address space that is partitioned into four sections (the last section is unused):

- Memory addresses 0x0000 – 0x3FFF: regular RAM (16K);
- Memory addresses 0x4000 – 0x5FFF: memory map of the screen (8K);
- Memory address 0x6000: memory map of the keyboard (1 word);
- Memory address 0x6001 – 0x7FFF: unused segment.

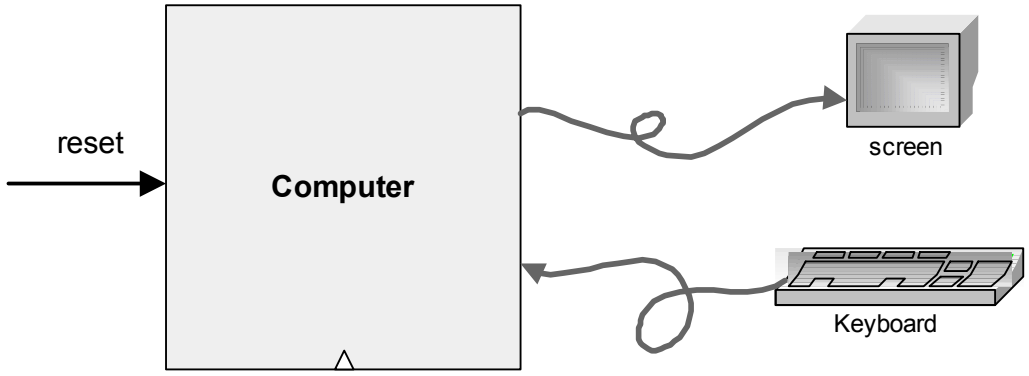



```
Chip Name: Memory        // complete memory address space
Inputs:    in[16],       // what to write
           load,         // write-enable bit
           address[15]   // where to write
Output:    out[16]       // Memory value at the given address
Function:  1. out=Memory[address]
           2. if load(t-1) then Memory[address]=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.
```

INTREFACE 4-13: Data Memory.

Computer

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

INTERFACE 4-14: Computer. Top-most chip of the Hack hardware platform. The first program that the computer runs is typically a ROM-resident bootstrapping code that invokes another program, e.g. the operating system.

4.3 Implementation

This section gives general guidelines on how the Hack platform can be built to deliver the various services described in its specification (Section 4.2). 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 (4.4).

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

The Central Processing Unit

The CPU implementation objective is to create a logic gate architecture that effects the CPU specification, i.e. is 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 the Hack instruction set, 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 inter-connect* them in order to effect the desired CPU operation. One possible solution is illustrated in Fig. 4-15.

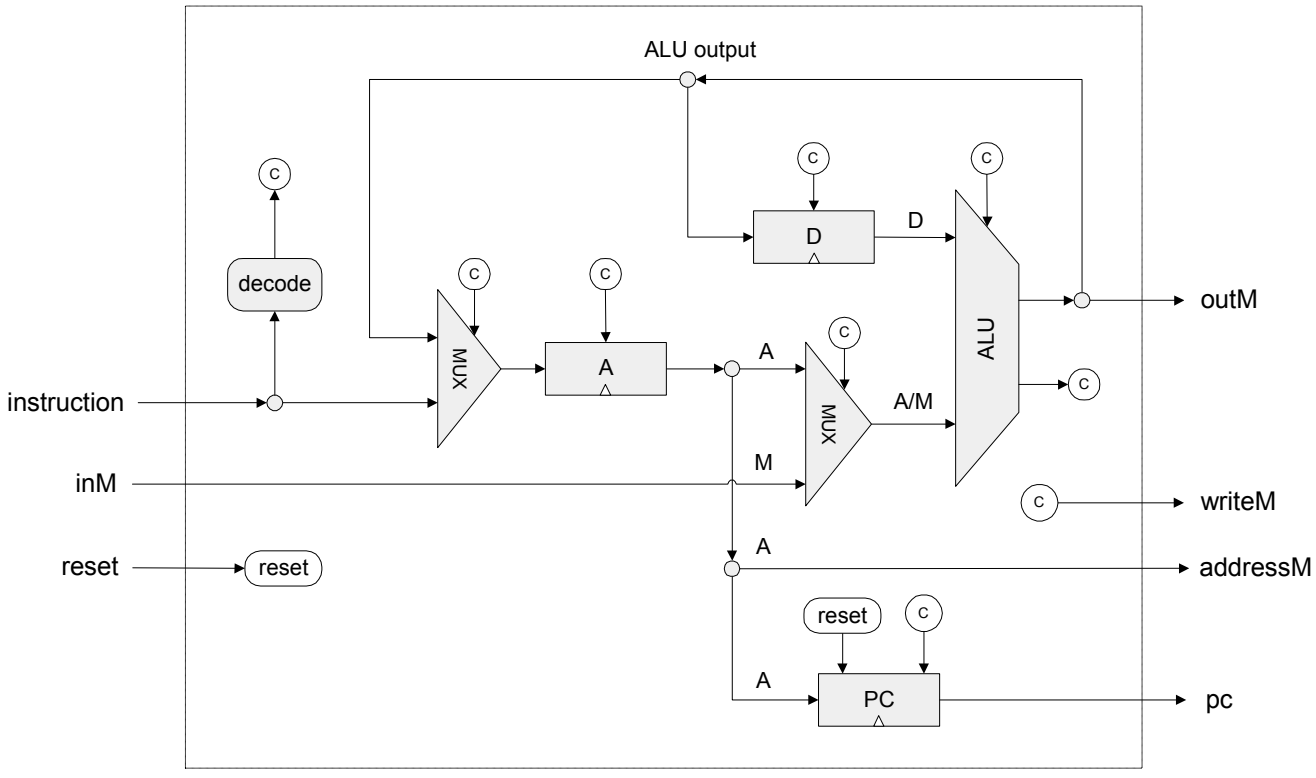


FIGURE 4-15: Proposed CPU Implementation. The diagram shows only *data* and *address paths*, i.e. wires that carry data and addresses from one place to another. The diagram does not show the CPU's *control logic*, except for inputs and outputs of control information. Thus it should be viewed as an incomplete chip diagram.

The key element missing in Fig. 4-15 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 Fig. 4-15).

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 11 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. The (rather trivial) control logic that effects all these decoding tasks is *not* shown in the proposed CPU implementation.

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. The (rather trivial) control logic that routes these control signals to their various destinations is *not* shown in the proposed CPU implementation.

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 Fig. 3-4).

Normally, the `PC` is incremented by 1 each clock cycle. Thus if we reset the `PC` to 0 and let the computer clock run, the `PC` will yield the outputs 0,1,2,3, ... and so on. In order to commence a new counting series, say 12,13,14, ..., we have to set the `PC` input to 12. This simple logic holds the key to the hardware implementation of the software notion of "flow of control", as we now turn to explain.

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* and then issue a jump directive (coded by the

`j`-bits of the *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. First, recall the that ALU chip interface (Fig. 2-6) has two output bits that signal its output status (`ALU` output is $\prec 0, \leq 0, = 0, \succ 0,$ or ≥ 0). Second, recall that the `PC` chip interface (Fig. 3-4) has a "`load`" control bit that enables it to accept a new value via its "`in`" input. 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 the continuing its steadfast counting).

The answer is that the `PC` has to be reset to a new counting base only when a jump needs to occur. And the "jump needs to occur" event 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, informing whether or not 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. The former behavior will cause the computer to fetch the instruction addressed by the `A` register, while the latter behavior will cause the computer to fetch the next instruction in the program's code.

It follows that 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.

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 `0x0000` to `0x6000` (see Fig 4-13). 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 (end of Section 3-3).

Computer

Once the `Memory` and the `CPU` chips have been implemented and tested, the construction of the overall computer is straightforward. Fig. 4-17 depicts a possible implementation.

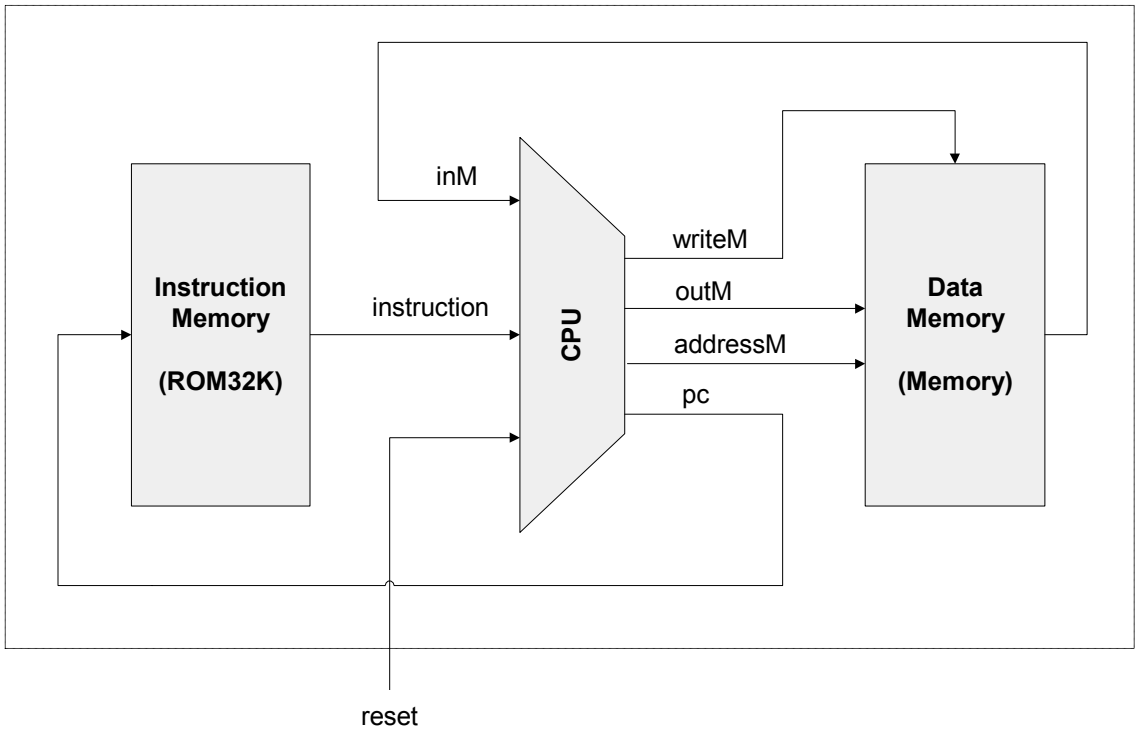


FIGURE 4-17: Proposed implementation of the top-most Computer chip. As documented in interface 4-14, the Computer chip has one input only: reset. Note that the implementation is simply a matter of connecting three chips to each other – a rather simple affair.

4.4 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.

From a functional standpoint, computers can be classified into two categories: *general-purpose* computers and *dedicated* computers, which are typically embedded in other systems like automobiles and airplanes. General-purpose computers store data and instructions in a single address space, and are able to load programs dynamically into memory from an external storage device like a disk. In contrast, embedded computers typically employ separate data and instruction memories, and the latter is typically implemented in ROM. In any particular application, a single program is burned into the embedded computer's ROM, and is the only one

executed by the embedded computer. Incidentally, this is also the operational model of many game computers, where the game software resides in an external cartridge which is simply a ROM chip encased in some fancy package. Clearly, Hack is more similar to embedded computers than to general-purpose computers. At the same time, note that the key characteristics of all these computers are very much alike: stored programs, fetch-decode-execute logic, and so on.

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 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 providing as rich and powerful instruction sets as possible, while the RISC (*Reduced Instruction Set Computer*) camp uses simpler instruction sets in order to promote as fast hardware implementations as possible. The Hack computer does not enter this debate, featuring neither a strong instruction set nor special hardware acceleration techniques.

4.5 Build It

Objective: Build the Hack computer platform, using already-built chips (in this and previous chapters), culminating in the construction of the top-most `Computer` chip. The designed computer platform should be capable of executing programs written in the Hack machine language.

Tips: Build the computer in the following order:

- **Memory**: This chip should be 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, though, as it provides a debugging-friendly GUI.

- **CPU**: This chip can be composed according to the proposed implementation given in Fig. 4-15, using the `ALU` and `Register` chips built in Chapters 2 and 3, respectively. We recommend using the built-in versions of these chips, in order to benefit from their GUI. In particular, the hardware simulator features two built-in chips called `ARegister` and `DRegister`, each having exactly the same functionality of the `Register` chip designed in chapter 3, but providing GUI side-effects.

  In the course of implementing the CPU, it is allowed to specify and build some internal chips of your own, in order to make the design more elegant and manageable. 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 Fig. 4-17 as a blueprint.

Testing: We supply test scripts and compare files for each one of the specified chips. It's important to complete the testing of your `Memory` and `CPU` chips before you set out to build the overall `Computer` chip.

One 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.bin`:** this program adds the two constants 2 and 3 and writes the result in `RAM[0]`. Test scripts: `ComputerAdd.tst`, `ComputerAdd.cmp`.
- **`max.bin`:** 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.bin`:** 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 respective `Xxx.tst` file and make sure that you understand the instructions given to the simulator. It may be helpful to consult section B.8 in Appendix B in order to understand the commands syntax.

The remaining instructions for this project are identical to those of Project 1 (section 1.6), except that every occurrence of the text "`project1`" should be replaced with the text "`project4`".