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	☑	
`Inc16`	2		
`DFF`	3		Foundation of all sequential chips
`Bit`	3		
`Register`	3		
`ARegister`	3	☑	Identical operation to `Register`, with GUI
`DRegister`	3	☑	Identical operation to `Register`, with GUI
`RAM8`	3	☑	
`RAM64`	3	☑	
`RAM512`	3	☑	
`RAM4K`	3	☑	
`RAM16K`	3	☑	
`PC`	3	☑	Program counter
`ROM32K`	5	☑	GUI allows loading a program from a text file
`Screen`	5	☑	GUI connects to a simulated screen
`Keyboard`	5	☑	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.

The Hack chip-set API

Below is a list of all the chip interfaces in the Hack chip-set, prepared by Warren Toomey. If you need to use a chip-part, you can copy-paste the chip interface and proceed to fill in the missing data. This is a very useful list to have bookmarked or printed.

```
Add16(a= ,b= ,out= );
ALU(x= ,y= ,zx= ,nx= ,zy= ,ny= ,f= ,no= ,out= ,zr= ,ng= );
And16(a= ,b= ,out= );
And(a= ,b= ,out= );
ARegister(in= ,load= ,out= );
Bit(in= ,load= ,out= );
CPU(inM= ,instruction= ,reset= ,outM= ,writeM= ,addressM= ,pc= );
DFF(in= ,out= );
DMux4Way(in= ,sel= ,a= ,b= ,c= ,d= );
DMux8Way(in= ,sel= ,a= ,b= ,c= ,d= ,e= ,f= ,g= ,h= );
DMux(in= ,sel= ,a= ,b= );
DRegister(in= ,load= ,out= );
FullAdder(a= ,b= ,c= ,sum= ,carry= );
HalfAdder(a= ,b= ,sum= , carry= );
Inc16(in= ,out= );
Keyboard(out= );
Memory(in= ,load= ,address= ,out= );
Mux16(a= ,b= ,sel= ,out= );
Mux4Way16(a= ,b= ,c= ,d= ,sel= ,out= );
Mux8Way16(a= ,b= ,c= ,d= ,e= ,f= ,g= ,h= ,sel= ,out= );
Mux8Way(a= ,b= ,c= ,d= ,e= ,f= ,g= ,h= ,sel= ,out= );
Mux(a= ,b= ,sel= ,out= );
Nand(a= ,b= ,out= );
Not16(in= ,out= );
Not(in= ,out= );
Or16(a= ,b= ,out= );
Or8Way(in= ,out= );
Or(a= ,b= ,out= );
PC(in= ,load= ,inc= ,reset= ,out= );
PCLoadLogic(cinstr= ,j1= ,j2= ,j3= ,load= ,inc= );
RAM16K(in= ,load= ,address= ,out= );
RAM4K(in= ,load= ,address= ,out= );
RAM512(in= ,load= ,address= ,out= );
RAM64(in= ,load= ,address= ,out= );
RAM8(in= ,load= ,address= ,out= );
Register(in= ,load= ,out= );
ROM32K(address= ,out= );
Screen(in= ,load= ,address= ,out= );
Xor(a= ,b= ,out= );
```

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

value (v = 0 or 1)

Binary:	0 v v v	v v v v	v v v v	v v v v

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.

comp (a c1 c2 c3 c4 c5 c6) *dest* (d1 d2 d3) *jump* (j1 j2 j3)

Binary:	1 1 1 a	c1 c2 c3 c4	c5 c6 d1 d2	d3 j1 j2 j3

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

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

dest	d1	d2	d3
null	0	0	0
M	0	0	1
D	0	1	0
MD	0	1	1
A	1	0	0
AM	1	0	1
AD	1	1	0
AMD	1	1	1

jump	j1	j2	j3
null	0	0	0
JGT	0	0	1
JEQ	0	1	0
JGE	0	1	1
JLT	1	0	0
JNE	1	0	1
JLE	1	1	0
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

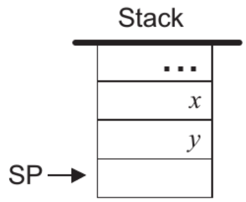

Figure 7.5 Arithmetic and logical stack commands.

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.)

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

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)