## Hack-Maschinensprache und Hack-Assembler

A-Anweisung
(*address instruction*)

**A ← value**

| binary: | 0 | $v_{14}$ | $v_{13}$ | $v_{12}$ | $v_{11}$ | $v_{10}$ | $v_9$ | $v_8$ | $v_7$ | $v_6$ | $v_5$ | $v_4$ | $v_3$ | $v_2$ | $v_1$ | $v_0$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | value | | | | | | | | | | | | | | |

C-Anweisung
(*compute instruction*)

**dest ← comp; jump**

| binary: | 1 | 1 | 1 | m | zx | nx | zy | ny | f | no | $d_2$ | $d_1$ | $d_0$ | $j_2$ | $j_1$ | $j_0$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | comp | | | | | | | dest | | | jump | | |

| m = 0 | zx | nx | zy | ny | f | no | m = 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 |

| $d_2$ | $d_1$ | $d_0$ | Mnemonic | Speicherziel |
|---|---|---|---|---|
| 0 | 0 | 0 | — | Wert wird nicht gespeichert. |
| 0 | 0 | 1 | M | Memory[A] |
| 0 | 1 | 0 | D | D-Register |
| 0 | 1 | 1 | MD | Memory[A] und D-Register |
| 1 | 0 | 0 | A | A-Register |
| 1 | 0 | 1 | AM | Memory[A] und A-Register |
| 1 | 1 | 0 | AD | A- und D-Register |
| 1 | 1 | 1 | AMD | Memory[A] und A- und D-Register |

Memory[A] ist die durch das A-Register adressierte Speicherzelle.

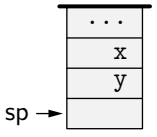
| $j_2$ (out < 0) | $j_1$ (out = 0) | $j_0$ (out > 0) | Mnemonic | Sprung |
|---|---|---|---|---|
| 0 | 0 | 0 | — | niemals |
| 0 | 0 | 1 | JGT | falls out > 0 |
| 0 | 1 | 0 | JEQ | falls out = 0 |
| 0 | 1 | 1 | JGE | falls out ≥ 0 |
| 1 | 0 | 0 | JLT | falls out < 0 |
| 1 | 0 | 1 | JNE | falls out ≠ 0 |
| 1 | 1 | 0 | JLE | falls out ≤ 0 |
| 1 | 1 | 1 | JMP | immer |

Der Status von out wird durch die Ergebnisbits zr und ng der ALU angezeigt.

## Sprache der Virtuellen Maschine

Arithmetisch-logische Operationen

| | | |
|---|---|---|
| add | x + y | Ganzzahladdition (Zweierkomplement) |
| sub | x − y | Ganzzahlsubtraktion (Zweierkomplement) |
| neg | −y | arithmetische Negation (Zweierkomplement) |
| eq | −1 falls x = y, sonst 0 | Test auf Gleichheit |
| gt | −1 falls x > y, sonst 0 | Test auf größer |
| lt | −1 falls x < y, sonst 0 | Test auf kleiner |
| and | x & y | bitweises Und |
| or | x \| y | bitweises Oder |
| not | ~x | bitweise Negation |




Speicherzugriff

| | |
|---|---|
| push *segment index*<br>pop *segment index* | Ablegen des Inhalts von *segment*[*index*] auf dem Stapel.<br>Speichern des obersten Stapelelementes in *segment*[*index*]. |
| Speichersegmente: | constant, static, local, argument, this, that, pointer, temp |

Programmablaufsteuerung

| | |
|---|---|
| label *labelname* | Definiert eine Marke im Programmtext, z.B. als Sprungziel. |
| goto *labelname* | Springt zu einer Marke im Programmtext (unbedingter Sprung). |
| if-goto *labelname* | Springt zu einer Marke im Programmtext, wenn das oberste Stapelelement verschieden von 0 ist. (Dieses Element wird vom Stapel entfernt.) |

Funktionen und Funktionsaufrufe

| | |
|---|---|
| function *fnname k* | Definiert eine Funktion mit dem Namen *fnname*. |
| call *fnname n* | Ruft die Funktion mit dem Namen *fnname* auf. |
| return | Kehrt aus einer Funktion zurück. |

## Programmiersprache Jack

| | |
|---|---|
| **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 on the range 0 . . . 32767. |
| stringConstant: | '"' a sequence of Unicode characters not including double quotes 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 (',' type varName)∗ ';' |
| className: | identifier |
| subroutineName: | identifier |
| varName: | identifier |
| **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' |

| | |
|---|---|
| 'x': | x appears verbatim |
| x: | x is a language construct |
| x?: | x appears 0 or 1 times |
| x*: | x appears 0 or more times |
| x\|y: | either x or y appears |
| (x,y): | x appears, then y |