Machine Language Specification

Overview

The computer we are designing is a von Neumann platform. 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 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 addressable 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 programs are loaded into it using some exogenous means. For example, the instruction memory can be implemented in a ROM chip that is pre-burned with the required program. Loading a new program is done by replacing the entire ROM chip, similar to replacing a cartridge in a game console. In order to simulate this operation, hardware simulators of the Hack plat- form must provide a means to load the instruction memory from a text file contain- ing a machine language program.

Registers

The Hack programmer is aware of two 16-bit registers called D and A. These registers can be manipulated explicitly by arithmetic and logical instructions like `A=D-1` or `D=!A` (where "!" means a 16-bit Not operation). 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 we now explain.

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"). Since our instructions are 16-bit wide, and since addresses are specified using 15 bits, it is impossible to pack both an operation code and an address in one instruction. Thus, the syntax of the language mandates that memory access instructions operate on an implicit memory location labeled "`M`", for example, `D=M+1`. In order to resolve this address, the convention is that `M` always refers to the memory word whose address is the current value of the A register. For example, if we want to effect the operation D = Memory[516] - 1, we have to use one instruction to set the A register to 516, and a subsequent instruction to specify `D=M-1`.

In addition, the A register is also used to facilitate direct access to the instruction memory. Similar to the memory access convention, jump instructions do not specify a particular address. Instead, the convention is that any jump operation always effects a jump to the instruction located in the memory word addressed by A. Thus, if we want to effect the

operation *goto* 35, we use one instruction to set A to 35, and a second instruction to code a *goto* command, without specifying an address. This sequence causes the computer to fetch the instruction located in InstructionMemory[35] in the next clock cycle.

Example

Since the 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 1 gives a C language solution and a possible compilation into the machine language.

C language

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

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 1 : C and Assembly versions of the same program. The infinite loop at the program's end is the standard way to terminate the execution of these programs.

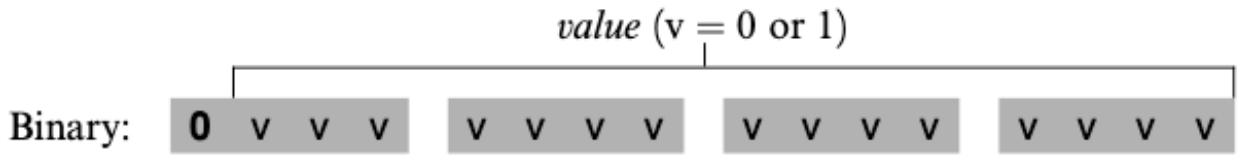
Although the syntax is more accessible than that of most machine languages, it may still look obscure to readers who are not familiar with low-level programming. In particular, note that every operation involving a memory location requires two commands: One for selecting the address on which we want to operate, and one for specifying the desired operation. Indeed, this 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.

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.



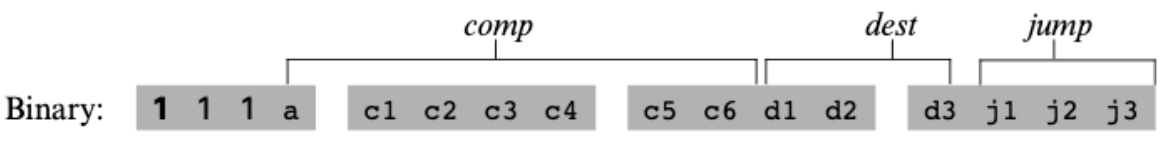
This instruction causes the computer to store the specified value in the A register. For example, the instruction `@5`, which is equivalent to `0000000000000101`, causes the computer to store the binary representation of 5 in the A register.

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 1.

The C-Instruction

The C-instruction is the programming workhorse of our 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.



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 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 below are documented in the language specification.

(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 2 : 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 we designed.

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 2, 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 dest part (see figure 3). 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.

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 3 : The dest field 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 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
```

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.

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 $out \geq 0$ jump
1	0	0	JLT	If $out < 0$ jump
1	0	1	JNE	If $out \neq 0$ jump
1	1	0	JLE	If $out \leq 0$ jump
1	1	1	JMP	Jump

Figure 4 : The jump field of the C-instruction. Out refers to the ALU output and jump implies | continue execution with the instruction addressed by the A register".

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
```

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.

Symbols

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

Syntax:

Assembly Language Files

By convention, assembly language programs are stored in text files with an `.asm` extension, for example, `Prog.asm`. An assembly language file is composed of text lines, each representing either an instruction or a symbol declaration:

- *Instruction*: an A-instruction or a C-instruction.

- *(Symbol)*: This pseudo-command causes the assembler to assign the label Symbol to the memory location in which the next command of the program will be stored. It is called "pseudo-command" since it generates no machine code.

Constants and Symbols

Constants must be non-negative and are always written in decimal notation. A user-defined symbol can be any sequence of letters, digits, un- derscore (_), dot (.), dollar sign ($), and colon (:) that does not begin with a digit.

Comments

Text beginning with two slashes (//) and ending at the end of the line is considered a comment and is ignored.

White Space

Space characters are ignored. Empty lines are ignored.

Case Conventions

All the assembly mnemonics must be written in uppercase. The rest (user-defined labels and variable names) is case sensitive. The convention is to use uppercase for labels and lowercase for variable names.