From NAND to Tetris

`I.BA_NAND2TETRIS.F26` — Spick

Author Lukas
Date 04. Jul. 2026
Pages 15

Inhaltsverzeichnis

1. Elementare Logikgatter

- $x \cdot y \Leftrightarrow xy \Leftrightarrow x$ AND y
- $x + y \Leftrightarrow x$ OR y
- $\overline{x} \Leftrightarrow$ NOT x

Minterm Wenn ein Term für genau eine Belegung den Wert 1 hat

Kanonische Darstellung Ausdrücken einer boolschen Funktion durch einen boolschen Ausdruck

1. Nehme alle Belegungen, für welche in der Wahrheitstabelle das Ergebnis 1 steht
2. Erstelle `AND`-verknüpfter Term für jede Zeile aus Literalen (variable oder negation)
3. Hänge diese Terme mit `OR` zusammen

NAND Jede Operation `AND`, `OR` und `NOT` kann aus `NAND` konsturiert werden

- 1 OR $0 = (1$ NAND $1)$ NAND $(0$ NAND $0) = 0$ NAND $1 = 1$
- 0 OR $0 = (0$ NAND $0)$ NAND $(0$ NAND $0) = 1$ NAND $1 = 0$

!: Jede boolsche Funktion kann mit den drei boolschen operatoren `AND`, `OR` und `NOT` ausgedrückt werden.

Anzahl boolsche Funktionen, welche sich über n binäre variablen definieren lassen ist 2^{2^n}.

1.1. Boolesche Funktionen

> Sie können einfache Boolesche-Funktion mit NOT, AND und OR Bausteinen implementieren (Standardverfahren mit den MIN-Termen) und aufzeichnen.

1. Boolsche Funktion in Wahrheitstabelle überführen
2. Alle Terme mit 1 im output extrahieren und in Min-Terme überführen
3. Terme mit `ODER` / `+` Zusammenhängen
4. Entsprechend vereinfachen

1.2. Testvektoren in Logisim

> Sie verstehen, wie in Logisim Schaltungen mit Testvektoren getestet werden und können solche Tests schreiben.

- TXT-Datei, welche oben die einzelnen In- und Outputs hat, unten dann die Belgungen davon, welche getestet werden

1.3. Logikgatter

> Sie kennen die Funktionalität aller Logikgatter, die vorgestellt wurden.

NOT

Funktion Invertiert den Eingang
NAND $\text{NOT}(A) = A$ NAND A

A	NOT(A)
0	1
1	0

AND

Funktion Gibt TRUE aus, nur wenn beide Eingänge TRUE sind
NAND $\text{AND}(A, B) =$
$\text{NOT}(A$ NAND $B) = (A$ NAND $B)$ NAND $(A$ NAND $B)$

A	B	AND(A, B)
0	0	0
0	1	0
1	0	0
1	1	1

OR

Funktion Gibt TRUE aus, wenn mindestens ein Eingang TRUE ist
NAND $\text{OR}(A, B) =$
$\text{NOT}(A)$ NAND $\text{NOT}(B) = (A$ NAND $A)$ NAND $(B$ NAND $B)$

A	B	OR(A, B)
0	0	0
0	1	1
1	0	1
1	1	1

NAND

Funktion Gibt FALSE aus, nur wenn beide Eingänge TRUE sind
NAND NAND-Gatter ist bereits ein Basis-Gatter und benötigt keine weiteree-mentierung

A	B	NAND(A, B)
0	0	1
0	1	1
1	0	1
1	1	0

NOR

Funktion Gibt TRUE aus, nur wenn beide Eingänge FALSE sind
NAND $\text{NOR}(A, B) =$
$\text{NOT}(A$ OR $B) = (A$ NAND $A)$ NAND $(B$ NAND $B)$

A	B	NOR(A, B)
0	0	1
0	1	0
1	0	0
1	1	0

XOR

Funktion Gibt TRUE aus, wenn ein Eingang TRUE und der andere FALSE ist
NAND $\text{XOR}(A, B) =$
$(A$ NAND $(A$ NAND $B))$ NAND $(B$ NAND $(A$ NAND $B))$

A	B	XOR(A, B)
0	0	0
0	1	1
1	0	1
1	1	0

XNOR

Funktion Gibt TRUE aus, wenn beide Eingänge gleich sind
NAND $\text{XNOR}(A, B) = \text{NOT}(\text{XOR}(A, B))$

A	B	XNOR(A, B)
0	0	1
0	1	0
1	0	0
1	1	1

MUX

Funktion Wählt eines von mehreren Eingangssignalen aus und leitet das ausgewählte Signal an eine einzelne Ausgangsleitung weiter
NAND $\text{MUX}(A, B, S) = (A \text{ NAND } \text{NOT}(S)) \text{ NAND } (B \text{ NAND } S)$
Hinweis MUX gibt A aus, wenn S = 0, und gibt B aus, wenn S = 1

A	B	S	MUX(A, B, S)
0	0	0	0
0	1	0	0
1	0	0	1
1	1	0	1
0	0	1	0
0	1	1	1
1	0	1	0
1	1	1	1

DEMUX

Funktion Nimmt ein einzelnes Eingangssignal und leitet es an eine von mehreren Ausgangsleitungen weiter
NAND $\text{DEMUX}(A, S) = (A \text{ NAND } \text{NOT}(S)), (A \text{ NAND } S)$
Hinweis DEMUX leitet A zur ersten Ausgangsleitung weiter, wenn S = 0, und zur zweiten Ausgangsleitung, wenn S = 1

A	S	DEMUX0(A, S)	DEMUX1(A, S)
0	0	0	0
0	1	0	0
1	0	1	0
1	1	0	1

2. Rechenwerk

2.1. Boolesche Arithmetik

> Sie verstehen die Boolesche Arithmetik, können insbesondere Summen und das Zweierkomplement einer Zahl binär berechnen und verstehen die Codierung für negative Zahlen.

Binäre Zahlen

Im Gegensatz zum Dezimalsystem (Basis 10) basiert das Binärsystem auf der Basis 2

Umrechnung $\left(x_n x_{\{n-1\}} \ldots x_0\right)_B = x_n \cdot B^n + x_{\{n-1\}} \cdot B^{\{n-1\}} + \ldots + x_1 \cdot B + x_0 = \sum_{i=0}^{n} x_i \cdot B^i$
Beispiel $(10011)_2 = 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 19$

Binäre Addition

Funktion Addition wird ziffernweise von rechts nach links durchgeführt
LSB Die rechtesten Ziffern (Least Significant Bits) werden zuerst addiert
Carry Der resultierende Carry-Bit wird zur Summe der nächsten signifikanteren Bitpaare addiert
Beispiel

```
x: 111000000111
y: 000111000111
Carry: 000000000001
x + y: 111111111000
```

Zweierkomplement (2's Complement)

Definition Methode zur Darstellung von vorzeichenbehafteten Zahlen
Formel 2's complement of x = $\begin{cases} 2^n - x \text{ if } x \neq 0 \\ 0 \text{ otherwise} \end{cases}$
Beispiel In einem 5-Bit-System ist das Zweierkomplement von −2: $2^5 - (00010)_b = (11110)_b$
Anwendung Ermöglicht die Darstellung von positiven und negativen Zahlen in gleichen Bitmustern

2.2. Multi-Bit-Addierer

> Sie verstehen wie ein Multi-Bit-Addierer aufgebaut ist (Carry-Ripple-Adder) und können aus einem Addierer einen Inkrementierer bauen.

Half-Adder

Chip Name HalfAdder
Eingänge a, b
Ausgänge sum, carry
Funktion

- sum = LSB von a + b
- carry = MSB von a + b

a	b	sum, carry
0	0	0, 0
0	1	1, 0
1	0	1, 0
1	1	0, 1

Full-Adder

Chip Name FullAdder
Eingänge a, b, c (einschliesslich Carry)
Ausgänge sum, carry
Funktion

- sum = LSB von a + b + c
- carry = MSB von a + b + c

a	b	c	sum, carry
0	0	0	0, 0
0	0	1	1, 0
0	1	0	1, 0
0	1	1	0, 1
1	0	0	1, 0
1	0	1	0, 1
1	1	0	0, 1
1	1	1	1, 1

Multi-Bit-Addierer (Carry-Ripple-Adder)

Chip Name Add16
Eingänge a[16], b[16]
Ausgänge out[16]
Funktion out = a + b
Aufbau Der Carry-Ripple-Adder verbindet mehrere Full-Adder in Serie, wobei der Carry-Ausgang eines Addierers zum Carry-Eingang des nächsten Addierers führt

Inkrementierer

Chip Name Inc16
Eingänge in[16]
Ausgänge out[16]
Funktion out = in + 1
Aufbau Ein Inkrementierer kann aus einem Addierer gebaut werden, indem man eine Konstante von 1 zu der Eingabe addiert

2.3. Hack-ALU

> Sie verstehen den Aufbau der Hack-ALU und können erklären, warum die Operationen gemäss der Spezifikation berechnet werden.

Arithmetisch-Logische Einheit (ALU)

Chip Name ALU
Eingänge x[16], y[16], zx, nx, zy, ny, f, no
Ausgänge out[16], zr, ng
Kontrollbits Werden in genau dieser Reihenfolge abgearbeitet / ausgewertet

1. zx: Wenn 1, dann x = 0
2. nx: Wenn 1, dann x = ¬x
3. zy: Wenn 1, dann y = 0

4. ny: Wenn 1, dann y = ¬y
5. f: Wenn 1, dann out = x + y; sonst out = x & y
6. no: Wenn 1, dann out = ¬out

Ausgabebits

- zr: Wenn out = 0, dann zr = 1; sonst zr = 0
- ng: Wenn out < 0, dann ng = 1; sonst ng = 0

ALU-Operationen nach Kontrollbits

zx	nx	zy	ny	f	no	Funktion
1	0	1	0	1	0	0
1	1	1	1	1	1	1
1	1	1	0	1	0	−1
0	0	1	1	0	0	x
1	1	0	0	0	0	y
0	0	1	1	0	1	!x
1	1	0	0	0	1	!y
0	0	1	1	1	1	-x
1	1	0	0	1	1	-y
0	1	1	1	1	1	x + 1
1	1	0	1	1	1	y + 1
0	0	1	1	1	0	x - 1
1	1	0	0	1	0	y - 1
0	0	0	0	1	0	x + y
0	1	0	0	1	1	x - y
0	0	0	1	1	1	y - x
0	0	0	0	0	0	x & y
0	1	0	1	0	1	x | y

ALU-Beispiele

Null-Ausgabe

Kontrollbits zx = 1, nx = 0, zy = 1, ny = 0, f = 1, no = 0
Ergebnis ALU gibt 0 aus
Erklärung zx und zy setzen beide Eingänge auf 0, dann wird 0 + 0 = 0 berechnet

Y inkrementieren

Kontrollbits zx = 0, nx = 0, zy = 0, ny = 0, f = 1, no = 1
Ergebnis ALU gibt y + 1 aus
Erklärung x und y bleiben unverändert, f = 1 berechnet x + y, dann wird das Ergebnis mit no = 1 invertiert, was y + 1 ergibt

X dekrementieren

Kontrollbits zx = 0, nx = 0, zy = 1, ny = 1, f = 1, no = 0
Ergebnis ALU gibt x - 1 aus
Erklärung zy = 1 setzt y auf 0, ny = 1 invertiert y (¬0 = 1), dann wird x + 1 berechnet und mit no = 0 nicht invertiert, was x - 1 ergibt

X plus Y

Kontrollbits zx = 0, nx = 0, zy = 0, ny = 0, f = 1, no = 0
Ergebnis ALU gibt x + y aus
Erklärung Beide Eingänge bleiben unverändert, f = 1 berechnet die Addition, no = 0 invertiert nicht

3. Speicherelemente

3.1. Kombinatorische und sequentielle Chips

Sie kennen den Unterschied zwischen kombinatorischen und sequentiellen Chips.

Kombinatorische Chips

Definition Digitale Schaltungen, bei denen der Ausgang ausschliesslich von den aktuellen Eingängen abhängt
Eigenschaften

- Zustandslos (stateless)
- Keine Rückkopplung
- Keine Speicherelemente
- Ausgänge werden durch die Kombination der aktuellen Eingabewerte bestimmt

Beispiele AND, OR, NOT, Addierer, Multiplexer

Sequentielle Chips

Definition Digitale Schaltungen, die Speicherelemente enthalten und Zustandsinformationen speichern können
Eigenschaften

- Zustandsbehaftet (stateful)
- Rückkopplung
- Speicherelemente (DFF)
- Ausgänge hängen von aktuellen Eingängen und der Historie ab

Synchronisationsmechanismen

- Taktsignal (Clock Signal)
- Rückkopplung (Feedback)
- Synchronisation durch Takt

Beispiele Register, RAM, Counter, DFF

Taktsignal (Clock Signal)

Definition Ein Taktsignal oszilliert zwischen hohem und niedrigem Zustand
Funktion Synchronisiert alle sequentiellen Elemente in einem System
Taktzyklen Ein vollständiger Zyklus von niedrig zu hoch und zurück definiert eine diskrete Zeiteinheit

3.2. Speichergatter

Sie kennen die Funktionalität aller Speichergatter, die vorgestellt wurden und können erklären, wie Register und RAM-Speicher aufgebaut sind.

Data Flip-Flop (DFF)

Chip Name DFF
Eingänge in
Ausgänge out
Funktion $\text{out}(t) = \text{in}(t-1)$
Beschreibung Das DFF ist das fundamentale Speicherelement der sequentiellen Logik. Es hat einen Eingang und einen Ausgang mit einer Verzögerung von einem Taktzyklus
Anwendung Grundbaustein für alle Speicherelemente in einem Computer

Register

Chip Name Register
Eingänge in[16], load
Ausgänge out[16]
Funktion

- if load(t − 1) = 1 then out(t) = in(t − 1)
- else out(t) = out(t − 1)

Beschreibung Ein Register ist ein Multi-Bit-Speicherelement, das einen Binärwert über die Zeit speichern kann. Es besteht aus einem Array von DFFs und enthält ein Load-Signal
Load-Signal Bestimmt, ob das Register seinen Wert aktualisieren oder den aktuellen Wert beibehalten soll
Aufbau Mehrere Single-Bit-Register können kombiniert werden, um Multi-Bit-Werte zu speichern

Random Access Memory (RAM)

Chip Name RAMn
Eingänge in[16], address[k], load
Ausgänge out[16]
Funktion

- if load(t − 1) = 1 then RAM[address](t) = in(t − 1)
- out(t) = RAM[address](t)

Beschreibung RAM wird aus mehreren Registern konstruiert und ermöglicht den wahlfreien Zugriff auf jeden Speicherort
Adressierung Jeder Speicherort wird durch eine eindeutige Adresse identifiziert, was direkten Zugriff unabhängig von der physikalischen Position ermöglicht
Hierarchischer Aufbau Kleinere RAM-Einheiten kombinieren sich zu grösseren Speicherbänken für skalierbare und effiziente Speicherlösungen

3.3. Hack-PC

Sie verstehen den Aufbau des Hack-PC und können sein Verhalten bei laufendem Takt erklären.

Program Counter (PC)

Chip Name PC
Eingänge in[16], inc, load, reset
Ausgänge out[16]
Funktion

- if reset(t − 1) = 1 then out(t) = 0
- else if load(t − 1) = 1 then out(t) = in(t − 1)
- else if inc(t − 1) = 1 then out(t) = out(t − 1) + 1
- else out(t) = out(t − 1)

Beschreibung Ein Counter ist ein sequentielles Element, das seinen gespeicherten Wert mit jedem Taktzyklus inkrementiert
Kontrollsignale

- reset: Setzt den PC auf 0 zurück
- load: Lädt einen neuen Wert in den PC
- inc: Inkrementiert den aktuellen Wert um 1

Anwendung Essentiell für die Kontrolle der Operationssequenz in einem Computer, besonders bei der Programmausführung

Verhalten bei laufendem Takt

Priorität der Kontrollsignale

1. reset hat höchste Priorität: Wenn reset = 1, wird der PC auf 0 gesetzt
2. load hat zweite Priorität: Wenn load = 1 (und reset = 0), wird ein neuer Wert geladen
3. inc hat dritte Priorität: Wenn inc = 1 (und reset = 0, load = 0), wird der Wert inkrementiert
4. Keine Änderung: Wenn alle Signale 0 sind, behält der PC seinen aktuellen Wert

Synchronisation Alle Zustandsänderungen erfolgen synchron mit dem Taktsignal

Verzögerung Die Eingaben zum Zeitpunkt t − 1 bestimmen die Ausgabe zum Zeitpunkt t

Schlüsselkonzepte

Taktsignal Synchronisiert Zustandsänderungen in sequentiellen Elementen
DFF Fundamentales Speicherelement mit einer Taktzyklus-Verzögerung
Register Multi-Bit-Speicherelement, gesteuert durch ein Load-Signal
RAM Hierarchische Speicherstruktur mit wahlfreiem Zugriff
Counter Sequentielles Element zur Verfolgung und Kontrolle von Sequenzen

4. Maschinensprache

4.1. Aufbau der Hackplattform

Sie verstehen den schematischen Aufbau der Hackplattform (A-Register, D-Register, RAM und ROM).

Speicherbefehle

Direkte Adressierung Laden eines wertes von einer Memory-Adresse
- `LOAD R1, 67    // R1 <- Memory[67]`

Unmittelbare Adressierung Laden der konstante direkt in das Register
- `LOADI R1, 67    // R1 <- 67`

Indirekte Adressierung Laden eines wertes an einer bestimmten Stelle, z. B. in einem Array

```
ADD R1,foo, j  // R1 <- foo + j
LOAD* R2,R1    // R2 <- Memory[R1]
STR R2, x      // x <- R2
```

Hardware-Komponenten

A-Register Kann sowohl Daten als auch Adressen speichern
D-Register Wird ausschliesslich für Datenspeicherung verwendet
RAM (Random Access Memory) Wird für Datenspeicherung verwendet
ROM (Read-Only Memory) Speichert die Programmanweisungen

Schematischer Aufbau

Funktion Die Hackplattform besteht aus einer CPU mit zwei Hauptregistern (A und D), RAM für Datenspeicherung und ROM für Programmspeicherung
Datenfluss Die CPU liest Anweisungen aus dem ROM, führt Berechnungen mit den Registern durch und speichert Daten im RAM
Adressierung Das A-Register wird verwendet, um auf spezifische Speicheradressen im RAM zuzugreifen

4.2. Hack-Maschinensprache

Sie kennen die Spezifikation der Hack-Maschinensprache (A-Befehle und C-Befehle) und können einfache Programme in mnemonischer Hack-Maschinensprache lesen und schreiben.

A-Befehle (A-Instructions)

Syntax `@value`
Binär `0 vvvvvvvvvvvvvvv` (MSB = `0` -> A-Befehl, dann 15 x `v`)
Funktion Setzt das A-Register auf einen spezifischen 15 Bit Wert
Verwendung Wird verwendet, um Adressen oder Konstanten zu laden
Beispiele
- `@100` : Setzt A-Register auf 100
- `@16384` : Setzt A-Register auf die Bildschirmspeicheradresse

C-Befehle (C-Instructions)

Syntax `dest = comp; jump`, wobei in der regel immer nur dest oder jump in kombination mit comp verwendet wird
Binär `1 1 1 a cccccc ddd jjj`
- MSB = `1` -> C-Befehl, OP-Code
- danach 2 Bits mit `1`, welche aber keinen Einfluss haben
- `a` -> Gibt an, ob Werte aus `A` oder `M` geholt werden sollen
- `cccccc` 6 x `c` -> Was soll gemacht werden, ALU hat 6 Steuerbits
- `ddd` -> Destination Feld, wo soll das Ergebnis hin
- `jjj` -> Jump-Feld, wo hin soll gesprungen werden

Komponenten
- dest: Gibt an, wo das Ergebnis der Berechnung gespeichert wird
- comp: Gibt an, welche Berechnung durchgeführt wird
- jump: Gibt an, welcher bedingte Sprung durchgeführt wird

Beispiele
- `D=A` : Kopiert den Wert des A-Registers in das D-Register
- `M=D+1` : Speichert D+1 an der durch A-Register adressierten Speicherstelle
- `D=M` : Lädt den Wert aus der durch A-Register adressierten Speicherstelle in das D-Register

Zielregister (dest)

Mögliche Ziele
- M: Speicherstelle, die durch das A-Register adressiert wird
- D: D-Register
- A: A-Register
- Kombinationen: MD, AM, AD, AMD

d1	d2	d3	mnemonisch	Zielort
0	0	0	null	Der Wert wird nirgendwo gespeichert
0	0	1	M	Memory[A] (Speicherregister)
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	A-Register und Memory[A]
1	1	0	AD	A-Register und D-Register
1	1	1	AMD	A-Register, Memory[A] und D-Register

Berechnungen (comp)

Konstanten 0, 1, −1
Register A, D, M
Unäre Operationen !A, !D, !M, -A, -D, -M
Binäre Operationen D+A, D+M, D-A, D-M, A-D, M-D, D&A, D&M, D|A, D|M

c1	c2	c3	c4	c5	c6	mnemonisch wenn a = 0	a = 1
1	0	1	0	1	0	0	
1	1	1	1	1	1	1	
1	1	1	0	1	0	−1	
0	0	1	1	0	0	D	
1	1	0	0	0	0	A	M
0	0	1	1	0	1	!D	
1	1	0	0	0	1	!A	!M
0	0	1	1	1	1	-D	
1	1	0	0	1	1	-A	-M
0	1	1	1	1	1	D + 1	
1	1	0	1	1	1	A + 1	M + 1
0	0	1	1	1	0	D - 1	
1	1	0	0	1	0	A - 1	M - 1
0	0	0	0	1	0	D + A	D + M
0	1	0	0	1	1	D - A	D - M
0	0	0	1	1	1	A - D	M - D
0	0	0	0	0	0	D & A	D & M
0	1	0	1	0	1	D | A	D | M

Sprungbefehle (jump)

- Entweder nächste Ausführung im Programm holen und ausführen
- Alternativ eine Anweisung holen, welche sich an einer anderen Stelle im Programm befindet (JMP)

JMP Unbedingter Sprung
JEQ Sprung, wenn gleich Null
JGT Sprung, wenn grösser als Null
JLT Sprung, wenn kleiner als Null
JGE Sprung, wenn grösser oder gleich Null
JLE Sprung, wenn kleiner oder gleich Null
JNE Sprung, wenn nicht gleich Null

j1	j2	j3	mnemonisch	Wirkung
out < 0 | out = 0 | out > 0				
0	0	0	null	Kein Sprung
0	0	1	JGT	Sprung, wenn out > 0
0	1	0	JEQ	Sprung, wenn out = 0
0	1	1	JGE	Sprung, wenn out ≥ 0
1	0	0	JLT	Sprung, wenn out < 0
1	0	1	JNE	Sprung, wenn out ≠ 0

j1	j2	j3	mnemonisch	Wirkung
1	1	0	JLE	Sprung, wenn out ≤ 0
1	1	1	JMP	Sprung in jedem Fall

Beispielprogramm: Summe von 1 bis 100

```
@i
M=1 // Initialisiere i auf 1
@sum
M=0 // Initialisiere sum auf 0
(LOOP)
@i
D=M // D = i
@100
D=D-A // D = i - 100
@END
D;JGT // Wenn D > 0, springe zu END
@i
D=M // D = i
@sum
M=D+M // sum = sum + i
@i
M=M+1 // i = i + 1
@LOOP
0;JMP // Springe zu LOOP
(END)
@END
0;JMP // Endlosschleife
```

Vordefinierte Symbole

Virtuelle Register Vereinfachung der Assemblerprogrammierung, `R0` bis `R15` sind fordefiniert und verweisen auf die Adressen 0 bis 15 des RAM
Vordefinierte Zeiger Die Zeiger `SP, LCL, ARG, THIS, THAT` zeigen ebenfalls auf die RAM-Adressen 0 bis 4, heisst `R2` == `ARG`
I/O-Zeiger `SCREEN` und `KBD` sind vordefiniert auf die RAM Adressen 16384 = 4000_h resp. 24576 = $(6000)_h$ und können zur Interkation mit Bildschirm und Tastatur verwendet werden
Bezeichnungssymbole Hinweis auf bestimmte Stellen im Programm, setzen von Labels im Programm, sind keine Befehle, sondern Pseudo-Befehle, werden mit `(label)` runden Klammern umschlossen, wird auch direktive genannt, kann nur ein mal definiert werden
Variable Symbole Können Namen für Variablen sein z. B. und dürfen nicht mit einem Label kollidieren, werden fortlaufend auf die RAM-Adressen ab 16 vergeben

4.3. Ein- und Ausgabe

Sie verstehen die Handhabung der Ein- und Ausgabe bei der Hackplattform und können mit der Bildschirm Speicherabbildung umgehen.

Bildschirm (Screen)

Speicherabbildung Der Bildschirm ist ein speicherabgebildetes Gerät in der Hackplattform
Speicherbereich RAM-Adressen 16384 (0x4000) bis 24575 (0x5FFF)
Gesamtgrösse 8192 Adressen
Auflösung 256 Zeilen x 512 Spalten
Pixel pro Wort Jede 16-Bit-Speicherstelle steuert 16 horizontale Pixel
Pixeldarstellung
- Bitwert 1: Schwarzes Pixel
- Bitwert 0: Weisses Pixel

Wörter pro Zeile 512 ÷ 16 = 32 Wörter

Bildschirm-Speicherlayout

Speicherorganisation Jede Zeile besteht aus 32 Wörtern a 16 Bit
Adressberechnung Für Zeile r und Spalte c (in Pixeln):
- Wortadresse = $16384 + (r \times 32) + (c \div 16)$
- Bitposition im Wort = $c \bmod 16$

Beispiel Erstes Pixel (Zeile 0, Spalte 0)
- Adresse: 16384
- Bitwert: 1 (MSB)

Beispiel: Erstes Pixel einschalten

```
@16384
M=1 // Schaltet das erste Pixel ein
```

Beispiel: Zweites Pixel einschalten

```
@16384
M=2 // Schaltet das zweite Pixel ein
```

Beispiel: Pixel 3 - 40 einschalten

- Einschalten des Pixel in der 3. Zeile und 40. Spalte
- `RAM[16384 + 3:32 +40/16] = RAM[16384 +96 +2] = RAM [16482]` Register ermitteln
- `Bit (40%16) = 8` von LSB zu MSB zählend codiert, also 8. Bit von «rechts» wird eingeschaltet

Beispiel: Horizontale Linie zeichnen

```
@16384
M=-1 // Setzt alle 16 Pixel des ersten Wortes auf weiss
@16385
M=-1 // Setzt die nächsten 16 Pixel auf weiss
// Wiederholen bis zum Ende der ersten Zeile (32 Mal insgesamt)
```

Tastatur (Keyboard)

Speicherabbildung Die Tastatur ist ein speicherabgebildetes Gerät
Speicheradresse RAM-Adresse 24576 (0x6000)
Wertdarstellung ASCII-Code der aktuell gedrückten Taste
Kein Tastendruck Wert ist 0

Taste	Code	Taste	Code
newline	10000000	page up	10001000
backspace	10000001	page down	10001001
left arrow	10000010	insert	10001010
up arrow	10000011	delete	10001011
right arrow	10000100	esc	10001100
down arrow	10000101	F1	10001101
home	10000110	...	
end	10000111	F12	10011000

Beispiel: Tastatureingabe verarbeiten

```
(LOOP)
@24576
D=M // Lese Tastatureingabe in D-Register
@LOOP
D;JEQ // Wenn keine Taste gedrückt, zurück zu LOOP
// D enthält jetzt den ASCII-Code der gedrückten Taste
```

(LOOP) Label, das den Anfang einer Schleife markiert
@24576 A-Befehl, der das A-Register auf die Tastaturadresse setzt
D=M C-Befehl, der den Wert aus der durch A-Register adressierten Speicherstelle in das D-Register lädt
@LOOP A-Befehl, der das A-Register auf die LOOP-Label-Adresse setzt
D;JEQ C-Befehl mit Sprungbedingung, der zu LOOP springt, wenn D gleich Null ist
Funktionsweise Das Programm wartet in einer Schleife, bis eine Taste gedrückt wird (D ≠ 0), dann wird der ASCII-Code in D gespeichert

HACK-Dateien

Binärcodedateien
- Bestehen aus Textzeilen
- Jede Zeile ist folge von 16 0 und 1 ASCII-Zeichen, welche eine einzelne Maschienensprache kodieren
- haben Extension `.hack`

Assemblersprache
- Beinhaltet Textzeilen, von denen jede ein Befehl oder eine Symboldeklaration darstellt
- Haben extension `.asm`

Befehl Eine A- oder eine C-Anweisung
A-Befehl `@100`
C-Befehl `D = D - A`
(Symbol) Zuweisung eines labels auf die Speicherstelle des nächsten Befehls, Psreudobefehl, z. B. `(LOOP)`
Konstanten Nicht negativ, werden in Dezimaler Notation geschrieben, z. B. `@100`
Symbole Beliebige Folge von Buchstaben, Ziffern, Unterstrich, Punkt, Dollarzeichen und Dopplepunkt sein, welche nicht mit Ziffer beginnt, z. B. `@END`
Kommentare Beginnen mit `//` und werden irnoriert
Whitespace Leerzeichen und leere Zeilen werden Ignoriert
Gross- und Kleinschreibung
- Mnemonics müssen in Grossbuchstaben geschrieben werden
- an den anderen Stellen wird zwischen gross und kleinschreibun unteschieden
- Konvenstion ist, Grossbuchstaben für Bezeichnungen, Kleinbuchstaben für Variabelnamen

5. von Neumann

5.1. Grundprinzip

Sie kennen das Grundprinzip des von Neumann Rechners und können speziell den Aufbau der Hackplattform erklären.

- feste Hardwareplatform
- fester Befehlssatz
- Befehle wie Bausteine Kombinieren, um komplexer Programmcode zu erstellen

Stored Program Konzept Dieselbe Hardware kann jedes Mal, wenn sie mit einem anderen Programm geladen wird, ein völlig anderes Verhalten zeigen.

Von Neumann Maschiene

- 1 Memory, in welchem Anweisungen und Daten vorgehalten werden
- CPU mit ALU und Register
- Bestandteile der **CPU**

  ALU Ausführung von Arithmetischen und logischen Operationen, z. B. Addieren, Bits maniplieren, schauen ob Zahl positiv, ...

  Register Lokale speicherung von Ergebnissen, um effizienter zu arbeiten wie jedes Mal in den SPeicher zu schreiben und von dort zu lesen, register hat dabei jeweils die Kapazität von einem Wort

  Steuereinheit
  - Befehl ist 16-Bit Binär-Wort, welcher vor der Ausführung dekodiert werden muss
  - Steuereinheit dekodiert befehl und gibt infos an ALU, Register und Speicher weiter
  - Ebenfalls zuständig, zu ermitteln welcher Befehl als nächstes ausgeführt werden soll

Register Speicherzugriffe sind eine langsame Angelegeneheit, weshalb spezifische Register für bestimmte Aufgaben geeignet sind

Datenregister Zur Kurzzeitspeicherung innerhalb der CPU

Adressregister Beinhaltet mögliche Adresse für den Speicherzugriff aus einer früheren Anweisung

Programmzählregister / Programm Counter Hier wird die Adresse der nächsten Instruktion im ROM abgespeichert
- Wenn die aktuelle Anweisung keinen Sprung enthält, wird der PC inkrementiert
- Wenn es ein `goto` gibt, lädt die CPU die Adresse n in den PC

Ein- und Ausgabe Durch Speicherabbildung / Memory mapping werden Geräte über Speicher zugänglich gemacht

Aufbau der HACK-Platform
- 16 Bit von Neumann maschiene
- Bestandteile
  - eine CPU
  - zwei Speichermodule, 1 x befehls, 1 x Datenspeicher
  - Zwei Memorymapped IO/Geräte (Bildschirm & Tastatur)
- Ausgang des PC ist mit Adresseingang des ROM verbunden, heisst ROM-Chip gibt immer das Wort `ROM[PC]` aus, also Inhalt des Speicherplatzes auf welchen die Adresse des PC zeigt

5.2. Fetch und Execute

Sie verstehen das zyklische Execute und Fetch Verhalten der Hackplattform und können dies den entsprechenden Hardwarebestandteilen zuordnen.

FDES Cycle Normaler zyklus einer CPU

Fetch Laden der nächsten Instruktion aus dem Memroy

Decode Dekodieren der Instruktion um die nächste Aktion festzustellen

Execute Ausführen der Instruktion

Store Speichern des Ergebnisses der Aktion

Hack-Platform

Ausführen (Execute) Unterschiedliche Bit-Teile der aktuellen Anweisung werden gleichzeitig and verschiedene Chips im Computer weitergeleitet
- MSB = 0: Adressbefehl -> A-Register wird auf die im Befehl eingebettet 15-Bit-Konstante gesetzt
- MSB = 1: Rechenbefehl -> a-, c-, d- und j-Bits werden als Steuerbits behandelt welche die ALU- und Register veranlassen zu rechnen

Holen (Fetch) Laden des nächsten Befehls, abhängig von den Sprungbits des aktuellen Befehls sowie der ALU-Ausgabe
- Wenn Sprung ausgeführt wird, dann wird der PC auf den Wert des A-Registers gestellt
- Sonst wird der PC einfach um eins erhöht

5.3. Hack-CPU

Sie verstehen die Architektur der Hack CPU und können erklären, wie und wo die Befehle der Hack-Maschinensprache decodiert werden.

Interface

Eingänge
- `inM[16]` : Inhalt von RAM[A], Memorywerte welche aktuell eingespiesen werden, addressierung mit aktuellem Wert des A-Registers
- `instruction[16]` : Anweisung zur Auführung, aus dem ROM, veranlassen die Decodierung
- `reset` : Signalisiert, ob aktuelles Programm neugestartet werden soll

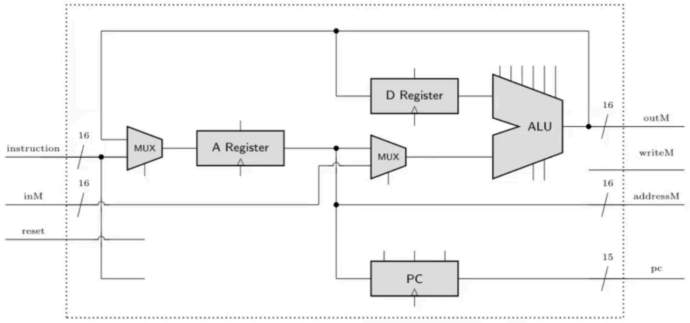
Ausgänge
- `outM[16]` : Ausgabe des M-Wertes, 16 Bit, welcher unter der mit `addressM[15]` angegebenen Addresse bei vorhandenem `writeM` -Bit geschrieben wird
- `writeM` : Gibt ann, ob der wert geschrieben werden soll oder nicht
- `addressM[15]` : Steuerung des entsprechenden RAM-Registers
- `pc[15]` : Adresse der nächsten Instruktion, auf welche der Programmcounter gesetzt werden soll

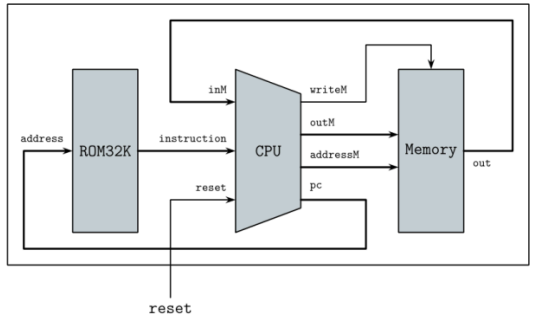
Konkreter Dekodierungsablauf

Befehlsdekodierung Herausfinden, was der Befehl bedeutet
- Das 16 Bit Wort im befehlseingang kann entweder A- oder C-Befehl sein
- MSB gibt ann ob A (MSB = 0) oder C (MSB = 1)
- C-Befehl: `1 xx a cccccc ddd jjj`, wobei a sagt, ob aus A-Register oder M, also dem Memory gelesen werden soll, c die steuerbits der ALU sind, d ermöglichen, bestimmte Stellen des Ergebnisses zu speichern und j die jumpinstruktionen sind, siehe auch oben wo das ganze detaillierter spezifiziwert ist
- A-Befehl: `0 xxxxxxxxxxxxxxx`, wobei x einfach als konstante zu interpretieren ist

Ausführen von Befehlen Den verschiedenen Teilen des Computers signalisieren, was sie tun sollen, um den Befehl auszuführen
- i, a, c, d und j werden gleichezeitig an verschiedene Teile der Architektur weitergeleitet
- Unterschiedliche Chips tun dann, was sie sollen um A- oder C- Befehl auszuführen

Abrufen des nächsten Befehls Herausfinden, welcher Befehl als nächstes ausgeführt werden soll
- CPU ermittelt auch die Position des nächsten Befehls und gibt diese über `pc` aus
- Wenn Sprung, dann `PC(t) = A(t-1)`, sonst `PC(t) = PC(t-1) + 1`
- Steuerlogik wird wie folgt implementiert
  - Ausgang A-Register mit Eingang PC verbinden
  - PC akzeptiert diesen Wert aus A-Register, wenn
    - j-Bit des aktuellen Befehls die Sprungbedinung angeben
    - Die ALU-Ausgangsstatusbits anzeigen, dass diese Bedingung erfüllt ist





6. Assembler

6.1. Maschinencode-Übersetzung

Sie können mnemonischen Hack-Maschinencode in zugehörigen Hack-Binärcode übersetzen und umgekehrt.

Befehle

A-Befehl
- **symbolisch:** `@value`

- **binär:**

```
0 v v v v v v v v v v v v v v v
(v = 0 oder 1)
```

- Jedes Assemblerprogramm darf **vordefinierte Symbole** für `value` verwenden:
  - `R0, R1, ..., R15` stehen für die Hack-RAM Adressen 0, 1, ..., 15.
  - `SP, LCL, ARG, THIS, THAT` stehen für die Hack-RAM Adressen 0, 1, 2, 3, 4.
  - `SCREEN` und `KBD` stehen für die Hack-RAM Adressen 16384 und 24576.
- Jedes Symbol `xxx` für `value`, das in einem Assemblerprogramm auftaucht, nicht vordefiniert ist und nicht an anderer Stelle durch eine Label Deklaration `(xxx)` definiert ist, wird als *Variable* behandelt. Die erste Variable, die in einem Programm auftaucht, wird auf `RAM[16]` abgebildet, die zweite auf `RAM[17]`, und so weiter.

C-Befehl

- **symbolisch:** `dest = comp; jump`
  - Entweder Feld `dest` oder `jump` kann leer sein.
  - Wenn `dest` leer ist, wird das `=` weggelassen;
  - Wenn `jump` leer ist, wird das `;` weggelassen.
- **binär:**

```
1 1 1 a c c c c c c d d d j j j
(a, c, d, j = 0 oder 1)
```

Übersetzungstabellen

`comp`

c1	c2	c3	c4	c5	c6	mnemonisch für a = 0	für a = 1
1	0	1	0	1	0	0	
1	1	1	1	1	1	1	
1	1	1	0	1	0	D	
0	0	1	1	0	0	D	
1	1	0	0	0	0	A	M
0	0	1	1	0	1	!D	
1	1	0	0	0	1	!A	!M
0	0	1	1	1	1	-D	
1	1	0	1	1	1	-A	-M
0	1	1	1	1	1	D + 1	
1	1	0	1	1	1	A + 1	M + 1
0	0	1	1	1	0	D - 1	
1	1	0	0	1	0	A - 1	M - 1
0	0	0	0	1	0	D + A	D + M
0	1	0	0	1	1	D - A	D - M
0	0	0	1	1	1	A - D	M - D
0	0	0	0	0	0	D & A	D & M
0	1	0	1	0	1	D | A	D | M

`dest`

d1	d2	d3	mnemonisch	Zielort
0	0	0	null	Der Wert wird nirgendwo gespeichert
0	0	1	M	Memory [A] (Speicherregister)
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	A-Register und Memory [A]
1	1	0	AD	A-Register und D-Register
1	1	1	AMD	A-Register, Memory [A] und D-Register

`jump`

j1	j2	j3	mnemonisch	Wirkung
out < 0	out = 0	out > 0		
0	0	0	null	Kein Sprung
0	0	1	JGT	Sprung, wenn out > 0
0	1	0	JEQ	Sprung, wenn out = 0
0	1	1	JGE	Sprung, wenn out ≥ 0
1	0	0	JLT	Sprung, wenn out < 0
1	0	1	JNE	Sprung, wenn out ≠ 0
1	1	0	JLE	Sprung, wenn out ≤ 0
1	1	1	JMP	Sprung

6.2. Symboltabelle

Sie können erklären wie ein Assembler diese Übersetzung systematisch macht und verstehen die Bedeutung der Symboltabelle.

> Symbolverwaltung ist die wichtigste funktion des assemblers

- der Assembler
  - Zerlegt die Anweisungen in ihre Felder
  - Erzeugt Bit-Codes für diese
  - löst die enthaltenen Referenzen mit Hilfe der Symboltabelle in numerische Werte auf
  - setzt die sich ergebenden Binärcodes zu einer Zeichenkette von 0 und 1 zusammen
  - Schreibt diese Zeichenfolge in die Ausgabedatei
- Symbole
  - es ist möglich, dass symbole verwendet werden, bevor diese definiert sind
  - Um dies zu ermöglichen, wird das Programm in folgenden Schritten durchlaufen

    Initialisierung Assember erstellt Symboltabelle und initialisiert diesem it allen vordefinierten Symbolen

    Erster Durchlauf
    - Assembler durchläuft das Programm Zeile für Zeile
    - Wenn eine A- oder eine C-Anweisung kommt, wird dieser Counter um 1 erhöht, bei kommentaren oder Labels ändern sich diese nicht
    - Bei jeder Label-Deklaration `(xxx)` wird diese in die Symboltabelle eingetragen mit der **aktuellen Zeilennummer + 1**, also ROM-Addresse der nächsten Anweisung im Programm
    - Am ende haben wir in der Symboltabelle alle Label-Symbole des Programms mit ihren Sprungaddressen

    Zweiter Durchlauf
    - Assembler geht erneut duech das gesammte Programm
    - Wenn eine `A-Anweisung` mit symbolischem Verweis auftaucht, also z. B. `@xxx`, wobei `xxx` ein Symbol und keine Zahl ist, sucht der Assembler in der Symboltabelle nach `xxx`
    - Wenn `xxx` gefunden wird, wird es durch seinen numerischen Wert ersetzt
    - Wenn `xxx` nicht gefunden wird, dann muss es sich um eine neue Variable handeln und es
      - Wird `xxx value` in die Tabelle eingefügt, wobei `value` die nächste verfügbare Adresse für Variablen im RAM ist
      - Wird die Übersetzung des Befehls unter verwendung dieser Adresse abgeschlossen

6.3. Parser- und Code-Modul

Sie verstehen die unterschiedlichen Aufgaben des Parser Moduls und des Code Moduls.

Parser

- Ermöglicht Zeilenweisen Zugriff auf die Eingabe
- zerlegt symbolische Anweisungen in ihre zugrundeliegenden Komponenten
- Beispiele
  - `@7` wird mit Aufruf von `symbol()` `7` retourniert
  - Für Anweisung `(LOOP)` wird durch `symbol()` entsprechend `LOOP` retourniert
  - Für `D=D+1;JLE` wird bei `dest()`, `comp()` oder `jump()` entsprechend `D`, `D+1` oder `JLE` retourniert

Routine	Argumente	Rückgabe	Funktion
Konstruktor	Input Datei	keine	Öffnet die Eingabedatei und bereitet sie zum Parsen vor.
hasMoreLines	keine	boolean	Gibt es weitere Zeilen in der Eingabe?
advance	keine	keine	Überspringt Leerraum und Kommentare, falls erforderlich. Liest die nächste Anweisung aus der Eingabe und macht sie zur aktuellen Anweisung. Diese Routine sollte nur aufgerufen werden, wenn hasMoreLines wahr ist. Anfänglich gibt es keine aktuelle Anweisung.
instructionType	keine	A_INSTRUCTION C_INSTRUCTION L_INSTRUCTION	Gibt den Typ der aktuellen Anweisung zurück: - A_INSTRUCTION für @xxx, wobei xxx eine Dezi-

			malzahl oder ein Symbol ist. - C_INSTRUCTION für dest=comp;jump - L_INSTRUCTION für (xxx), wobei xxx ein Symbol ist.
symbol	keine	string	Für die aktuelle Anweisung (xxx) wird das Symbol xxx und für @xxx wird das Symbol oder die Dezimalzahl xxx als Zeichenkette zurückgegeben. Die Routine sollte nur aufgerufen werden, wenn der instructionType A_INSTRUCTION oder L_INSTRUCTION ist.
dest	keine	string	Gibt den symbolischen dest Teil der aktuellen C-Anweisung zurück (8 Symbole). Sollte nur aufgerufen werden, wenn instructionType C_INSTRUCTION ist.
comp	keine	string	Gibt den symbolischen comp Teil der aktuellen C-Anweisung zurück (28 Symbole). Sollte nur aufgerufen werden, wenn instructionType C_INSTRUCTION ist.
jump	keine	string	Gibt den symbolischen jump Teil der aktuellen C-Anweisung zurück (8 Symbole). Sollte nur aufgerufen werden, wenn instructionType C_INSTRUCTION ist.

Code

- Übersetztung symbolischer Hack Mnemonics in Binärcodes, gemäss Sprachspezifikationen
- Alle n-Bit-Codes werden als Zeicchenketten mit 0 und 1 zurückgegeben
- Beispiele
  - `dest("DM")` gibt `"011"`
  - `comp("A+1")` gibt `"0110111"`
  - `comp("M+1")` gibt `"1110111"`
  - `jump("JNE")` gibt `"101"`

Routine	Argumente	Rückgabe	Funktion
dest	string	3-bit als string	Gibt den Binärcode des dest Mnemonics zurück.
comp	string	7-bit als string	Gibt den Binärcode des comp Mnemonics zurück.
jump	string	3-bit als string	Gibt den Binärcode des jump Mnemonics zurück.

HackAssembler

- Nutzt *Code* und *Parser* zur Zusammensetzung des schlussendlichen Binarcodes
- In der Grundversion kein Support für Referenzen / Labels
- Für symbolische Verweise braucht es eine Symboltabelle, in welcher symbole auf RAM doer ROM-Adressen gemappt werden, kann mit einer Hash-Table realisiert werden
  - API sieht wie folgt aus

Routine	Argumente	Rückgabe	Funktion
Konstruktor	keine	keine	Erstellt eine neue leere Symboltabelle und initialisiert die vordefinierten Symbole.
addEntry	symbol (string)naddress (int)	keine	Fügt das Paar `<symbol,address>` in die Tabelle.
contains	symbol (string)	boolean	Enthält die Symboltabelle das angegebene Symbol?
getAddress	symbol (string)	int	Gibt die mit dem Symbol verbundene Addresse zurück.

7. Virtuelle Maschine

7.1. Stack-Arithmetik

> Sie verstehen die Stack-Arithmetik und können die Veränderung des Stacks bei der Abarbeitung eines einfachen Hack-VM-Programmes beschreiben (Speichersegmente, globaler Stack, Arbeitsstack, gespeicherte Frames).

Ein **Stack** ist eine lineare Datenstruktur nach dem **LIFO-Prinzip** (Last-In-First-Out). Er unterstützt zwei Grundoperationen: `push` (Wert oben auf den Stack legen) und `pop` (obersten Wert entfernen).

Verarbeitungsbefehle

- Stack Befehle
  `push <segment> <index>` Scheibt wert von `segment[index]` auf den Stack
  - Befehl `push x` ist `RAM[SP] = x; SP++`

  `pop <segment> <index>` Entnimmt den obersten Stackwert und speichert ihn in `segment[index]`
  - Befehl `pop x` ist `SP--; x = RAM[SP]`

  - wobei
    `segment` Aus `argument, local, static, this, that, pointer, temp` stammt
    `index` Eine nicht negative ganze Zahl ist
- Arithmetische Befehle
  `add` $x + y$, ganzzahlige Addition im 2er Komplement
  `sub` $x - y$, ganzzahlige Subtraktion im 2er Komplement
  `neg` $-x$, Arithmetische Negation im 2er Komplement
- Vergleichsbefehle
  `eq` $x == y$, Gleichheit
  `gt` $x > y$, grösser als
  `lt` $x < y$, kleiner als
- Logische Befehle
  `and` $x \wedge y$, bitweise UND
  `or` $x \vee y$, bitweise ODER
  `not` $\neg y$, bitweises Nicht

Speichersegmente

Die Hack-VM verwendet verschiedene Speichersegmente:

Segment	Verwendung
Argument	Funktionsargumente
Local	Lokale Variablen
Static	Statische Variablen, werden ab addresse 16 über den assembler variablen zugeordnet, beim erstellen der Symboltabelle
Constant	Konstante Werte
This / That	Zeiger auf Objektfelder
Pointer	Basisadressen von this und that, basisadresse ist immer 3, `this` ist index `0`, `that` index `1`, also adressen `ram[3+0]` für `this` und `ram[3+1]` für `that`
Temp	Temporäre Speicherung, 8 segmente welche mit `temp` fix die basisadresse `5` hat, und dann über indices `0` - `7` auf die entsprechenden Segmente zugegriffen werden kann

RAM-Nutzung (globaler Stack & Frames)

Der Hack-RAM besteht aus 32K 16-Bit-Wörtern. Die VM nutzt folgende Adressbereiche:

RAM-Adresse	Verwendung
0 – 15	16 virtuelle Register
16 – 255	Statische Variablen
256 – 2047	Stack

Die Pointer-Register, die den Zustand des Stacks und der gespeicherten Frames verwalten:

Name	Ort	Verwendung
SP	RAM[0]	Adresse der Stack-Spitze
LCL	RAM[1]	Basisadresse Local-Segment
ARG	RAM[2]	Basisadresse Argument-Segment
THIS	RAM[3]	Basisadresse This-Segment
THAT	RAM[4]	Basisadresse That-Segment
TEMP	RAM[5–12]	Temp-Segment
R13–R15	RAM[13–15]	Freie Register für Assembly

Veränderung des Stacks (Beispiel)

Ausgehend von einem leeren Stack:

Operation	Stack-Inhalt	Effekt
push 15 (x)	15	SP erhöht

Operation	Stack-Inhalt	Effekt
push 7	15, 7	SP erhöht
`lt`	false	15 < 7 ist falsch
push 8 (y)	false, 8	SP erhöht
push 8	false, 8, 8	SP erhöht
`eq`	false, true	8 == 8 ist wahr
`or`	true	true OR false = true
`pop d`	(leer)	true ⟶ Speicherstelle d

Arithmetische Operationen nehmen die Operanden vom Stack, führen die Berechnung aus und legen das Ergebnis zurück auf den Stack.

CMD	Berechnung
`add`	$x + y$ (Zweierkomplement)
`sub`	$x - y$ (Zweierkomplement)
`neg`	$-y$ (Negation)
`eq`	$x == y$
`gt`	$x > y$
`lt`	$x < y$
`and`	x and y (bitweise)
`or`	x or y (bitweise)
`not`	not y (bitweise)

7.2. VM-Code-Übersetzung

> Sie können einfachen Hack-VM-Code (Verarbeitungsbefehle und Kontrollbefehle) gemäss Spezifikation in gültigen mnemonischen Hack-Maschinencode übersetzen.

Verarbeitungsbefehle

push constant 5

```
@5      // Konstante 5 laden
D=A     // Wert 5 in D speichern
@SP     // Stack-Pointer
A=M     // Adresse der Stack-Spitze
M=D     // Wert 5 an die Spitze schreiben
@SP
M=M+1   // Stack-Pointer erhöhen
```

pop local 0

```
@SP
AM=M-1  // SP verringern, Spitze adressieren
D=M     // obersten Wert in D
@LCL    // Basisadresse Local-Segment
A=M     // Basisadresse holen
M=D     // Wert in local 0 schreiben
```

add

```
@SP
AM=M-1  // SP verringern, neue Spitze
D=M     // oberster Wert in D
A=A-1   // zum zweiten Wert
M=M+D   // Summe an Stelle speichern
```

sub

```
@SP
AM=M-1
D=M
A=A-1
M=M-D
```

neg

```
@SP
A=M-1
M=-M
```

Kontrollbefehle (Verzweigungen)

label LOOP_START

Beschriftet aktuelle Stelle im code, nur markierte Stellen können angesprungen werden.

```
(LOOP_START)
```

goto LOOP_START

Bewirkt einen unbedingten Sprung zum angegebenen Label

```
@LOOP_START  // Label adressieren
0;JMP        // unbedingter Sprung
```

if-goto LOOP_START

- Bedingter Sprung
- Oberster Wert wird vom Stack gepoppt, sofern nicht Null wird am angegebene Label weiter gearbeitet
- Sprungziel muss in der selben funktion sein

```
@SP
AM=M-1       // SP verringern, Spitze adressieren
D=M          // Spitze in D speichern
@LOOP_START
D;JNE        // Sprung wenn D != 0
```

Funktionsbefehle

`function <name> <nVars>` Markiert beginn einer Funktion namens `<name>` mit `nVars` Variablen

```
// fügt Funktions-Einstiegslable in den Code
(f)

// initialisiert nVars lokale Variablen auf 0
push 0
push 0
...
push 0
```

`call <funcName> <nArgs>` Ruft die Funktion namens `<funcName>` auf und teilt mit, dass `<nArgs>` Argumente vor aufrufen auf den Stack geschoben werden

```
// erzeugt ein Label und schiebt es auf den Stack
push rAdd

// speichert LCL, ARG, THIS und THAT des Caller's
push LCL
push ARG
push THIS
push THAT

// positioniert ARG und LCL neu
ARG = SP - 5 - nArgs
LCL = SP

// übergibt die Kontrolle an den Callee
goto f

// fügt das Rückkehrlabel in den Code
(rAdd)
```

`return` Übergibt die Ausführung an den Befehl, der im Code der Funktion, die die aktuelle Funktion aufgreufen hat, unmittelbar auf den `call`-Befehl folgt

```
frame = LCL  // frame ist temp Variable

// speichert Rückkehradresse in temp Variable rAdd
rAdd = *(frame-5)

// setzt die Werte für den Caller
*ARG = pop()  // Rückgabewert
SP = ARG + 1  // Stackpointer
THAT = *(frame-1)
THIS = *(frame-2)
ARG  = *(frame-3)
LCL  = *(frame-4)

// geht zu der Rückkehradresse zurück
goto rAdd
```

Caller-Sicht

- Bevor ich eine Funktion aufrufe, muss ich so viele Argumente (nArgs) auf den Stack legen, wie der Callee erwartet.
- Als nächstes rufe ich den Callee mit dem Befehl `call FileName.functionName nArgs` auf.
- Nachdem der Callee zurückkehrt, sind die Argumente, die ich vor dem Aufruf auf den Stack geschoben habe, verschwunden, und ein Rückgabewert (der immer

vorhanden ist) erscheint oben auf dem Stack. Abgesehen von dieser Änderung ist mein Arbeitsstack genau derselbe wie vor dem Aufruf [VM].
- Nach der Rückkehr des Callee sind alle meine Speichersegmente genau dieselben wie vor dem Aufruf [VM], außer dass sich der Inhalt meines `static` Segments geändert haben kann und das `temp` Segment undefiniert ist.

Callee-Sicht

- Bevor ich mit der Ausführung beginne, wurde mein `argument` Segment mit den vom Caller übergebenen Argumentwerten initialisiert, und mein `local` Segment für lokale Variablen wurde zugewiesen und mit Nullen initialisiert.
- Mein `static` Segment wurde auf das statische Segment der VM-Datei gesetzt, zu der ich gehöre, und mein Arbeitsstack ist leer. Die Speichersegmente `this`, `that`, `pointer` und `temp` sind noch undefiniert [VM].
- Bevor ich zurückkehre, muss ich einen Rückgabewert auf den Stack legen.

7.3. VM-Translator

> Sie verstehen den Übersetzungsvorgang des Hack-VM-Translators und können die Aufgabe des Parsers und des CodeWriters beschreiben.

Der **VM-Translator** übersetzt VM-Code in mnemonischen Hack-Maschinencode. Der Vorgang besteht aus zwei Hauptkomponenten: dem **Parser** und dem **CodeWriter**.

Aufgaben des Parsers

Der Parser liest und analysiert die VM-Befehle:

1. **VM-Datei lesen:** Liest die Eingabedatei mit den VM-Befehlen.
2. **Befehle tokenisieren:** Zerlegt jeden Befehl in seine Grundbestandteile (Befehlstyp, Argumente, Speichersegmente).
3. **Befehlstypen erkennen:** Identifiziert den Typ jedes Befehls (arithmetisch, push, pop, label, goto, if-goto, function, call, return).
4. **Zugriff bereitstellen:** Bietet Methoden, um auf den aktuellen Befehl und seine Komponenten zuzugreifen und zum nächsten Befehl weiterzugehen.

Aufgaben des CodeWriters

Der CodeWriter übersetzt die geparsten VM-Befehle in Hack-Assembly-Code:

1. **Ausgabedatei initialisieren:** Initialisiert die Out-Datei.
2. **Assembly-Code schreiben:** Übersetzt jeden vom Parser gelieferten Befehl in die entsprechenden Hack-Instruktionen.
3. **Befehlstypen behandeln:** Enthält Methoden für die verschiedenen Befehlsarten (arithmetisch, Speicherzugriff, Programmfluss, Funktionsbefehle).
4. **Eindeutige Labels erzeugen:** Generiert eindeutige Labels für korrekte Verzweigungen und Funktionsbehandlung.

Einstieg

- eine Datei muss `Main.jack` heissen
- Eine Funktion darin muss `main` heissen
- Per konvention wird `Sys.init` als erstes aufgerufen, welche dann `Main.main` ausführt

VM Zuordnung

Symbol	Verwendung
SP	Dieses Symbol zeigt auf die RAM-Adresse, die direkt auf die Adresse folgt, die den obersten Wert des Stacks enthält.
LCL, ARC, THIS, THAT	Diese Symbole verweisen jeweils auf die Basis RAM-Adressen der virtuellen Segmente local, argument, this und that der aktuell laufenden VM Funktion.
Xxx.i (statische Variablen)	Jeder Verweis auf `static i` in der Datei Xxx.vm wird in das Assembler Symbol Xxx.i übersetzt. Der Assembler weist diese symbolischen Variablen dem RAM zu, beginnend bei Adresse 16.
`functionName$label` (Ziele von goto)	Sei foo eine Funktion in der Datei Xxx.vm. Die Verarbeitung jedes label bar Befehls innerhalb von foo erzeugt das Symbol `Xxx.foo$bar` und injiziert es in den Assemblercode. Bei der Übersetzung von goto bar und if-goto bar Befehlen in foo muss das Label `Xxx.foo$bar` anstelle von bar verwendet werden.
functionName (Einstiegspunkte)	Bei der Verarbeitung jedes function foo Befehls in der Datei Xxx.vm wird ein Symbol Xxx.foo in den Assemblercode injiziert, das den Einstiegspunkt in den Funktionscode markiert. Danach übersetzt der Assembler dieses Symbol in die physikalische Adresse, an der der Funktionscode beginnt.
`functionName$ret.i` (Rücksprungpunkte)	Bei der Verarbeitung jedes call Befehls in foo wird ein Symbol `Xxx.foo$ret.i` in den Assemblercode injiziert, wobei i eine laufende Ganzzahl ist (ein solches Symbol wird für jeden Aufruf erzeugt). Dieses Symbol wird verwendet, um die Rücksprungadresse im Code des Callers zu setzen. Danach übersetzt der Assembler das Symbol in die physische Speicheradresse des Befehls, der unmittelbar auf den Aufrufbefehl folgt.
R13, R14, R15	Diese Symbole können für jeden Zweck verwendet werden. Wenn der VM Übersetzer beispielsweise Assemblercode generiert, der Low-Level Variablen für die temporäre Speicherung verwenden muss, können diese nützlich sein.

8. Die Hochsprache

8.1. Jack-Syntax

> Sie kennen die Jack-Syntax und können Jack-Programme lesen und einfache Jack-Programme selbst schreiben.

Jack ist eine einfache, objektbasierte Hochsprache, inspiriert von Java und C++, jedoch **ohne Vererbung**. Sie dient als Grundlage zum Bau eines Compilers und eines Betriebssystems.

Ein Jack-Programm besteht aus **Klassen**, die jeweils in einer separaten Datei gespeichert werden. Jede Klasse kann **statische Variablen** und **Feldvariablen**, **Konstruktoren**, **Methoden** und **Funktionen** enthalten.

- Symbole
  - `()` Gruppieren von Audrücken und einschliessen von Argumentlisten und Parameterlisten
  - `[]` Indizierung von Arrays
  - `{}` Gruppierung von Programmeinheiten und Anweisungen
  - `,` Trennen von Variabeln in Listen
  - `;` Abschliessen / terminieren einer Anweisung
  - `=` Zuweisung und Vergleiche
  - `.` Zugriffsoperator für Mitglieder einer Klasse
  - `+ - * / & | ~ < >` Als Operatoren
- Reservierte Wörter
  - **Programmkomponenten** class, constructor, methode, function
  - **Primitive Datentypen** int, boolean, char, void
  - **Variabeldeklaration** var, static, field
  - **Anweisungen** let, do, if, else, while, return
  - **Konstante Werte** true, false, null
  - **Objektreferenz** this
- Konstanten
  - **Zahlenkonstanten** Werte zwischen 0 und 32767
    - negative zahlen sind keine Konstanten sondern ausdrücke, auf welche `-` operator angewendet wird
  - **Zeichenketten** Werden durch `"` eingeschlossen
    - Alles ausser «newline» und `"` sind erlaubt
  - **Boolsche Konstanten** `true` oder `false`
  - **Null-Referenz** als `null`
- Bezeichner
  - Bezeichner sind aus einer beliebig langen Folge von Buchstaben (A–Z, a–z), Ziffern (0–9) und dem Unterstrich `_` zusammengesetzt. Das erste Zeichen muss ein Buchstabe oder der Unterstrich `_` sein. Die Sprache unterscheidet zwischen Groß- und Kleinschreibung: x und X werden als unterschiedliche Bezeichner behandelt.

```
class name {
    /* Vor den Unterprogrammdeklarationen stehen immer die
Variablen */

    Deklarationen der Feldvariablen
    Deklarationen der statischen Variablen

    /* Dann folgen in beliebiger Reihenfolge Konstruktoren,
Methoden und Funktionen */

    Deklarationen der Unterprogramme
}
```

- Variablen

Typ	Beschreibung	Deklariert in	Geltungsbereich
Statische Variable	*static type varName;* Von jeder statischen Variablen existiert eine Kopie, die von allen Unterprogrammen der Klasse gemeinsam genutzt wird (wie private statische Variablen in Java)	Klasse	Klasse, in der sie deklariert ist
Feldvariable	*field type varName;* Jedes Objekt (Instanz der Klasse) hat eine private Kopie der Feldvariablen (wie Objektvariablen in Java)	Klasse	Klasse, in der sie deklariert ist
Lokale Variable	*var type varName;* Wird zu Beginn der Ausführung des Unterprogramms erstellt und bei der Rückkehr des Unterprogramms verworfen	Unterprogramm	Unterprogramm, in der sie deklariert ist
Parametervariable	Stellen die Argumente dar, die an das Unterprogramm übergeben werden. Werden wie lokale Variablen behandelt, deren Werte durch den Caller des Unterprogramms initialisiert werden	Unterprogramm	Unterprogramm, in der sie deklariert ist

- Anweisungen

Anweisung	Syntax	Beschreibung
let	let varName = expression;n let varName[expression1] = expression2;	Eine Zuweisungsoperation.n Zugelassen sind statische Variablen, Feldvariablen,n lokale Variablen oder Parametervariablen.
if	if (expression) {n statements1;n } else {n statements2;n }	Typische if-Anweisung mit optionaler else-Klausel.n Die geschweiften Klammern sind obligatorisch,n auch wenn es sich bei statements um eine einzelne Anweisung handelt.
while	while (expression) {n statements;n }	Typische while-Anweisung.n Die geschweiften Klammern sind obligatorisch,n auch wenn es sich bei statements um eine einzelne Anweisung handelt.
do	do function-call;n do method-call;	Wird verwendet, um eine Funktion oder Methode aufzurufen,n wobei der zurückgegebene Wert (falls vorhanden) ignoriert wird.
return	return expression;n return;	Wird verwendet, um einen Wert aus einem Unterprogramm zurückzugeben.n Die zweite Form muss von void-Unterprogrammen verwendet werden.n Konstruktoren müssen den Wert this zurückgeben.

- Ausdrücke
  - Es gibt folgende
    Konstanten konstante Werte
    Variabelname Kann statische, feld- lokale- oder parametervariable sein
    `this` referenz auf aktuelles Objekt
    Array Element Zugriff mit `arr[exp]`, wobei `arr` ein Variablennamen von Typ Array ist
    subroutine Unterprogrammaufruf welcher nicht `void` zurück gibt
  - Kann mit `-` oder `~` (bitweise) negiert werden
  - Kann mit `exp <op> exp` verarbeitet werden, wobei `<op>` aus `+, -, *, /, &, |, <, >` ist
  - Kann mit klammern umschlossen werden
- Operatorenpriorität
  - Es gibt keine Operatiorenpriorität
  - Ausdrücke in Klammern werden zuerst ausgewertet
- Unterprogrammaufruf
  - Aufrufen einer Funktion, methode oder konstruktor
  - Entweder als `klasse.funktionsname`, `variable.methode` oder `this.some` `subroutineName(exp1, exp2, ..., expn)`
- (De)Konstruktion
  - Konstruktion in zwei schritten
    1. Referenzvariable auf Zeiger auf Objekt deklarieren
    2. Konstruktuor aufrufen, einer per default mit dem namen `new` muss vorhanden sein `let some = className.new(123)`
  - Wenn objekte nicht mehr gebraucht werden -> `Memory.deAlloc(c)`, es gibt keinen GC

Hello-World-Programm

```
class Main {
  function void main() {
    do Output.printString(&quot;Hello World&quot;);
    do Output.println(); // Neue Zeile
    return; // return ist notwendig
  }
}
```

Die Klasse `Main` enthält eine einzige Funktion `main`, die `Hello World` ausgibt.

Beispiel: Durchschnitt berechnen

Das folgende Programm liest Ganzzahlen von der Tastatur, speichert sie in einem Array und berechnet ihren Durchschnitt:

```
/** Inputs integers and computes their average. */
class Main {
  function void main() {
    var Array a;
    var int length, i, sum;
    let i = 0;
    let sum = 0;
    let length = Keyboard.readInt(&quot;How many numbers?
&quot;);
    let a = Array.new(length); // Array konstruieren
    while (i &lt; length) {
      let a[i] = Keyboard.readInt(&quot;Enter a number: &quot;);
      let sum = sum + a[i];
      let i = i + 1;
    }
    do Output.printInt(sum / length);
    do Output.println();
    return;
  }
}
```

Wichtige Syntax-Elemente: Variablen werden mit `var` deklariert, Zuweisungen erfolgen mit `let`, Methoden- und Funktionsaufrufe ohne Rückgabewert mit `do`, und jede Funktion muss mit `return` abschliessen.

8.2. Grammatiken und Chomsky-Hierarchie

> Sie können für einfache formale Sprachen eine Grammatik angeben und kennen die Chomsky-Hierarchie (reguläre, kontextfreie, kontextsensitive und rekursiv aufzählbare Grammatiken).

Formale Grammatiken können mithilfe von **Produktionsregeln** spezifiziert werden. Die **Chomsky-Hierarchie** gliedert Grammatiken in vier Typen mit zunehmender Mächtigkeit:

Grammatik Bestehen aus Terminalen und Nicht Terminalen

- In `<asdf>` nicht terminale
- EInfach so `lorem` -> Wort / terminal

```
<Satz> -> <Subjekt> <Prädikat> <Objekt>
<Subjekt> -> <Artikel> <Adjektiv> <Substatntiv>
<Artikel> -> Der | Die | Das
...
```

- Besteht aus
  Variablemenge V - Nonterminale
  Terminalalphabet Σ mit $V \cap \Sigma = \emptyset$
  Produktionsregeln P
  Startvariable S mit $S \in V$
- Produktion P hat die Form $I \in (V \cup \Sigma)^+ \rightarrow r \in (V \cup \Sigma) *$

Alphabet Endliche Menge von Symbolen

Zeichen

Typ	Grammatik	Beschreibung
Typ 3	Regulär	Können durch reguläre Ausdrücke dargestellt werden., *Auf der rechten Seite dürfen immer nur zwei Symbole stehen, zuerst ein Terminal, danach ein Non-Terminal, rechte Seite darf zudem das leere Wort sein*
Typ 2	Kontextfrei	Geeignet zur Beschreibung der Syntax von Programmiersprachen., *linke seite darf nur aus einem einzigen variablen symbol bestehen, rechts ist man frei*
Typ 1	Kontextsensitiv	Mächtiger; können gewisse Einschränkungen natürlicher Sprachen beschreiben., *rechte seite ist länger als die linke seite*
Typ 0	Rekursiv aufzählbar	Am mächtigsten; können alle berechenbaren Sprachen beschreiben., *links darf nicht leer sein, sonst ist man komplett fei, mächstigste sprache*

Jack-Programme können mit einer **kontextfreien Grammatik (Typ 2)** analysiert werden.

Das Wortproblem ist für Typ 2 und 3 lösbar.

CNF

CNF Chomsky-Normalform, es geht, eine L2-Sprache so umzuformen, dass man nur noch «Variable -> terminal» oder «Variable -> Variable Variable» (2 Variablen)

- Syntaxbäume von CNF-Grammatiken sind Binärbäume

Übersetzen in CNF in 4 Schritten möglich

1. Alle Regeln wegnehmen, welche auf der rechten Seite das leere Wort haben, dazu prüfen, was durch entfernen der entsprechenden Variable möglich ist
2. Entfernen der Produktionen, welche auf der rechten Seite nur eine einzige Variable haben, indem wir alle damit möglichen Produktionen mit nehmen
3. Entfernen von Kombinationen aus Nonterminal und Terminal, indem für Terminale eine weitere neue Variable eingesetzt wird, welche dann direkt auf nur ein Terminal projeziert
4. Wir erstetzen alle diese, welche mehr als zwei Variablen haben, also zu lange sind, wir führen also erneut eine variable ein, welche ein Subset der bisherigen Produktion ist, und setzen es in die ursprungsproduktion ein (Siehe S_2 unten)

Start	Schritt 1	Schritt 2	Schritt 3	Schritt 4
S → AB S → ABA A → aA A → a B → Bb B → ε	S → AB S → A S → ABA A → aA A → a B → Bb B → b	S → AB S → aA S → a S → ABA S → AA A → aA A → a B → Bb B → b	S → AB S → V_a A S → a S → ABA S → AA A → V_a A A → a B → B V_b B → b V_a → a V_b → b	S → AB S → V_a A S → a S → S_2 A S_2 → AB S → AA A → V_a A A → a B → B V_b B → b V_a → a V_b → b

Nachdem wir eine Sprache in CNF gebraucht haben, ist ihr Syntax-Baum Binär, was er vorher nicht war. Das ganze ist nun durch Dynamische Programmierung lösbar.

8.3. CYK-Algorithmus

> Sie können den CYK-Algorithmus durchführen und damit für kontextfreie Sprachen entscheiden, ob ein Wort zur Sprache gehört oder nicht.

Der **CYK-Algorithmus** (Cocke-Younger-Kasami) bestimmt, ob eine gegebene Zeichenkette zu einer kontextfreien Sprache gehört. Er nutzt **dynamische Programmierung** zum effizienten Parsen.

- **Eingabe:** Eine Zeichenkette und eine kontextfreie Grammatik in **Chomsky-Normalform**.
- **Ausgabe:** Ob die Zeichenkette von der Grammatik erzeugt werden kann.

Wir versuchen dabei den CNF-Baum von unten nach oben aufzubauen

Beispiel

Wir prüfen, ob das Wort `abacba` zur Sprache gehört. Die Grammatik ist definiert als:

$$S \rightarrow SA \mid a$$
$$A \rightarrow BS$$
$$B \rightarrow BB \mid BS \mid b \mid c$$

Schritt 0 & 1: Wort eintragen und Länge-1-Teilwörter

Zuerst wird das Wort in die erste Zeile geschrieben. Dann übersetzt man jedes Zeichen 1:1 in sein nicht-terminales Gegenstück.

0.	a	b	a	c	b	a
1.	S	B	S	B	B	S

Schritt 2: Länge-2-Teilwörter

Man betrachtet jeweils ein Paar benachbarter Zeichen und prüft, ob sich das Paar in ein Nicht-Terminal übersetzen lässt. Falls ja, wird es eingetragen, sonst die leere Menge ∅. Beispiel: `BS` findet sich bei `A` und `B`, daher werden beide eingetragen.

0.	a	b	a	c	b	a
1.	S	B	S	B	B	S
2.	∅	A, B	∅	B	A, B	

Schritt 3: Länge-3-Teilwörter

Nun vergleicht man Werte über Kreuz: Für die erste Zelle bildet man die Teilwörter aus `S` (Zeile 1) und `A, B` (Zeile 2), also `SA, SB`. `SA` lässt sich zu `S` übersetzen, `SB` nicht. Zusätzlich wird ∅ aus Zeile 2 mit dem zweiten `S` aus Zeile 1 verglichen. Das Ergebnis beider Vergleiche kommt in die Ergebniszelle.

0.	a	b	a	c	b	a
1.	S	B	S	B	B	S
2.	∅	A, B	∅	B	A, B	
3.	S	B	∅	A, B		

Schritt 4: Länge-4-Teilwörter

Sobald ein ∅ auftritt, kann für dieses Paar kein Treffer gefunden werden, da stets zwei Teile X und Y benötigt werden. Bei Länge 4 gibt es jedoch je drei Paare zu vergleichen (`SB` , `∅∅` , `SB`). Für keines existiert ein Nicht-Terminal, daher ∅.

0.	a	b	a	c	b	a
1.	S	B	S	B	B	S
2.	∅	A, B	∅	B	A, B	
3.	S	B	∅	A, B		
4.	∅	B	S			

Schritt 5: Länge-5-Teilwörter

Das Muster wird fortgeführt.

0.	a	b	a	c	b	a
1.	S	B	S	B	B	S
2.	∅	A, B	∅	B	A, B	
3.	S	B	∅	A, B		
4.	∅	B	S			
5.	∅	A, B				

Schritt 6: Länge-6-Teilwörter

Im letzten Schritt benötigen wir ein `S` als Ergebnis, um zu bestätigen, dass das Wort zur Grammatik gehört.

0.	a	b	a	c	b	a
1.	S	B	S	B	B	S
2.	∅	A, B	∅	B	A, B	
3.	S	B	∅	A, B		
4.	∅	B	S			
5.	∅	A, B				
6.	S					

Warum suchen wir nach `S`?

Die letzte Zelle (6,1) repräsentiert die gesamte Zeichenkette `abacba`. Das Vorhandensein von `S` zeigt, dass die gesamte Zeichenkette aus dem **Startsymbol** `S` abgeleitet werden kann — das Wort gehört also zur Sprache.

Enthält die unterste linke Zelle `S` (auch neben anderen Nicht-Terminalen), bestätigt dies mindestens eine Ableitungssequenz, die von `S` ausgeht. Das Vorhandensein **anderer** Nicht-Terminale allein bestätigt hingegen nicht, dass die Zeichenkette von `S` ableitbar und somit Teil der Sprache ist.

9. Compiler

9.1. Parse-Bäume

> Sie können mit der Jack-Grammatik Jack-Code prüfen und Parse-Bäume lesen bzw. zeichnen.

Ein **Parse-Baum** (Ableitungsbaum) repräsentiert die grammatikalische Struktur eines Programms. Beim **Parsing** wird eine Tokenfolge anhand der Grammatikregeln analysiert, wobei ein Baum aufgebaut wird, der die Struktur des Programms widerspiegelt.

Lexikalische Analyse

Bevor ein Parse-Baum entstehen kann, zerlegt die **lexikalische Analyse** eine Zeichenfolge in **Tokens**.

Für den Code:

```
while (count &lt;= 100) {
  count++;
}
```

erzeugt der lexikalische Analysator die Tokens: `while`, `(`, `count`, `<=`, `100`, `)`, `{`, `count`, `++`, `;` und `}`. Diese Tokens sind die Bausteine der weiteren syntaktischen Analyse.

Jack-Grammatik (vereinfacht)

Die Jack-Grammatik ist **kontextfrei** und definiert die Syntax für Klassen, Subroutinen, Variablendeklarationen und Anweisungen:

```
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)* ';'
```

Beispiel eines Parse-Baums

Für die Anweisung `let a = 2 + 3;` ergibt sich folgender Parse-Baum:

```
letStatement
├── 'let'
├── varName: a
├── '='
├── expression
│   ├── term: 2
│   ├── op: +
│   └── term: 3
└── ';'
```

Der rekursive Abstiegsparser (recursive descent) arbeitet top-down und behandelt damit die verschachtelte Struktur der Sprache.

9.2. Jack zu VM-Code

> Sie können einfachen Jack-Code in Hack-VM-Code übersetzen.

Beispiel 1: Variablendeklaration

```
class Main {
  function void main() {
    var int a;
    let a = 5;
  }
}
```

wird übersetzt zu:

```
function Main.main 1
push constant 5
pop local 0
```

Beispiel 2: Arithmetische Operation

```
class Main {
  function void main() {
    var int a;
    var int b;
    let a = 5;
    let b = a + 3;
  }
}
```

wird übersetzt zu:

```
function Main.main 2
push constant 5
pop local 0
push local 0
push constant 3
add
pop local 1
```

Beispiel 3: If-Anweisung

```
class Main {
  function void main() {
    var int a;
    let a = 5;
    if (a &gt; 3) {
      let a = a - 1;
    }
  }
}
```

wird übersetzt zu:

```
function Main.main 1
push constant 5
pop local 0
push local 0
push constant 3
gt
if-goto IF_TRUE
goto IF_FALSE
label IF_TRUE
push local 0
push constant 1
sub
pop local 0
label IF_FALSE
```

Beispiel 4: While-Schleife

```
class Main {
  function void main() {
    var int a;
    let a = 0;
    while (a &lt; 5) {
      let a = a + 1;
    }
  }
}
```

wird übersetzt zu:

```
function Main.main 1
push constant 0
pop local 0
label WHILE_EXP
push local 0
push constant 5
lt
if-goto WHILE_TRUE
goto WHILE_FALSE
label WHILE_TRUE
push local 0
push constant 1
add
pop local 0
goto WHILE_EXP
label WHILE_FALSE
```

Ausdrucksauswertung

Ausdrücke werden mittels **rekursiver Post-Order-Traversierung** des Parse-Baums in stackbasierten VM-Code übersetzt. Der Ausdruck `x + g(2, y, -z) * 5` wird zu:

```
push x
push 2
push y
push z
neg
call g
push 5
call multiply
add
```

9.3. Jack-Compiler

> Sie verstehen den Übersetzungsvorgang des Jack-Compilers und können die Aufgabe des Tokenizers, der SymbolTabelle, des VMWriters und der CompilationEngine beschreiben.

Der **JackCompiler** arbeitet auf einer Quelle (Datei oder Verzeichnis mit `.jack` -Dateien). Für jede Eingabedatei erzeugt er eine `.vm` -Ausgabedatei und nutzt dafür die Module **JackTokenizer**, **SymbolTable**, **VMWriter** und **CompilationEngine**.

Aufgaben der Module

Modul	Aufgabe
JackTokenizer	Zerlegt den Eingabestrom in Jack-Tokens, entfernt Kommentare und Whitespaces und liefert Typ und Wert des aktuellen Tokens (z. B. via `hasMoreTokens` , `advance` , `tokenType`).
SymbolTable	Verwaltet Bezeichner-Geltungsbereiche und -Eigenschaften. Nutzt zwei Hash-Tabellen: eine für den Klassen-Scope und eine für den Subroutinen-Scope. Speichert Name, Typ, Art (kind) und Index.
VMWriter	Gibt VM-Befehle gemäss der VM-Befehlssyntax in eine Datei aus.
CompilationEngine	Liest die Eingabe vom JackTokenizer und schreibt die Ausgabe in den VMWriter. Organisiert als eine Reihe von `compileXxx()` -Routinen, die je ein syntaktisches Element der Jack-Sprache behandeln (z. B. `compileClass` , `compileSubroutine` , `compileLet` , `compileWhile`).

Beispiel einer Symboltabelle

Die Symboltabelle erfasst alle vom Programm eingeführten Bezeichner und ordnet ihnen Typ, Art und Index zu:

Name	Type	Kind	#
nAccounts	int	static	0
id	int	field	0
name	String	field	1
balance	int	field	2
sum	int	argument	0
status	boolean	local	0

Der Index wird dabei erhöht, sofern bereits eine Variable mit gleichem Typ und Kind existiert. Bei der method level symboltabelle wird als nulltes argument mit dem namen this immer die eigene Instanz eingefügt.

Übersetzungsvorgang am Beispiel

Die Anweisung `let a = 2 + 3;` durchläuft folgende Schritte:

1. **Tokenisierung:** Zerlegung in Tokens: `let` , `a` , `=` , `2` , `+` , `3` , `;` .
2. **Parsing:** Aufbau des Parse-Baums (`letStatement` mit `expression` aus `term 2` , `op +` , `term 3`) gemäss den Grammatikregeln.
3. **Symboltabellen-Verwaltung:** Erfassung der Variable `a` (Type `int` , Kind `local` , Index `0`).
4. **Codegenerierung:** Übersetzung des Parse-Baums in VM-Code.

Der finale VM-Code lautet:

```
push constant 2  // Konstante 2 auf den Stack
push constant 3  // Konstante 3 auf den Stack
add              // oberste zwei Stack-Elemente addieren
pop local 0      // Ergebnis in lokale Variable a
```

Dieser Ablauf folgt dem modularen **Analyse-Synthese-Paradigma**, das dem Bau der meisten Compiler zugrunde liegt.

10. Betriebssystem

Das Jack-Betriebssystem (OS) bietet eine High-Level- Schnittstelle zur Interaktion mit der Hardware. Es umfasst acht Hauptklassen, die jeweils unterschiedliche Dienste bereitstellen, welche für die reibungslose Ausführung von Jack-Programmen entscheidend sind.

10.1. Jack-Klassen

> Sie kennen die Dienste der 8 Jack-Klassen des Betriebssystems.

Klasse	Dienste
Memory	Speicherverwaltung: Zugriff und Manipulation des RAM. `alloc` fordert einen Speicherblock an, `deAlloc` gibt zuvor allozierten Speicher wieder frei.
Array	Vereinfacht Erstellung und Manipulation von Arrays. Arrays sind dynamisch alloziert; `Array.new` erzeugt ein Array gegebener Grösse.
Math	Mathematische Operationen: Addition, Subtraktion, Multiplikation, Division sowie komplexere Funktionen wie Quadratwurzel oder trigonometrische Funktionen.
String	Erstellung und Manipulation von Strings, z. B. `String.new` , `String.appendChar` , `String.setChar` . Ermöglicht u. a. String-Vergleiche.
Output	Textausgabe auf dem Bildschirm: `Output.printString` , `Output.printInt` , `Output.printChar` , `Output.moveCursor` (Cursor positionieren) und `Output.clearScreen` (Anzeige löschen).
Screen	Low-Level-Zeichenoperationen auf Pixelebene: `Screen.drawPixel` , `Screen.drawLine` , `Screen.drawRectangle` .
Keyboard	Erfassung von Benutzereingaben über die Tastatur: `Keyboard.keyPressed` und `Keyboard.readKey` .
Sys	System-Operationen wie Programminitialisierung und -beendigung: `Sys.init` startet das OS, `Sys.halt` stoppt die Ausführung.

10.2. Algorithmen

> Sie verstehen die Algorithmen für die Multiplikation, die Division, das Zeichnen einer Linie und können das Heap Management und die Textausgabe beschreiben.

Multiplikation

Die Multiplikation wird ohne Hardware-Multiplikator durch wiederholtes Addieren realisiert.

Grundidee:

- Zerlege den Multiplikator bitweise.
- Für jedes gesetzte Bit wird der entsprechend verschobene Multiplikand addiert.

- Nach jedem Schritt wird der Multiplikand verdoppelt und der Multiplikator halbiert.

Pseudocode:

```
result = 0
while y > 0:
    if y mod 2 == 1:
        result += x
    x = x * 2
    y = y / 2
```

Komplexität: O(log n)

Division

Die Division basiert auf rekursiver Halbierung.

Idee:
- Teile den Dividend zunächst durch 2.
- Berechne rekursiv den Quotienten.
- Verdopple das Zwischenergebnis.
- Falls der Rest groß genug ist, wird 1 addiert.

Pseudocode:

```
divide(x, y):
    if y > x:
        return 0

    q = divide(x / 2, y)
    q = 2 * q

    if x - q * y >= y:
        q = q + 1

    return q
```

Komplexität: O(log n)

Linienzeichnen

Das Ziel ist das Zeichnen einer Geraden zwischen zwei Punkten.

Gegeben:
- Startpunkt (x1, y1)
- Endpunkt (x2, y2)

Steigung:

```
a = (y2 - y1) / (x2 - x1)
```

Für jeden x-Wert wird der entsprechende y-Wert berechnet:

```
y = y1 + a * (x - x1)
```

Je nach Steigung:
- |a| <= 1: Iteration über x
- |a| > 1: Iteration über y

Jeder berechnete Punkt wird mit drawPixel gesetzt.

Heap Management

Der Heap verwaltet dynamisch reservierten Speicher für Objekte und Arrays.

Wichtige Operationen:

alloc(size)

- Sucht einen freien Speicherblock.
- Reserviert den benötigten Speicher.
- Gibt die Startadresse zurück.

deAlloc(address)

- Gibt einen Speicherblock frei.
- Fügt ihn wieder der Freispeicherliste hinzu.

Freispeicherliste

Jeder freie Block enthält:
- Größe des Blocks
- Zeiger auf den nächsten freien Block

Ablauf:
1. Suche passenden Block.
2. Falls nötig Block aufteilen.
3. Speicher zurückgeben.
4. Beim Freigeben Block wieder einhängen.

Textausgabe

Die Textausgabe erfolgt über den Bildschirm-Speicher (Memory-Mapped I/O).

Zeichen darstellen

Jedes Zeichen besitzt:
- ASCII-/Hack-Code
- Bitmap-Font

Der Font beschreibt für jede Zeile eines Zeichens die zu setzenden Pixel.

Zeichen ausgeben

1. Zeichencode bestimmen.
2. Bitmap aus Font-Tabelle laden.
3. Pixel der Bitmap auf den Bildschirm schreiben.
4. Cursor weitersetzen.

Cursor

Der Cursor speichert:
- aktuelle Zeile
- aktuelle Spalte

Nach jedem Zeichen:
- Spalte erhöhen
- Zeilenumbruch bei Bedarf

Kernaussagen

- Multiplikation erfolgt durch Bitverschiebungen und Additionen.
- Division erfolgt durch rekursive Halbierung.
- Linien werden durch Berechnung der Zwischenpunkte gezeichnet.
- Heap Management verwaltet dynamisch reservierten Speicher mittels Freispeicherliste.
- Textausgabe verwendet Bitmap-Fonts und schreibt Pixel direkt in den Bildschirmspeicher.

10.3. Betriebssystem-Methoden

> Sie können Methoden des Betriebssystem, die in der Sprache Jack geschrieben sind, lesen und selbst die Dienste des Betriebssystems in einfachen Anwendungen einsetzen.

Die OS-Dienste werden in Jack-Anwendungen über einfache Methodenaufrufe genutzt. Das folgende Beispiel kombiniert mehrere OS-Klassen:

```
class Main {
  function void main() {
    var int a, b, sum;
    // Keyboard: Eingaben von der Tastatur lesen
    let a = Keyboard.readInt(&quot;Erste Zahl: &quot;);
    let b = Keyboard.readInt(&quot;Zweite Zahl: &quot;);
    // Math: Berechnung (implizit über VM-Befehle)
    let sum = a + b;
    // Output: Ergebnis ausgeben
    do Output.printString(&quot;Summe: &quot;);
    do Output.printInt(sum);
    do Output.println();
    return;
  }
}
```

Typische Dienstaufrufe im Überblick:

Aufruf	Dienst
`Keyboard.readInt(msg)`	Liest eine Ganzzahl ein
`Keyboard.readLine(msg)`	Liest eine Zeile als String
`Output.printString(s)`	Gibt einen String aus
`Output.printInt(i)`	Gibt eine Ganzzahl aus
`Screen.drawRectangle(...)`	Zeichnet ein Rechteck
`Memory.alloc(size)`	Alloziert einen Speicherblock
`Sys.halt()`	Beendet das Programm

Beim Lesen von OS-Methoden gilt: Sie sind selbst in Jack geschrieben und folgen denselben Syntaxregeln (`function` / `method` , `let` , `do` , `return`). Durch das Verständnis dieser Konventionen lassen sich die OS-Dienste gezielt in eigenen Anwendungen einsetzen.