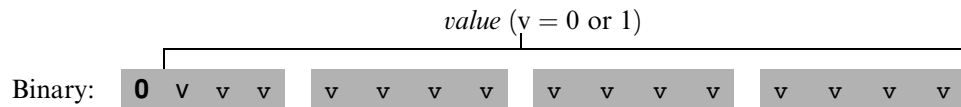


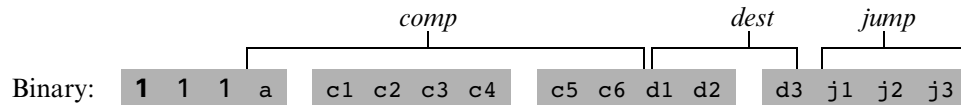
Chip name	Specified in chapter	Has GUI	Comment	
Nand	1		Foundation of all combinational chips	
Not	1			
And	1			
Or	1			
Xor	1			
Mux	1			
DMux	1			
Not16	1			
And16	1			
Or16	1			
Mux16	1			
Or8way	1			
Mux4way16	1			
Mux8way16	1			
DMux4way	1			
DMux8way	1			
HalfAdder	2			
FullAdder	2			
Add16	2			
ALU	2	<input checked="" type="checkbox"/>		
Incl6	2			
DFF	3		Foundation of all sequential chips	
Bit	3			
Register	3			
AResister	3	<input checked="" type="checkbox"/>		
DRegister	3	<input checked="" type="checkbox"/>		
RAM8	3	<input checked="" type="checkbox"/>		
RAM64	3	<input checked="" type="checkbox"/>		
RAM512	3	<input checked="" type="checkbox"/>		
RAM4K	3	<input checked="" type="checkbox"/>		
RAM16K	3	<input checked="" type="checkbox"/>		
PC	3	<input checked="" type="checkbox"/>	Program counter	
ROM32K	5	<input checked="" type="checkbox"/>		GUI allows loading a program from a text file
Screen	5	<input checked="" type="checkbox"/>		
Keyboard	5	<input checked="" type="checkbox"/>		GUI connects to the actual keyboard

Figure A.6 All the built-in chips supplied with the present version of the hardware simulator. A built-in chip has an HDL interface but is implemented as an executable Java class.

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



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 translation of each of the three fields *comp*, *dest*, *jump* to their binary forms is specified in the following three tables.

<i>comp</i> (when a=0)	c1	c2	c3	c4	c5	c6	<i>comp</i> (when a=1)
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

<i>dest</i>	d1	d2	d3	<i>jump</i>	j1	j2	j3
null	0	0	0	null	0	0	0
M	0	0	1	JGT	0	0	1
D	0	1	0	JEQ	0	1	0
MD	0	1	1	JGE	0	1	1
A	1	0	0	JLT	1	0	0
AM	1	0	1	JNE	1	0	1
AD	1	1	0	JLE	1	1	0
AMD	1	1	1	JMP	1	1	1

6.2.3 Symbols

Predefined Symbols Any Hack assembly program is allowed to use the following predefined symbols.

Label *RAM address* (*hexa*)

SP	0	0x0000
LCL	1	0x0001
ARG	2	0x0002
THIS	3	0x0003
THAT	4	0x0004
R0-R15	0-15	0x0000-f
SCREEN	16384	0x4000
KBD	24576	0x6000

Note that each one of the top five RAM locations can be referred to using two predefined symbols. For example, either R2 or ARG can be used to refer to RAM[2].

Command	Return value (after popping the operand/s)	Comment
add	$x + y$	Integer addition (2's complement)
sub	$x - y$	Integer subtraction (2's complement)
neg	$-y$	Arithmetic negation (2's complement)
eq	true if $x = y$, else false	Equality
gt	true if $x > y$, else false	Greater than
lt	true if $x < y$, else false	Less than
and	x And y	Bit-wise
or	x Or y	Bit-wise
not	Not y	Bit-wise

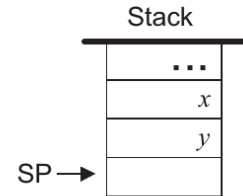


Figure 7.5 Arithmetic and logical stack commands.

Memory Access Commands All the memory segments are accessed by the same two commands:

- `push segment index` Push the value of `segment[index]` onto the stack.
- `pop segment index` Pop the top stack value and store it in `segment[index]`.

Program Flow Commands

```
label symbol           // Label declaration
goto symbol           // Unconditional branching
if-goto symbol       // Conditional branching
```

Function Calling Commands

```
function functionName nLocals
```

```
call functionName nArgs
```

```
return
```

(In this list of commands, *functionName* is a symbol and *nLocals* and *nArgs* are non-negative integers.)

Lexical elements:	The Jack language includes five types of terminal elements (tokens):
keyword:	'class' 'constructor' 'function' 'method' 'field' 'static' 'var' 'int' 'char' 'boolean' 'void' 'true' 'false' 'null' 'this' 'let' 'do' 'if' 'else' 'while' 'return'
symbol:	'{' '}' '(' ')' '[' ']' '.' ' ,' ';' '+' '-' '*' '/' '&' ' ' '<' '>' '=' '~'
integerConstant:	A decimal number in the range 0 .. 32767.
StringConstant	'"' A sequence of Unicode characters not including double quote or newline '"'
identifier:	A sequence of letters, digits, and underscore ('_') not starting with a digit.
Program structure:	A Jack program is a collection of classes, each appearing in a separate file. The compilation unit is a class. A class is a sequence of tokens structured according to the following context free syntax:
class:	'class' className '{' classVarDec* subroutineDec* '}'
classVarDec:	('static' 'field') type varName (',' varName)* ';' ;
type:	'int' 'char' 'boolean' className
subroutineDec:	('constructor' 'function' 'method') ('void' type) subroutineName '(' parameterList ')' ; subroutineBody
parameterList:	((type varName) (',' type varName)*)?
subroutineBody:	'{' varDec* statements '}'
varDec:	'var' type varName (',' varName)* ';' ;
className:	identifier
subroutineName:	identifier
varName:	identifier

Figure 10.5 Complete grammar of the Jack language.

Statements:	
statements:	statement*
statement:	letStatement ifStatement whileStatement doStatement returnStatement
letStatement:	'let' varName ('[' expression ']')? '=' expression ';'
ifStatement:	'if' '(' expression ')' '{' statements '}' ('else' '{' statements '}')?
whileStatement:	'while' '(' expression ')' '{' statements '}'
doStatement:	'do' subroutineCall ';'
ReturnStatement	'return' expression? ';'
Expressions:	
expression:	term (op term)*
term:	integerConstant stringConstant keywordConstant varName varName '[' expression ']' subroutineCall '(' expression ')' unaryOp term
subroutineCall:	subroutineName '(' expressionList ')' (className varName) '.' subroutineName '(' expressionList ')'
expressionList:	(expression (',' expression)*)?
op:	'+' '-' '*' '/' '&' ' ' '<' '>' '='
unaryOp:	'-' '~'
KeywordConstant:	'true' 'false' 'null' 'this'

Figure 10.5 (continued)