Example Since the Hack language is self-explanatory, we start with an example. The only non-obvious command in the language is *@value*, where *value* is either a number or a symbol representing a number. This command simply stores the specified value in the A register. For example, if `sum` refers to memory location 17, then both `@17` and `@sum` will have the same effect: `A←17`.

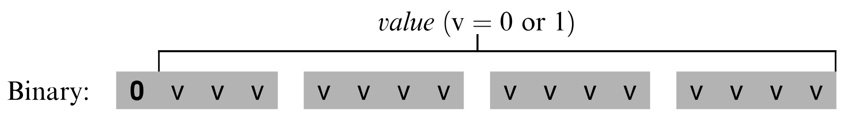
And now to the example: Suppose we want to add the integers 1 to 100, using repetitive addition. Figure 4.2 gives a C language solution and a possible compilation into the Hack language.

Although the Hack syntax is more accessible than that of most machine languages, it may still look obscure to readers who are not familiar with low-level programming. In particular, note that every operation involving a memory location requires two Hack commands: One for selecting the address on which we want to operate, and one for specifying the desired operation. Indeed, the Hack language consists of two generic instructions: an *address instruction*, also called *A*-instruction, and a *compute instruction*, also called *C*-instruction. Each instruction has a binary representation, a symbolic representation, and an effect on the computer, as we now specify.

4.2.2 The *A*-Instruction

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

A-instruction: *@value* // Where *value* is either a non-negative decimal number
// or a symbol referring to such number.



This instruction causes the computer to store the specified value in the A register. For example, the instruction `@5`, which is equivalent to `0000000000000101`, causes the computer to store the 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 subsequent *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 subsequent *C*-instruction that specifies a jump, by first loading the address of the jump destination to the A register. These uses are demonstrated in figure 4.2.

C language

```
// Adds 1+...+100.
  int i = 1;
  int sum = 0;
  While (i <= 100){
     sum += i;
     i++;
  }
```

Hack machine language

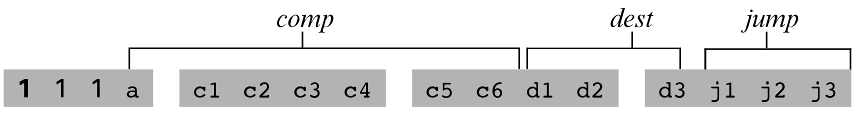
```
// Adds 1+...+100.
       @i     // i refers to some mem. location.
       M=1    // i=1
       @sum   // sum refers to some mem. location.
       M=0    // sum=0
 (LOOP)
       @i
       D=M    // D=i
       @100
       D=D-A  // D=i-100
       @END
       D;JGT  // If (i-100)>0 goto END
       @i
       D=M    // D=i
       @sum
       M=D+M  // sum=sum+i
       @i
       M=M+1  // i=i+1
       @LOOP
       0;JMP  // Goto LOOP
  (END)
       @END
       0;JMP  // Infinite loop
```

Figure 4.2 C and assembly versions of the same program. The infinite loop at the program's end is our standard way to "terminate" the execution of Hack programs.

4.2.3 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.

C-instruction: *dest=comp;jump* // Either the *dest* or *jump* fields may be empty.
// If *dest* is empty, the "=" is omitted;
// If *jump* is empty, the ";" is omitted.

Binary:


The leftmost bit 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. The overall semantics of the symbolic instruction *dest* = *comp;jump* is as follows. The *comp* field instructs the ALU what to compute. The *dest* field instructs where to store the computed value (ALU output). The *jump* field specifies a jump condition, namely, which command to fetch and execute next. We now describe the format and semantics of each of the three fields.

The Computation Specification The Hack ALU is designed to compute a fixed set of functions on the D, A, and M registers (where M stands for 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 figure 4.3 are documented in the language specification.

Recall that the format of the *C*-instruction is `111a cccc ccdd djjj`. Suppose we want to have the ALU compute `D-1`, the current value of the D register minus 1. According to figure 4.3, this can be done by issuing the instruction 111**0 0011 10**00 0000 (the 7-bit operation code is in bold). To compute the value of `D|M`, we issue the instruction 111**1 0101 01**00 0000. To compute the constant −1, we issue the instruction 111**0 1110 10**00 0000, and so on.

The Destination Specification The value computed by the *comp* part of the *C*-instruction can be stored in several destinations, as specified by the instruction's 3-bit

(when a=0) *comp* mnemonic	c1	c2	c3	c4	c5	c6	(when a=1) *comp* mnemonic
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

Figure 4.3 The *compute* field of the *C*-instruction. D and A are names of registers. M refers to the memory location addressed by A, namely, to Memory[A]. The symbols + and − denote 16-bit 2's complement addition and subtraction, while !, |, and & denote the 16-bit bit-wise Boolean operators Not, Or, and And, respectively. Note the similarity between this instruction set and the ALU specification given in figure 2.6.

dest part (see figure 4.4). 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.

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 to also store the result in the D register. According to figures 4.3 and 4.4, this can be accomplished by the following instructions:

```
0000 0000 0000 0111    // @7
1111 1101 1101 1000    // MD=M+1
```

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

Figure 4.4 The *dest* field of the *C*-instruction.

The first instruction causes the computer to select the memory register whose address is 7 (the so-called M register). The second instruction computes the value of $M + 1$ and stores the result in both M and D.

The Jump Specification The *jump* field of the *C*-instruction tells the computer what to do next. There are two possibilities: The computer should either fetch and execute the next instruction in the program, which is the default, or it should fetch and execute an instruction located elsewhere in the program. In the latter case, we assume that the `A` register has been previously set to the address to which we have to jump.

Whether or not a jump should actually materialize depends on the three `j`-bits of the *jump* field and on the ALU output value (computed according to the *comp* field). The first `j`-bit specifies whether to jump in case this value is negative, the second `j`-bit in case the value is zero, and the third `j`-bit in case it is positive. This gives eight possible jump conditions, shown in figure 4.5.

The following example illustrates the jump commands in action:

Logic

```
if Memory[3]=5 then goto 100
else goto 200
```

Implementation

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

j1 ($out < 0$)	j2 ($out = 0$)	j3 ($out > 0$)	**Mnemonic**	**Effect**
0	0	0	null	No jump
0	0	1	JGT	If *out* > 0 jump
0	1	0	JEQ	If *out* $= 0$ jump
0	1	1	JGE	If *out* ≥ 0 jump
1	0	0	JLT	If *out* < 0 jump
1	0	1	JNE	If *out* $\neq 0$ jump
1	1	0	JLE	If *out* ≤ 0 jump
1	1	1	JMP	Jump

Figure 4.5 The *jump* field of the *C*-instruction. *Out* refers to the ALU output (resulting from the instruction's *comp* part), and *jump* implies "continue execution with the instruction addressed by the A register."

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

Conflicting Uses of the A Register As was just illustrated, the programmer can use the A register to select either a *data memory* location for a subsequent *C*-instruction involving M, or an *instruction memory* location for a subsequent *C*-instruction involving a jump. Thus, to prevent conflicting use of the A register, in well-written programs a *C*-instruction that may cause a jump (i.e., with some non-zero `j` bits) should not contain a reference to M, and vice versa.

4.2.4 Symbols

Assembly commands can refer to memory locations (addresses) using either constants or *symbols*. Symbols are introduced into assembly programs in the following three ways:

■ *Predefined symbols:* A special subset of RAM addresses can be referred to by any assembly program using the following predefined symbols:

• *Virtual registers:* To simplify assembly programming, the symbols `R0` to `R15` are predefined to refer to RAM addresses 0 to 15, respectively.

• *Predefined pointers:* The symbols `SP`, `LCL`, `ARG`, `THIS`, and `THAT` are predefined to refer to RAM addresses 0 to 4, respectively. Note that each of these memory